UltraFAST™

# UltraFast Design Methodology Guide for the Vivado Design Suite

**UG949 (v2017.4) December 20, 2017**

XILINX
ALL PROGRAMMABLE™

# Revision History

12/20/2017: Released with Vivado® Design Suite 2017.4 without changes from 2017.3.

| Date | Version | Revision |
|---|---|---|
| 11/10/17 | 2017.3 | In Chapter 2, Board and Device Planning, updated Power Distribution System, added power budget tip in Specific Considerations for PCB Design, added Power Rail Consolidation Impacting Power, and added power budget tip in Device Power and the Overall System Design Process.<br>In Chapter 3, Design Creation, updated Use Register Replication, added XOR functions in Check Inferred Logic, updated Balance Latency, added CLOCK_LOW_FANOUT tip in Low Fanout Clocks, added Using the CLOCK_LOW_FANOUT Constraint, and updated 7 Series Device Clocking.<br>In Chapter 4, Implementation, updated Strategies and added Using Interactive Report Files.<br>In Chapter 5, Design Closure, updated Baselining the Design, added flow diagram to Analyzing and Resolving Timing Violations, added `-min_level` and `-max_level` options in Review the Logic Level Distribution, added `retiming_forward` and `retiming_backward` in Improving Logic Levels, updated Reducing Control Sets, added Prioritize Critical Logic Using the group_path Option, added Fixing Large Hold Violations Prior to Routing, added directives tip in Strategies and Directives, and added Quick option in Select Incremental Compile Directives for High Reuse Mode. |
| 05/26/2017 | 2017.1 | Updated content based on the new Vivado IDE look and feel throughout.<br>In Chapter 3, Design Creation, added details on the `ASYNC_REG` attribute in Use Register Replication, added Decomposing Deeper Memory Configurations for Balanced Power and Performance, added table and updated examples in Using the CLOCK_DEDICATED_ROUTE Constraint, added SelectIO Clocking, added note about DCP files in Planning IP Requirements, and updated example in Report Timing from or to the Port.<br>In Chapter 4, Implementation, replaced Bottom-Up Synthesis Flow with Block-Level Synthesis Strategy and updated Using Incremental Compile Flows.<br>In Chapter 5, Design Closure, updated attribute values in Verifying That No Clocks Are Missing, added details on the congestion tables in Report Design Analysis Congestion Report, added Improving the Netlist with Block-Level Synthesis Strategies, updated Reducing Control Sets, added information on `-merge_equivalent_drivers` and `-fanout_opt` options in Use Register Replication, added `-no_bufg_opt` option in Promote High Fanout Nets to Global Routing, added information on router directives in Use Alternate Placer and Router Directives, added information on the MUXF_REMAP property in Disable LUT Combining and MUXF Inference, updated Use Block-Level Synthesis Strategies, added information on automatic optimization in Limit High-Fanout Nets in Congested Areas, and updated Using Incremental Compile.<br>In Appendix A, Additional Resources and Legal Notices, removed references to Vivado Design Suite QuickTake Video: Customizing and Instantiating IP and Vivado Design Suite QuickTake Video: Designing with Vivado IP Integrator and added reference to UltraFast Design Methodology Training Course. |

XILINX
ALL PROGRAMMABLE™

# Table of Contents

## Appendix A: Additional Resources and Legal Notices

# Introduction

## About the UltraFast Design Methodology

The Xilinx® UltraFast™ design methodology is a set of best practices intended to help streamline the design process for today's All Programmable devices. The size and complexity of these designs require specific steps and design tasks to ensure success at each stage of the design. Following these steps and adhering to the best practices will help you achieve your desired design goals as quickly and efficiently as possible.

Xilinx provides the following resources to help you take advantage of the UltraFast design methodology:

- This guide, which describes the various design tasks, analysis and reporting features, and best practices for design creation and closure.
- *UltraFast Design Methodology Quick Reference Guide* (UG1231) [Ref 2], which highlights key design methodology steps in an easy-to-use, double-sided card format.
- *UltraFast Design Methodology Checklist* (XTP301) [Ref 3], which is available in the Xilinx Documentation Navigator and as a standalone spreadsheet. You can use this checklist to identify common mistakes and decision points throughout the design process.
- Methodology-related design rule checks (DRCs) for each design stage, which are available using the `report_methodology` Tcl command in the Vivado® Design Suite.
- UltraFast Design Methodology System-Level Design Flow diagram representing the entire Vivado Design Suite design flow, which is available in the Xilinx Documentation Navigator. You can click a design step in the diagram to open related documentation, collateral, and FAQs to hep get you started.

## Using This Guide

This guide provides a set of best practices that maximize productivity for both system integration and design implementation. It includes high-level information, design guidelines, and design decision trade-offs for the following topics:

- Chapter 2, Board and Device Planning: Covers decisions and design tasks that Xilinx recommends accomplishing prior to design creation. These include I/O and clock planning, PCB layout considerations, device capacity and throughput assessment, alternate device definition, power estimation, and debugging.
- Chapter 3, Design Creation: Covers the best practices for RTL definition, IP configuration and management, and constraints assignment.
- Chapter 4, Implementation: Covers the options available and best practices for synthesizing and implementing the design.
- Chapter 5, Design Closure: Covers the various design analysis and implementation techniques used to close timing on the design or to reduce power consumption. It also includes considerations for adding debug logic to the design for hardware verification purposes.

This guide includes references to other documents such as the Vivado Design Suite User Guides, Vivado Design Suite Tutorials, and Quick-Take Video Tutorials. This guide is not a replacement for those documents. Xilinx still recommends referring to those documents for detailed information, including descriptions of tool use and design methodology. For a listing of reference documents, see Appendix A, Additional Resources and Legal Notices.

***Note:*** This information is designed for use with the Vivado Design Suite, but you can use most of the conceptual information with the ISE® Design Suite as well.

## Using the UltraFast Design Methodology Checklist

To take full advantage of the UltraFast design methodology, use this guide with the *UltraFast Design Methodology Checklist* (XTP301) [Ref 3]. The checklist is available from the Xilinx Documentation Navigator or as a standalone spreadsheet.

The questions in the UltraFast Design Methodology Checklist highlight typical areas in which design decisions are likely to have downstream impact and draw attention to issues that are often overlooked or ignored. Each tab in the checklist:

- Targets a specific role within a typical design team.
- Includes common questions and recommended actions to take during each design flow step, including project planning, board and device planning, IP and submodule design, and top-level design closure.
- Includes a Documentation and Training section that lists resources related to the design flow step.
- Provides links to content in this guide or other Xilinx documentation, which offer guidance on addressing the design concerns raised by the questions.

**VIDEO:** *For a demonstration of the checklist, see the* Vivado Design Suite QuickTake Video: Introducing the UltraFast Design Methodology Checklist.

## Using the UltraFast Design Methodology DRCs

The Vivado Design Suite contains a set of methodology-related DRCs you can run using the `report_methodology` Tcl command. This command has rules for each of the following design stages:

- Before synthesis in the elaborated RTL design to validate RTL constructs
- After synthesis to validate the netlist and constraints
- After implementation to validate constraints and timing related concerns.

**RECOMMENDED:** *For maximum effect, run the methodology DRCs at each design stage and address any issues prior to moving to the next stage.*

For more information on the design methodology DRCs, see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21], and see the report_methodology Tcl command in the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 14].

## Using the UltraFast Design Methodology System-Level Design Flow Diagram

The following figure shows the various design steps and features included in the Vivado Design Suite. From the Xilinx Documentation Navigator, you can access an interactive version of this graphic in which you can click each step for links to related resources.

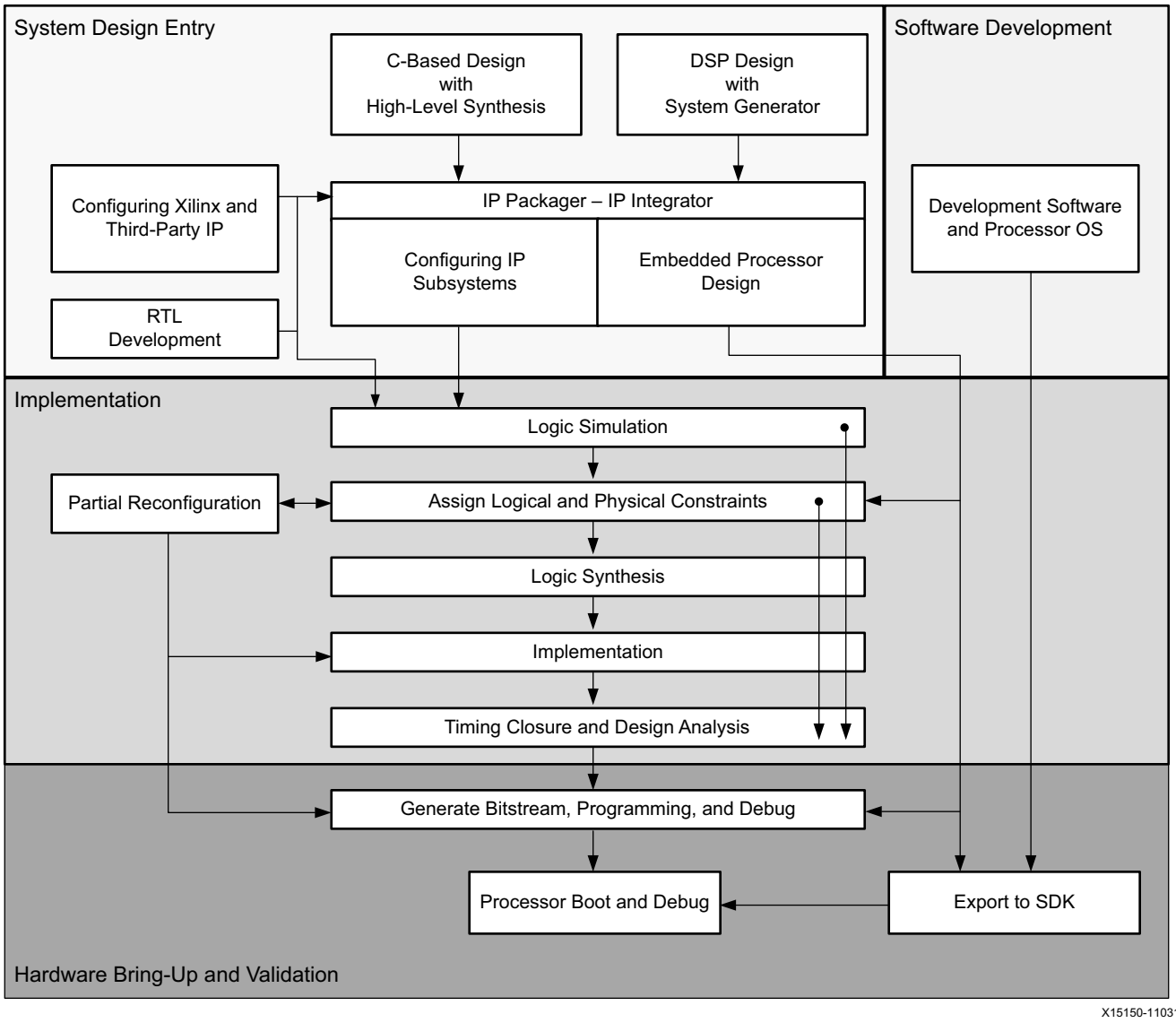



*Figure 1-1:* **UltraFast Design Methodology System-Level Design Flow**

# Understanding UltraFast Design Methodology Concepts

It is important to take the correct approach from the start of your design and to pay attention to design goals from the early stages, including RTL, clock, pin, and PCB planning. Properly defining and validating the design at each design stage helps alleviate timing closure, routing closure, and power usage issues during subsequent stages of implementation.

## Maximizing Impact Early in the Development Cycle

As shown in the following figure, early stages in the design flow (C, C++, and RTL synthesis) have a much higher impact on design performance, density, and power than the later implementation stages. Therefore, if the design does not meet timing goals, Xilinx recommends that you revisit the synthesis stage, including HDL and constraints, rather than iterating for a solution in the implementation stages only.

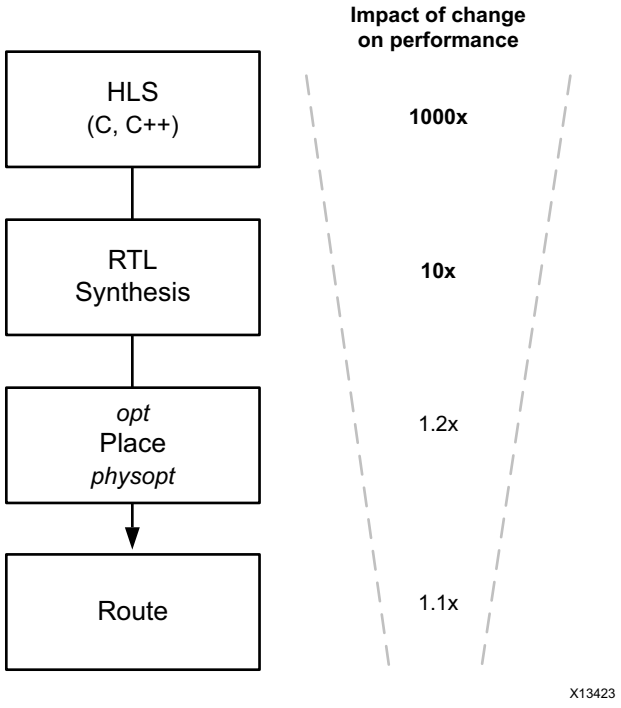



*Figure 1-2:* **Impact of Design Changes Throughout the Flow**

## Validating at Each Design Stage

The UltraFast design methodology emphasizes the importance of monitoring design budgets, such as area, power, and timing, and correcting the design from early stages as follows:

- Create optimal RTL constructs with Xilinx templates, and validate your RTL with methodology DRCs prior to synthesis.

  Because the Vivado tools use timing-driven algorithms throughout, the design must be properly constrained from the beginning of the design flow.

- Perform timing analysis after synthesis.

  To specify correct timing, you must analyze the relationship between each master clock and related generated clocks in the design. In the Vivado tools, each clock interaction is timed unless explicitly declared as an asynchronous or false path.

- Meet timing using the right constraints before proceeding to the next design stage.

  You can accelerate overall timing and implementation convergence by following this recommendation and by using the interactive analysis environment of the Vivado Design Suite.

**TIP:** *You can achieve further acceleration by combining these recommendations with the HDL design guidelines in this guide.*

The following figure shows this recommended design methodology.

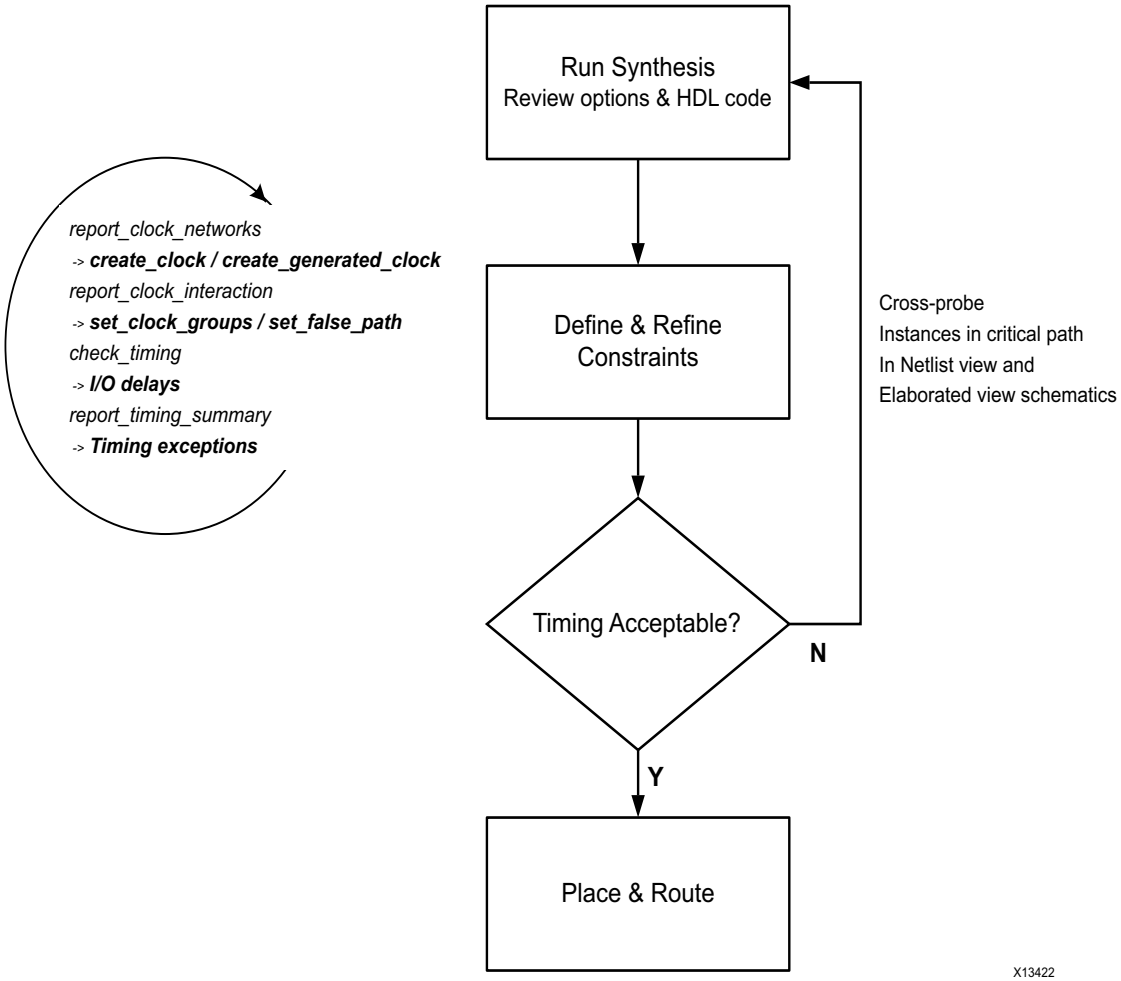



*Figure 1-3:* **Design Methodology for Rapid Convergence**

Synthesis is considered complete when the design goals are met with a positive margin or a relatively small negative timing margin. For example, if post-synthesis timing is not met, placement and routing results are not likely to meet timing. However, you can still go ahead with the rest of the flow even if timing is not met. Implementation tools might be able to close timing if they can allocate the best resources to the failing paths. In addition, proceeding with the flow provides a more accurate understanding of the negative slack magnitude, which helps you determine how much you need to improve the post-synthesis worst negative slack (WNS). You can use this information when you return to the synthesis stage with improvements to HDL and constraints.

### Taking Advantage of Rapid Validation

This guide also introduces the concept of rapid validation of specific aspects of the system architecture and micro-architecture as follows:

- In the context of system design, the I/O bandwidth is validated in-system, before implementing the entire design. Validating I/O bandwidth can highlight the need to revise system architecture and interface choices before finalizing on I/Os. For more information, see Interface Bandwidth Validation in Chapter 2.
- As part of design implementation, baselining is used to write the simplest set of constraints, which can identify internal device timing challenges. Baselining can identify the need to revise RTL micro-architecture choices before moving to the implementation phase. For more information, see Baselining the Design in Chapter 5.

## Using the Vivado Design Suite

The Vivado Design Suite has a flexible use model to accommodate various development flows and different types of designs. For detailed information on how to use the features within the Vivado Design Suite, see the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 6] and other Vivado Design Suite documentation.

### Managing Vivado Design Suite Sources with a Revision Control System

Most design teams manage their design sources and results with a commercially available revision control system. The Vivado Design Suite allows various use models for managing design and IP data. For more information on using the Vivado tools with a revision control system, see this link in the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 6].

### Upgrading to New Vivado Design Suite Releases

New releases of the Vivado Design Suite often contain updates to Xilinx IP. Carefully consider whether you want to upgrade your IP, because upgrading can result in design changes. In addition, you must follow specific rules when using IP configured with previous releases going forward. For more information, see this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 10].

# Accessing Additional Documentation and Training

This guide supplements the information in the Vivado Design Suite documentation, including user guides, reference guides, tutorials, and QuickTake videos. The Xilinx Documentation Navigator provides access to the Vivado Design Suite documentation and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.
- At the Linux command prompt, enter: `docnav`

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

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

**TIP:** *For quick access to information on different parts of the Vivado IDE, click the Quick Help button ? in the window or dialog box. For detailed information on Tcl commands, enter the command followed by* `-help` *in the Tcl Console.*

# Board and Device Planning

## Overview of Board and Device Planning

Properly planning the FPGA orientation on the board and assigning signals to specific pins can lead to dramatic improvements in overall system performance, power consumption, and design cycle times. Visualizing how the FPGA device interacts physically and logically with the printed circuit board (PCB) enables you to streamline the data flow through the device.

Failing to properly plan the I/O configuration can lead to decreased system performance and longer design closure times. Xilinx highly recommends that you consider I/O planning in conjunction with board planning.

For more information, see the following resources:

- *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) [Ref 5]
- Vivado Design Suite QuickTake Video: I/O Planning Overview

## PCB Layout Recommendations

The layout of the FPGA device on the board relative to other components with which it interacts can significantly impact the I/O planning.

### Aligning with Physical Components on the PCB

The orientation of the FPGA device on the PCB should first be established. Consider the location of fixed PCB components, as well as internal FPGA resources. For example, aligning the GT interfaces on the FPGA package to be as close to the components with which they interface on the PCB will lead to shorter PCB trace lengths and fewer PCB vias.

A sketch of the PCB including the critical interfaces can often help determine the best orientation for the FPGA device on the PCB, as well as placement of the PCB components. Once done, the rest of the FPGA I/O interface can be planned.

High-speed interfaces such as memory can benefit from having very short and direct connections with the PCB components with which they interface. These PCB traces often have to be matched length and not use PCB vias, if possible. In these cases, the package pins closest to the edge of the device are preferred in order to keep the connections short and to avoid routing out of the large matrix of BGA pins.

The I/O Planning View Layout in the Vivado IDE is useful in this stage for visualizing I/O connectivity relative to the physical device dimensions, showing both top-side and bottom-side views.

**IMPORTANT:** *For thermally-challenged designs, be aware of device placement in relation to other high-power components to minimize thermal coupling and maximize airflow. Avoid placement where the device is positioned in the exhaust of another high power component or where board heating might negatively impact the operating temperature. Xilinx recommends thermal simulation to understand how the placement and environmental conditions can affect the junction temperature of the device.*

The following figure shows the I/O Planning view layout.

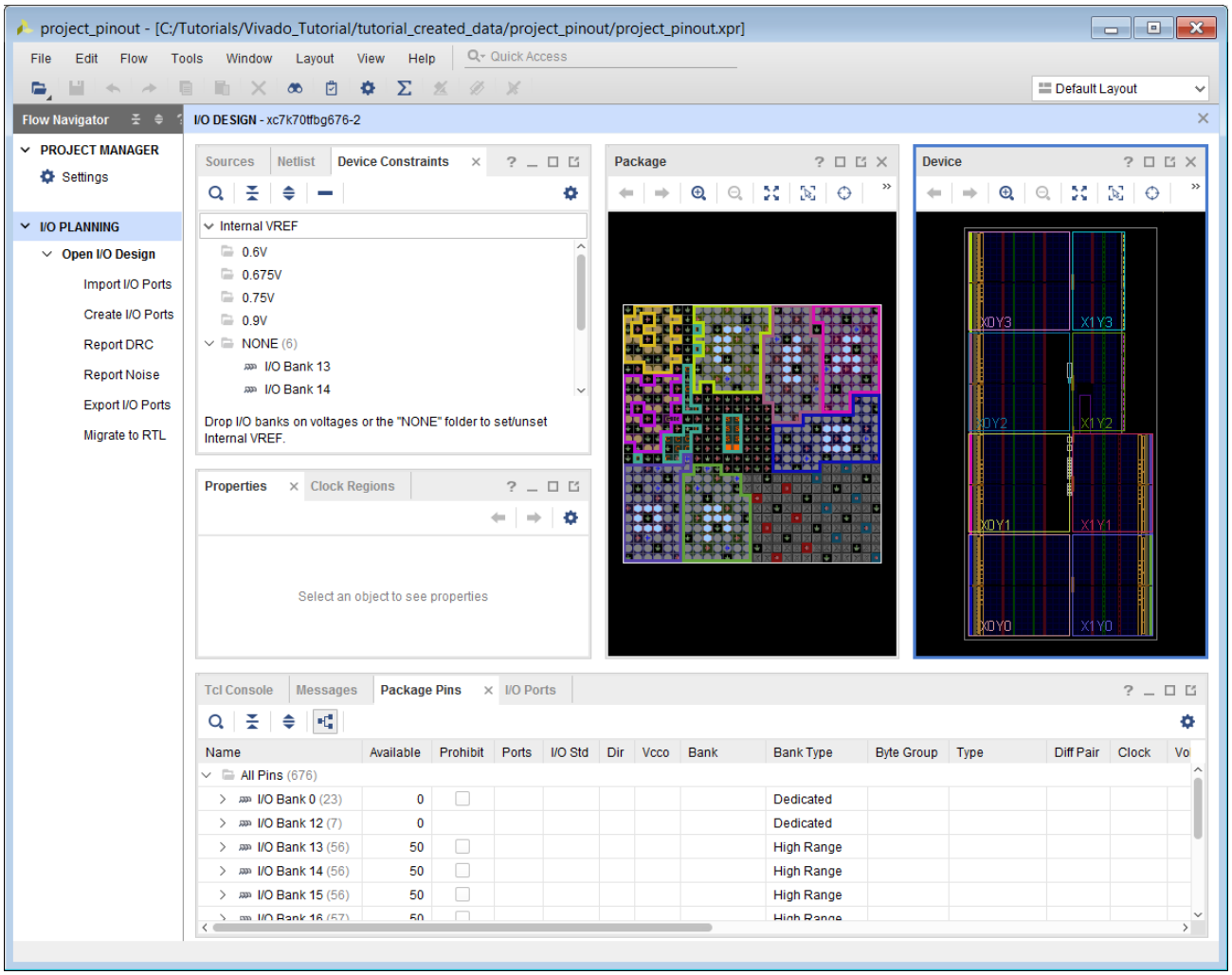



*Figure 2-1:* **I/O Planning View Layout**

## Power Distribution System

Board designers are faced with a unique task when designing a Power Distribution System (PDS) for a Xilinx device. Most other large, dense integrated circuits (such as large microprocessors) come with very specific bypass capacitor requirements. Because these devices are designed only to implement specific tasks in their hard silicon, their power supply demands are fixed and fluctuate only within a certain range.

Xilinx devices do not share this property. Devices can implement an almost infinite number of applications at undetermined frequencies, and in multiple clock domains. For this reason, it is critical that you refer to the *PCB Design Guide* [Ref 36] for your device to fully understand the device PDS.

Key factors to consider during PDS design include:

- Selecting the right voltage regulators to meet the noise and current requirements based on Power Estimation. For more information, see Power Analysis and Optimization in Chapter 5.

**TIP:** *Consider adding a shunt resistor to allow the power on each rail to be monitored.*

- Consolidating power. For more information, see this link in the *UltraScale Architecture PCB Design User Guide* (UG583) [Ref 36].
- Setting up the XADC power supply (Vrefp and Vrefn pins).
- Running power distribution network (PDN) simulation. The recommended amount of decoupling capacitors in the *PCB Design Guide* [Ref 36] for your device is based on specific device utilization and step load. If the device utilization and step load do not match your design, a PDN simulation is recommended. Running PDN simulations can help to confirm the exact amount of decoupling capacitors required to guarantee power supplies that are within the recommended operating range.

For more information on PDN simulation, see the Xilinx White Paper: *Simulating FPGA Power Integrity Using S-Parameter Models* (WP411) [Ref 51].

## Specific Considerations for PCB Design

The PCB should be designed considering the fastest signal interfacing with the FPGA device. These high-speed signals are extremely sensitive to trace geometry, vias, loss, and crosstalk. These aspects become even more prominent for multi-layer PCBs. For high-speed interfaces perform a signal integrity simulation. A board redesign with improved PCB material or altered trace geometries may be necessary to obtain the desired performance.

Xilinx recommends following these steps when designing your PCB:

1. Review the following device documentation:
   - *PCB Design Guide* [Ref 36] for your device.
   - Board design guidelines in the *Transceiver User Guide* [Ref 41] for your device.
2. Review memory IP and PCIe® design guidelines in the IP product guides.
3. Use the Vivado® tools to validate your I/O planning:
   - Run simultaneous switching noise (SSN) analysis.
   - Run built-in DRCs.
   - Export I/O buffer information specification (IBIS) models.
4. Run signal integrity analysis as follows:
   - For gigabit transceivers (GTs), run Spice or IBIS-AMI simulations using channel parameters.
   - For lower performance interfaces, run IBIS simulation to check for issues with overshoot or undershoot.
5. Use the Xilinx Power Estimator (XPE) with Process set to **Maximum** to generate an early estimate of the power consumption for the design.

**TIP:** *You can use the results of this early estimation with the* `set_operating_conditions -design_power_budget <Power in Watts>` *Tcl command to ensure the power budget is checked during design implementation.*

6. Complete and adhere to the schematic checklist for your device.

# Clock Resource Planning and Assignment

Xilinx recommends that you select clocking resources as one of the first steps of your design, well before pinout selection. Your clocking selections can dictate a particular pinout and can also direct logic placement for that logic, especially for SSI technology devices. Proper clocking selections can yield superior results. Consider the following:

- Constraint creation, particularly in large devices with high utilization in conjunction with clock planning.
- Manual placement of clocking resources if needed for design closure. Clocking Guidelines in Chapter 3 provides more details on clocking resources, if you need to do manual placement.
- Device-specific functionality that might require up-front planning to avoid issues and take advantage of device features. For information on 7 series features, see this link and

this link in the *7 Series FPGAs Clocking Resources User Guide* (UG472) [Ref 40]. For information on UltraScale™ device features, see this link in the *UltraScale Architecture Clocking Resources User Guide* (UG572) [Ref 40].

# I/O Planning Design Flows

The Vivado IDE allows you to interactively explore, visualize, assign, and validate the I/O ports and clock logic in your design. The environment ensures correct-by-construction I/O assignment. It also provides visualization of the external package pins in correlation with the internal die pads.

You can visualize the data flow through the device and properly plan I/Os from both an external and internal perspective. After the I/Os are assigned and configured through the Vivado IDE, constraints are then automatically created for the implementation tools.

For more information on Vivado Design Suite I/O and clock planning capabilities, see the following resources:

- *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) [Ref 5]
- Vivado Design Suite QuickTake Video: I/O Planning Overview

## Types of Vivado Design Suite Projects for I/O Planning

You can perform I/O planning with either of the following types of projects:

- I/O planning project

  An I/O planning project is an easy entry point that allows you to specify select I/O constraints and generate a top-level RTL file from the defined pins.

- RTL project

  An RTL project allows synthesis and implementation, which enables more comprehensive design rule checks (DRCs). An RTL project also allows generation of IP cores, which is important for memory interface pinout planning and any cores using GTs.

**TIP:** *You can also start by using an I/O planning project and migrate to an RTL project later.*

You can run more comprehensive DRCs on a post-synthesis netlist. The same is true after implementation and bitstream generation. Therefore, Xilinx recommends using a skeleton design that includes clocking components and some basic logic to exercise the DRCs. This builds confidence that the pin definition for the board will not have issues later.

The recommended sign-off process is to run the RTL project through to bitstream generation to exercise all the DRCs. However, not all design cycles allow enough time for this process. Often the I/O configuration must be defined before you have synthesizable RTL. Although the Vivado tools enable pre-RTL I/O planning, the level of DRCs performed are fairly basic. Alternatively, you can use a dummy top-level design with I/O standards and pin assignments to help perform DRCs related to banking rules.

### *Pre-RTL I/O Planning*

If your design cycle forces you to define the I/O configuration before you have a synthesized netlist, take great care to ensure adherence to all relevant rules. The Vivado tools include a Pin Planning Project environment that allows you to import I/O definitions using a CSV or XDC format file. You can also create a dummy RTL with just the port directions defined. Availability of port direction makes simultaneous-switching-noise (SSN) analysis more accurate as input and output signals have different contributions to SSN.

I/O ports can also be created and configured interactively. Basic I/O bank DRC rules are provided.

See the *PCB Design Guide* [Ref 36] for your device to ensure proper I/O configuration. For more information, see this link in the *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) [Ref 5].

### *Netlist-Based I/O Planning*

The recommended time in the design cycle to assign I/Os and clock logic constraints is after the design has been synthesized. The clock logic paths are established in the netlist for constraint assignment purposes. The I/O and clock logic DRCs are also more comprehensive.

See the *PCB Design Guide* [Ref 36] for your device to ensure proper I/O configuration. For more information, see this link in the *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) [Ref 5].

## Defining Alternate Devices

It is often difficult to predict the final device size for any given design during initial planning. Logic can be added or removed during the course of the design cycle, which can result in the need to change the device size.

The Vivado tools enable you to define alternate devices to ensure that the I/O pin configuration defined is compatible across all selected devices, as long as the package is the same.

**IMPORTANT:** *The device must be in the same package.*

To migrate your design with reduced risk, carefully plan the following at the beginning of the design process: device selection, pinout selection, and design criteria. Take the following into account when migrating to a larger or smaller device in the same package: pinout, clocking, and resource management. For more information, see this link in the *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) [Ref 5].

## Pin Assignment

Good pinout selection leads to good design logic placement, shorter routes, reduced power consumption, and improved performance. Good pinout selection is especially important for large FPGA devices, because a pinout that is spread out can cause related signals to span longer distances. For more information, see this link in the *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) [Ref 5].

### *Using Xilinx Tools in Pinout Selection*

Xilinx tools assist in interactive design planning and pin selection. These tools are only as effective as the information you provide them. Tools such as the Vivado design analysis tool can assist pinout efforts. These tools can graphically display the I/O placement, show relationships among clocks and I/O components, and provide DRCs to analyze pin selection.

If a design version is available, a quick top-level floorplan can be created to analyze the data flow through the device. For more information, see the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21].

### *Required Information*

For the tools to work effectively, you must provide as much information about the I/O characteristics and topologies as possible. You must specify the electrical characteristics, including the I/O standard, drive, slew, and direction of the I/O.

You must also take into account all other relevant information, including clock topology and timing constraints. Clocking choices in particular can have a significant influence in pinout selection and vice versa, as discussed in Clocking Guidelines in Chapter 3.

For IP that have I/O requirements, such as transceivers, PCIe, and memory interfaces, you must configure the IP prior to completing I/O pin assignment, as described in Pinout Selection. For more information on specifying the electrical characteristics for an I/O, see this link in the *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) [Ref 5].

## Pinout Selection

Xilinx recommends careful pinout selection for some specific signals as discussed below.

### *Interface Data, Address, and Control Pins*

Group the same interface data, address, and control pins into the same bank. If you cannot group these components into the same bank, group them into adjacent banks. For stacked silicon interconnect (SSI) technology devices, adjacent banks must also be located within the same super logic region (SLR).

### *Interface Control Signals*

Place the following interface control signals in the middle of the data buses they control (clocking, enables, resets, and strobes).

### *Very High Fanout, Design-Wide Control Signals*

Place very high fanout, design-wide control signals towards the center of the device.

For SSI technology devices, place the signals in the SLR located in the middle of the SLR components they drive.

### *Configuration Pins*

To design an efficient system, you must choose the FPGA configuration mode that best matches the system requirements. Factors to consider include:

- Using dedicated vs. dual purpose configuration pins.

  Each configuration mode dedicates certain FPGA pins and can temporarily use other multi-function pins during configuration only. These multi-function pins are then released for general use when configuration is completed.

- Using configuration mode to place voltage restrictions on some FPGA I/O banks.
- Choosing suitable terminations for different configuration pins.
- Using the recommended values of pull-up or pull-down resistors for configuration pins.

**RECOMMENDED:** *Even though configuration clocks are slow speed, perform signal integrity analysis on the board to ensure clean signals.*

There are several configuration options. Although the options are flexible, there is often an optimal solution for each system. Consider the following when choosing the best configuration option:

- Setup
- Speed
- Cost
- Complexity

See Configuration. For more information on FPGA configuration options, see *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 24].

### *Memory Interfaces*

Additional I/O pin planning steps are required when using Xilinx Memory IP. After the IP is customized, you then assign the top-level IP ports to physical package pins in either the elaborated or synthesized design in the Vivado IDE. All of the ports associated with each Memory IP are group together into an I/O Port Interface for easier identification and assignment. A Memory Bank/Byte Planner is provided to assist you with assigning Memory I/O pin groups to byte lanes on the physical device pins. For more information, see this link in the *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) [Ref 5].

Take care when assigning memory interfaces and try to limit congestion as much as possible, especially with devices that have a center I/O column. Bunching memory interfaces together can create routing bottlenecks across the device. The *Xilinx Zynq-7000 SoC and 7 Series Devices Memory Interface User Guide* (UG586) [Ref 48] and the *LogiCORE IP UltraScale Architecture-Based FPGAs Memory Interface Solutions Product Guide* (PG150) [Ref 49] contain design and pinout guidelines. Be sure that you follow the trace length match recommendations in these guides, verify that the correct termination is used, and validate the pinout in by running the DRCs after memory IP I/O assignment.

### *Gigabit Transceivers (GTs)*

Gigabit transceivers (GTs) have specific pinout requirements, and you must consider the following:

- Sharing of reference clocks
- Sharing of PLLs within a quad
- Placement of hard blocks, such as PCIe, and their proximity to transceivers
- In SSI technology devices, crossing of SLR boundaries

Xilinx recommends that you use the GT wizard to generate the core. Alternatively, you can use the Xilinx IP core for the protocol. For pinout recommendations, see the related product guide.

For clock resource balancing, the Vivado placer attempts to constrain loads clocked by GT output clocks (TXOUTCLK or RXOUTCLK) next to the GTs sourcing the clocks. For stacked silicon interconnect (SSI) technology devices, if the GTs are located in the clock regions adjacent to another SLR, the routing resources required for signals entering or exiting SLLs have to compete with the routing resources required by the GT output clock loads. Therefore, GTs located in clock regions next to SLR crossings might reduce the available routing connections to and from the SLL crossings available in those clock regions.

### *High Speed I/O*

HP (high-performance) and HR (high-range) banks have difference in the speed with which they can transmit and receive signals. Depending upon the I/O speed you need, choose between HP or HR banks.

### *Internal VREF and DCI Cascade Constraints*

Based on the settings for DCI Cascade and Internal VREF, you can free up pins to be used for regular I/Os. These settings also ensure that related DRC checks are run to validate the legality of the constraints. For more information, see the *SelectIO Resources User Guide* [Ref 39] for your device.

## Interface Bandwidth Validation

Create small connectivity designs to validate each interface on the FPGA. These small designs exercise only the specific hardware interface, which enables the following:

- Full DRC checks on pinout, clocking, and timing
- Hardware test design when the board is returned
- Rapid implementation through the Vivado tools, providing the fastest way to debug the interface

There are multiple options to assist in generating test data for these interfaces. For some of the interface IP cores, the Vivado tools can provide the test designs:

- IBERT for SerDes
- Example design within IP cores

**TIP:** *If a test design does not exist, consider using AXI traffic generators.*

You might need to create a separate design for a system-level test in a production environment. Usually, this is a single design that includes tested interfaces and optionally includes processors. You can construct this design using the small connectivity designs to take advantage of design reuse. Although this design is *not* required early in the flow, it can

enable better DRC checks and early software development, and you can quickly create it using the Vivado IP integrator.

# Designing with SSI Devices

## SSI Pinout Considerations

When planning pinouts for components that are located in a particular SLR, place the pins into the same SLR. For example, when using the device DNA information as a part of an external interface, place the pins for that interface in the master SLR in which the DNA_PORT exists. Additional considerations include the following:

- Group all pins of a particular interface into the same SLR.
- For signals driving components in multiple SLRs, place those signals in the middle SLR.
- Balance CCIO or CMT components across SLRs.
- Reduce SLR crossings.

## Super Logic Region (SLR)

A super logic region (SLR) is a single device die slice contained in an SSI technology device. Each SLR contains a subset of device resources, such as CLBs, block RAMs, DSP tiles, and GTs, with a similar structure to non-SSI devices.

Multiple SLR components are stacked vertically and connected through an interposer to create an SSI technology device. The bottom SLR is SLR0, and subsequent SLR components are named incrementally as they ascend vertically. For example, the XC7V2000T device includes four SLR components. The bottom SLR is SLR0, the SLR directly above SLR0 is SLR1, the SLR directly above SLR1 is SLR2, and the top SLR is SLR3.

***Note:*** The Xilinx tools clearly identify SLR components in the graphical user interface (GUI) and in reports.

## SLR Nomenclature

Understanding SLR nomenclature for your target device is important in:

- Pin selection
- Floorplanning
- Analyzing timing and other reports
- Identifying where logic exists and where that logic is sourced or destined

You can use the Vivado Tcl command `get_slrs` to get specific information about SLRs for a particular device. For example, use the following commands:

  - `llength [get_slrs]` to obtain the number of SLRs in the device
  - `get_slrs -of_objects [get_cells my_cell]` to get the SLR in which `my_cell` is placed

## Master Super Logic Region

Every SSI technology device has a single master SLR. The master SLR contains the primary configuration logic that initiates configuration of the device and all other SLR components. The master SLR contains the circuitry that is used for configuration, DNA_PORT, and EFUSE_USER. When using these components, the place and route tools can assign associated pins and logic to the appropriate SLR. In general, no additional intervention is required.

**TIP:** *To query which SLR is the master SLR in the Vivado Design Suite, you can enter the* `get_slrs -filter IS_MASTER` *Tcl command.*

## Silicon Interposer

The silicon interposer is a passive layer in the SSI technology device, which routes the following between SLR components:

- Configuration
- Global clocking
- General interconnect

## Super Long Line (SLL) Routes

Super Long Line (SLL) routes connect signals from one SLR to another inside the device.

**TIP:** *To determine the number of available SLLs between SLRs, use SLR properties. For example:*

```
get_property NUM_TOP_SLLS [get_slrs SLR0]
get_property NUM_BOT_SLLS [get_slrs SLR1]
```

## Propagation Limitations

**TIP:** *For high-speed propagation across SLRs, be sure to register signals that cross SLR boundaries.*

SLL signals are the only data connections between SLR components.

The following do not propagate across SLR components:

- Carry chains
- DSP cascades
- Block RAM address cascades
- Other dedicated connections, such as DCI cascades and block RAM cascades

The tools normally take this limit on propagation into account. To ensure that designs route properly and meet your design goals, you must also take this limit into account when you:

- Build a very long DSP cascade and manually place such logic near SLR boundaries
- Specify a pinout for the design

## SLR Utilization Considerations

The Vivado implementation tools use a special algorithm to partition logic into multiple SLRs. For challenging designs, you can improve timing closure for designs that target SSI technology devices using the following guidelines.

To improve timing closure and compile times, you can use Pblocks to assign logic to each SLR and validate that individual SLRs do not have excessive utilization across all fabric resource types. For example, a design with block RAM utilization of 70% might cause issues with timing closure if the block RAM resources are not balanced across SLRs and one SLR is using over 85% block RAM.

The following example utilization report for a vu160 shows that the overall block RAM utilization is 56% while the block RAM utilization in SLR0 is 89% (897 out of 1008 available). Timing closure might be more difficult to achieve for the design in SLR0 than with a balanced block RAM utilization across SLRs.

```
3. BLOCKRAM
-----------
+-------------------+------+-------+-----------+-------+
|     Site Type     | Used | Fixed | Available | Util% |
+-------------------+------+-------+-----------+-------+
| Block RAM Tile    | 1843 |     0 |      3276 | 56.26 |
|   RAMB36/FIFO*    | 1820 |     1 |      3276 | 55.56 |
|     FIFO36E2 only |   78 |       |           |       |
|     RAMB36E2 only | 1742 |       |           |       |
|   RAMB18          |   46 |     0 |      6552 |  0.70 |
|     RAMB18E2 only |   46 |       |           |       |
+-------------------+------+-------+-----------+-------+
```

```
14. SLR CLB Logic and Dedicated Block Utilization
-------------------------------------------------
+-----------+--------+---------+------------+-------------+---------------+-----------+-------+------+
| SLR Index |  CLBs  | (%)CLBs | Total LUTs | Memory LUTs | (%)Total LUTs | Registers | BRAMs | DSPs |
+-----------+--------+---------+------------+-------------+---------------+-----------+-------+------+
| SLR2      |  40109 |   89.61 |     167520 |         156 |         46.78 |    327600 |   512 |    0 |
| SLR1      |  42649 |   95.28 |     205484 |        2297 |         57.38 |    355918 |   434 |    0 |
| SLR0      |  35379 |   97.62 |     163188 |          24 |         56.29 |    313392 |   897 |    0 |
+-----------+--------+---------+------------+-------------+---------------+-----------+-------+------+
| Total     | 118137 |         |     536192 |        2477 |               |    996910 |  1843 |    0 |
+-----------+--------+---------+------------+-------------+---------------+-----------+-------+------+
```

Xilinx recommends assigning block RAM and DSP groups to SLR Pblocks to minimize SLR crossings of shared signals. For example, an address bus that fans out to a group of block RAMs that are spread out over multiple SLRs can make timing closure more difficult to achieve, because the SLR crossing incurs additional delay for the timing critical signals.

Device resource location or user I/O selection anchors IP to SLRs, for example, GT, ILKN, PCIe, and CMAC dedicated block or memory interface controllers. Xilinx recommends the following:

- Pay special attention to dedicated block location and pinout selection to avoid data flow crossing SLR boundaries multiple times.
- Keep tightly interconnected modules and IPs within the same SLR. If that is not possible, you can add pipeline registers to allow the placer more flexibility to find a good solution despite the SLR crossing between logic groups.
- Keep critical logic within the same SLR. By ensuring that main modules are properly pipelined at their interfaces, the placer is more likely to find SLR partitions with flip-flop to flip-flop SLR crossings.

In the following figure, a memory interface that is constrained to SLR0 needs to drive user logic in SLR1. An AXI4-Lite slave interface connects to the memory IP backend, and the well-defined boundary between the memory IP and the AXI4-Lite slave interface provides a good transition from SLR0 to SLR1.

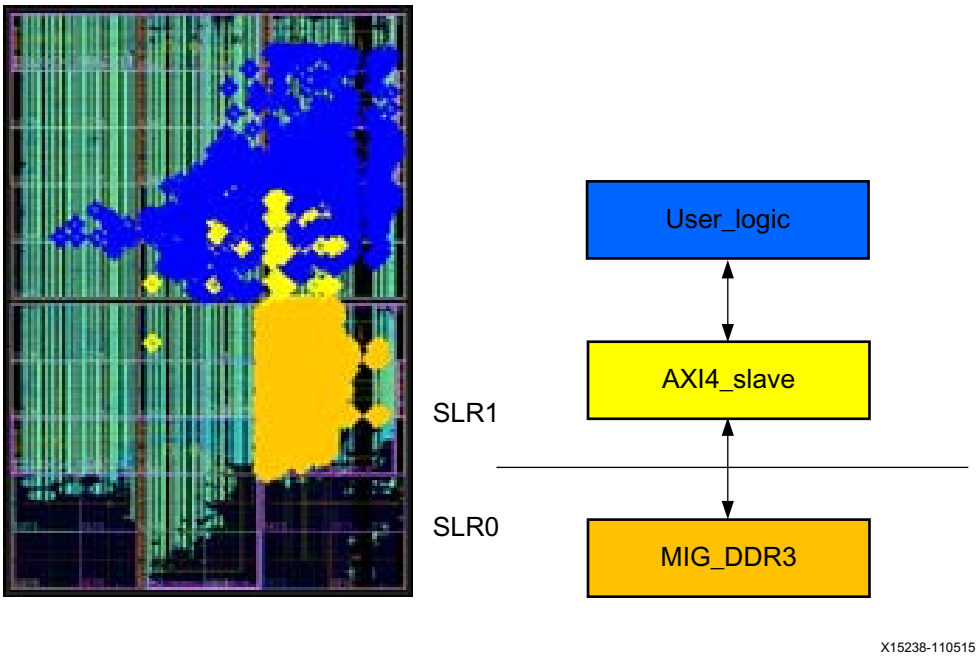



*Figure 2-2:* **Memory Interface in SLR0 Driving User Logic in SLR1**

## SLR Crossing for Wide Buses

When data flow requirements require that wide buses cross SLRs, use pipelining strategies to improve timing closure and alleviate routing congestion of long resources. For wide buses operating above 250 MHz, Xilinx recommends using at least three pipeline stages to cross an SLR: one at the top, one at the bottom, and one in the middle of the SLR. Additional pipeline stages might be required for very high performance buses or when traversing horizontal as well as vertical distances.

The following figure illustrates a worst case crossing for a vu190-2 device. This example starts at an Interlaken dedicated block in the bottom left of SLR0 to a packet monitor block assigned to the top right of SLR2. Without pipeline registers for the data bus to and from the packet monitor, the design misses the 300 MHz timing requirement by a wide margin.

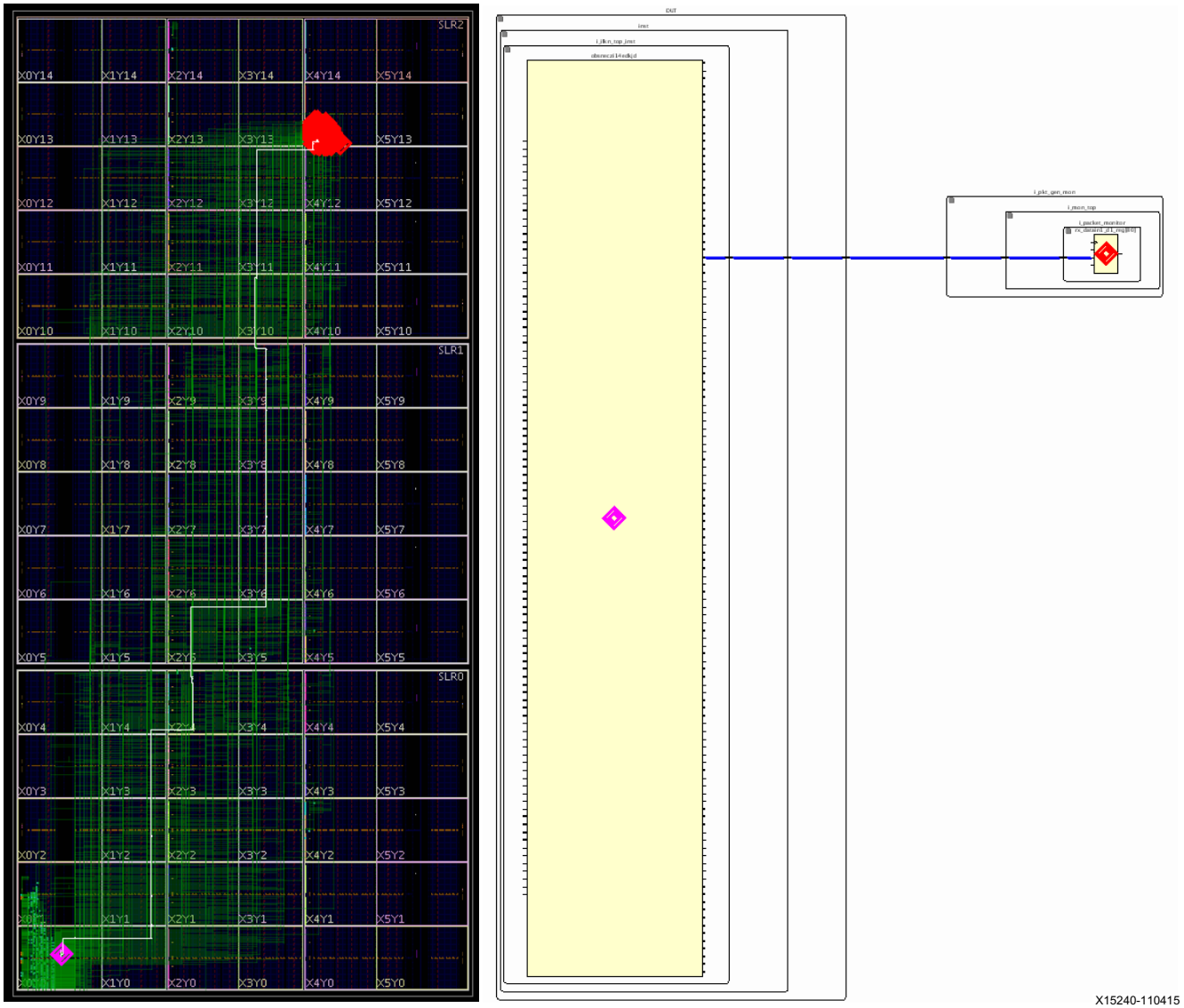



*Figure 2-3:* **Data Path Crossing SLR without Pipeline Flip-Flop**

However, adding seven pipeline stages to aid in the traversal from SLR0 to SLR2 allows the design to meet timing. It also reduces the use of vertical and horizontal long routing resources, as shown in the following figure.

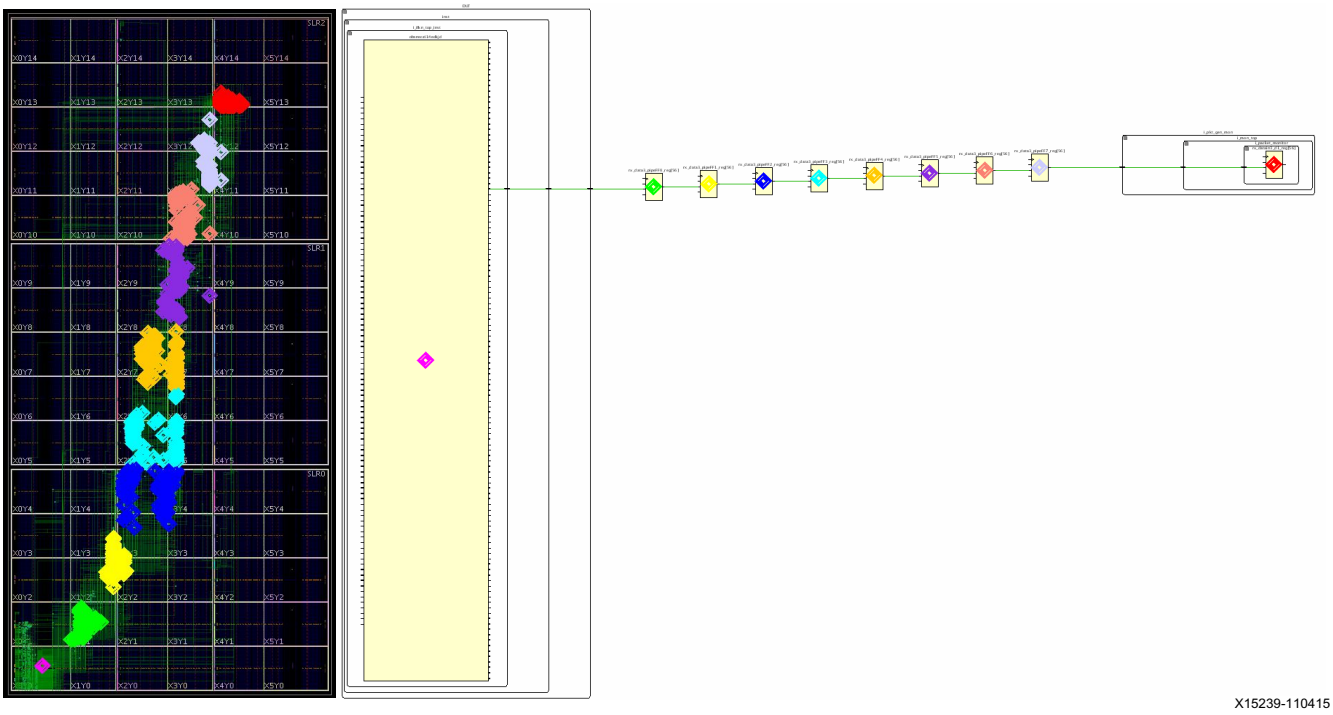



*Figure 2-4:* **Data Path Crossing SLR with Pipeline Flip-Flop Added**

# Device Power Aspects and System Dependencies

When planning the PCB, you must take power into consideration:

- The device and the user design create system power supply and heat dissipation requirements.
- Power supplies must be able to meet maximum power requirements and the device must remain within the recommended voltage and temperature operating conditions during operation. Power estimation and thermal modeling might be required to ensure that the device stays within these limits.
- Plan for the consolidation of power rails and their impact on power domain switching.

For these reasons, you must understand the power and cooling requirements of the device. These must be designed on the board.

## Power Supply Paths on Devices

Multiple power supplies are required to power a device. Some of this power must be provided in a specific sequence. Consider the use of power monitoring or sequencing circuitry to provide the correct power-on sequence to the device and GTs as well as any additional active components on the board. More complex environments might benefit from the use of a microcontroller or system and power management bus such as SMBUS or PMBUS to control the power and reset process. Specific details regarding on/off sequencing can be found in the device data sheet. For more information on supply consolidation and topologies, see the *PCB Design Guide* [Ref 36] for your device.

The separate sources provide the required power for the different device resources. This allows different resources to work at different voltage levels for increased performance or signal strength, while preserving a high immunity to noise and parasitic effects.

## Power Modes

A device goes through several power phases from power up to power down with varying power requirements.

### *Power-On*

Power-on power is the transient spike current that occurs when power is first applied to the device. This current varies for each voltage supply and depends on the device construction, the ability of the power supply source to ramp up to the nominal voltage, and the device operating conditions, such as temperature and sequencing between the different supplies.

Spike currents are not a concern in modern device architectures when the proper power-on sequencing guidelines are followed.

### *Startup Power*

Startup power is the power required during the initial bring-up and configuration of the device. This power generally occurs over a very short period of time and thus is not a concern for thermal dissipation. However, current requirements must still be met. In most cases, the active current of an operating design will be higher and thus no changes are necessary. However, for lower-power designs where active current can be low, a higher current requirement during this time may be necessary. Xilinx Power Estimator (XPE) can be used to understand this requirement. When Process is set to **Maximum**, the current requirement for each voltage rail will be specified to either the operating current or the startup current, whichever is higher. XPE will display the current value in blue if the startup current is the higher value.

### *Standby Power*

Standby power (also called *design static power*) is the power supplied when the device is configured with your design and no activity is applied externally or generated internally.

Standby power represents the minimum continuous power that the supplies must provide while the design operates.

### *Active Power*

Active power (also called *design dynamic power*) is the power required while the device is running your application. Active power includes standby power (all static power), plus power generated from the design activity (design dynamic power). Active power is instantaneous and varies at each clock cycle depending on the input data pattern and the design internal activity.

## Environmental Factors Impacting Power

In addition to the design itself, environmental factors affect power. These factors influence the voltage and the junction temperature of the device, which impacts the power dissipation. For details, see this link in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 22].

## Power Rail Consolidation Impacting Power

To take advantage of the power management switching of power domains, your design must keep some discrete power rails. This allows individual rails to be powered off with the power domain switching logic. For more information, see this link in the *UltraScale Architecture PCB Design User Guide* (UG583).

## Power Models Accuracy

The accuracy of the characterization data embedded in the tools evolves over time to reflect the device availability or manufacturing process maturity. For details, see this link in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 22].

### Device Power and the Overall System Design Process

From project conception to completion, various aspects of the design process affect power. For details, see this link in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 22].

**TIP:** *During the design process, you can compare the total power of the design to the power budget using the* `set_operating_conditions -design_power_budget <Power in Watts>` *Tcl command. If the power budget is exceeded, early intervention is the easiest way to correct design power.*

### Worst Case Power Analysis Using Xilinx Power Estimator (XPE)

Xilinx recommends designing the board for worst-case power. For details, see this link in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 22].

## Configuration

Configuration is the process of loading application-specific data into the internal memory of the FPGA device.

Because Xilinx FPGA configuration data is stored in CMOS configuration latches (CCLs), the configuration data is volatile and must be reloaded each time the FPGA device is powered up.

Xilinx FPGA devices can load themselves through configuration pins from an external nonvolatile memory device. They can also be configured by an external smart source, such as a:

- Microprocessor
- DSP processor
- Microcontroller
- Personal Computer (PC)
- Board tester

Board Planning should consider configuration aspects up front, which makes it easier to configure as well as debug.

Each device family has a *Configuration User Guide* [Ref 38] that is the primary resource for detailed information about each of the supported configuration modes and their trade-offs on pin count, performance, and cost.

## Board Design Tips

When designing a board, it is important to consider which interfaces and pins will assist with debug capability beyond configuration. For example, Xilinx recommends that you ensure the JTAG interface is accessible even when the interface is not the primary configuration mode. The JTAG interface allows you to check the device ID and device DNA information, and you can use the interface to enable indirect flash programming solutions during prototyping.

In addition, signals such as the INIT_B and DONE are critical for FPGA configuration debug. The INIT_B signal has multiple functions. It indicates completion of initialization at power-up and can indicate when a CRC error is encountered. Xilinx recommends that you connect the INIT_B and DONE signals to LEDs using LED drivers and pull-ups. For recommended pull-up values, see the *Configuration User Guide* [Ref 38] for your device.

The *Schematic Checklists* [Ref 52] include these recommendations along with other key suggestions. Use these checklists to identify and check recommended board-level pin connections.

# Design Creation

## Overview of Design Creation

After planning your device I/O, planning how to lay out your PCB, and deciding on your use model for the Vivado® Design Suite, you can begin creating your design. Design creation includes:

- Planning the hierarchy of your design
- Identifying the IP cores to use and customize in your design
- Creating the custom RTL for interconnect logic and functionality for which a suitable IP is not available
- Creating timing, power, and physical constraints
- Specifying additional constraints, attributes, and other elements used during synthesis and implementation

When creating your design, the main points to consider include:

- Achieving the desired functionality
- Operating at the desired frequency
- Operating with the desired degree of reliability
- Fitting within the silicon resource and power budget

Decisions made at this stage affect the end product. A wrong decision at this point can result in problems at a later stage, causing issues throughout the entire design cycle. Spending time early in the process to carefully plan your design helps to ensure that you meet your design goals and minimize debug time in lab.

# Defining a Good Design Hierarchy

The first step in design creation is to decide how to partition the design logically. The main factor when considering hierarchy is to partition a part of the design that contains a specific function. This allows a specific designer to design a piece of IP in isolation as well as isolating a piece of code for reuse.

However, defining a hierarchy based on functionality only does not take into account how to optimize for timing closure, runtime, and debugging. The following additional considerations made during hierarchy planning also help in timing closure.

## Add I/O Components Near the Top Level

Where possible, add I/O components near the top level for design readability. When you infer a component, you provide a description of the function you want to accomplish. The synthesis tool then interprets the HDL code to determine which hardware components to use to perform the function. Components that can be inferred are simple single-ended I/O (IBUF, OBUF, OBUFT and IOBUF) and single data rate registers in the I/O.

I/O components that need to be instantiated, such as differential I/O (IBUFDS, OBUFDS) and double data-rate registers (IDDR, ODDR, ISERDES, OSERDES), should also be instantiated near the top level. When you instantiate a component, you add an instance of that component to your HDL file. Instantiation gives you full control over how the component is used. Therefore, you know exactly how the logic will be used.

## Insert Clocking Elements Near the Top Level

Inserting the clocking elements towards the top level allows for easier clock sharing between modules. This sharing may result in fewer clocking resources needed, which helps in resource utilization, improved performance, and power.

Aside from the module the clocks are created in, clock paths should only drive down into modules. Any paths that go through (down from top and then back to top) can create a delta cycle problem in VHDL simulation that is difficult and time consuming to debug.

## Register Data Paths at Logical Boundaries

Register the outputs of hierarchical boundaries to contain critical paths within a single module or boundary. Consider registering the inputs also at the hierarchical boundaries. It is always easier to analyze and repair timing paths which lie within a module, rather than a path spanning multiple modules. Any paths that are not registered at hierarchy boundaries should be synthesized with hierarchy rebuilt or flat to allow cross hierarchy optimization. Registering the datapaths at logical boundaries helps to retain traceability (for debug)

through the design process because cross hierarchical optimizations are kept to a minimum and logic does not move across modules.

## Address Floorplanning Considerations

A floorplan ensures that cells belonging to a specific portion in the design netlist are placed at particular locations on the device. You can use manual floorplanning to accomplish the following:

- Partition logic to a particular SLR when using SSI technology devices.
- Close timing on a design when timing is not met using standard flows.

If the cells are not contained within a level of hierarchy, all objects must be included individually in the floorplan constraint. If synthesis changes the names of these objects, you must update the constraints. A good floorplan is contained at the hierarchy level, because this requires only a one line constraint.

Floorplanning is not always required. Floorplan only when necessary.

For more information on floorplanning, see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21].

**RECOMMENDED:** *Although the Vivado tools allow cross hierarchy floorplans, these require more maintenance. Avoid cross hierarchy floorplans where possible.*

## Optimize Hierarchy for Functional and Timing Debug

As discussed earlier in this section, keeping the critical path within the same hierarchical boundary is helpful in debugging and meeting timing requirements. Similarly, for functional debug (and modification) purposes, signals that are related should be kept in the same hierarchy. This allows the related signals to be probed and modified with relative ease, as signal names optimized by synthesis are easier to trace when contained in a single level of hierarchy.

## Apply Attributes at the Module Level

Applying attributes at the module level can keep code tidier and more scalable. Instead of having to apply an attribute at the signal level, you can apply the attribute at the module level and have the attribute propagated to all signals declared in that region. Applying attributes at the module level also allows you to override global synthesis options. For this reason, it is sometimes advantageous to add a level of hierarchy in order to apply module level constraints in the RTL.

**CAUTION!** *Some attributes (e.g., DONT_TOUCH) do not propagate from a module to all the signals inside the module.*

## Optimize Hierarchy for Advanced Design Techniques

Advanced design techniques such as bottom-up synthesis, partial reconfiguration, and out-of-context design require planning at the hierarchical level. The design must choose the appropriate level of hierarchy for the technique being used. These techniques are not covered in this version of the document. For more information, see this link in the *Vivado Design Suite User Guide: Hierarchical Design* (UG905) [Ref 20].

## Example of Upfront Hierarchical Planning for High Speed DSP Designs

The following example is not applicable to all designs, but demonstrates what can be done with hierarchy. DSP designs generally allow latency to be added to the design. This allows registers to be added to them to be optimized for performance. In addition, registers can be used to allow for placement flexibility. This is important because at high speed, you cannot traverse the die in one clock cycle. Adding registers can allow hard-to-reach areas to be used. The following figure shows how effective hierarchy planning results in faster timing closure.

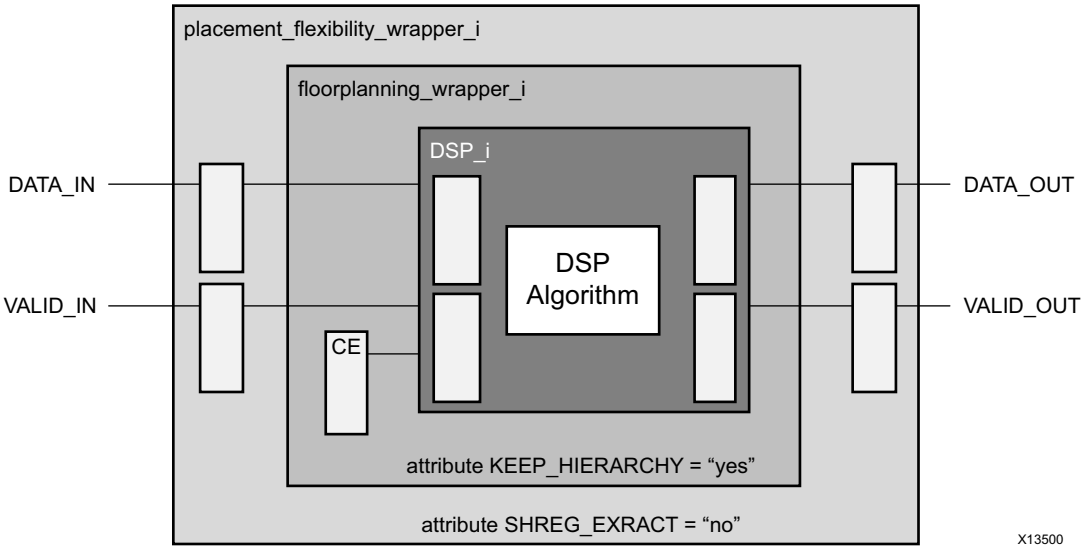



*Figure 3-1:* **Effective Hierarchy Planning Example**

There are three levels of hierarchy in this part of the design:

- `DSP_i`

  In the DSP_i algorithm block, both the inputs and outputs are registered. Because registers are plentiful in an FPGA device, it is preferable to use this method to improve the timing budget.

- `floorplanning_wrapper_i`

  In `floorplanning_wrapper_i`, there is a CE signal. CE signals are typically heavily-loaded and can present a timing challenge. They should be included in a floorplan. By creating a floorplanning wrapper, this module can be manually floorplanned later if needed.

  In addition, KEEP_HIERARCHY has been added at the module level to ensure that hierarchy is preserved for floorplanning regardless of any other global synthesis options.

- `placement_flexibility_wrapper_i`

  In `placement_flexibility_wrapper_i`, the DATA_IN, VALID_IN, DATA_OUT and VALID_OUT signals are registered. Because these signals are not intended to be part of the floorplan, they are outside `floorplanning_wrapper_i`. If they were in the floorplan, they would not be able to fulfill the requirement for placement flexibility.

  In addition, more registers can be added later as long as both DATA_IN + VALID_IN or DATA_OUT and VALID_OUT are treated as pairs. If more registers are added, the synthesis tool may infer SRLs which will force all registers into one component and not help placement flexibility. To prevent this, SHREG_EXTRACT has been added at the module level and set to NO.

# RTL Coding Guidelines

You can create custom RTL to implement glue logic functions as well as functions without suitable IP. For optimal results, follow the coding guidelines in this section. For additional guidelines, see this link in the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 16].

## Using Vivado Design Suite HDL Templates

Use the Vivado Design Suite Language Templates when creating RTL or instantiating Xilinx® primitives. The Language Templates include recommended coding constructs for proper inference to the Xilinx device architecture. Using the Language Templates can ease the design process and lead to improved results. To open the Language Templates from the

Vivado IDE, select the **Language Templates** option in the **Flow Navigator**, and select the desired template.

## Control Signals and Control Sets

A control set is the grouping of control signals (set/reset, clock enable and clock) that drives any given SRL, LUTRAM, or register. For any unique combination of control signals, a unique control set is formed. The reason this is an important concept is registers within a 7 series slice all share common control signals and thus only registers with a common control set may be packed into the same slice. For example, if a register with a given control set has just one register as a load, the other seven registers in the slice it occupies will be unusable.

Designs with several unique control sets may have many wasted resources as well as fewer options for placement, resulting in higher power and lower performance. Designs with fewer control sets have more options and flexibility in terms of placement, generally resulting in improved results.

In UltraScale™ devices, there is more flexibility in control set mapping within a CLB. Resets that are undriven do not form part of the control set as the tie off is generated locally within the slice. However, it is good practice to limit unique control sets to give maximum flexibility in placement of a group of logic.

### *Resets*

Resets are one of the more common and important control signals to take into account and limit in your design. Resets can significantly impact your design's performance, area, and power.

Inferred synchronous code may result in resources such as:

- LUTs
- Registers
- Shift Register LUTs (SRLs)
- Block or LUT Memory
- DSP48 registers

The choice and use of resets can affect the selection of these components, resulting in less optimal resources for a given design. A misplaced reset on an array can mean the difference between inferring one block RAM, or inferring several thousand registers.

Asynchronous resets described at the input or output of a multiplier might result in registers placed in the slice(s) rather than the DSP block. In these and other situations, the amount of resources is impacted. Overall power and performance can also be significantly impacted. In most cases, this impacts performance. It also has a negative impact on device utilization and power consumption.

### When and Where to Use a Reset

Xilinx devices have a dedicated global set/reset signal (GSR). This signal sets the initial value of all sequential cells in hardware at the end of device configuration.

If an initial state is not specified, sequential primitives are assigned a default value. In most cases, the default value is zero. Exceptions are the FDSE and FDPE primitives that default to a logic one. Every register will be at a known state at the end of configuration. Therefore, it is not necessary to code a global reset for the sole purpose of initializing a device on power up.

Xilinx highly recommends that you take special care in deciding when the design requires a reset, and when it does not. In many situations, resets might be required on the control path logic for proper operation. However, resets are generally less necessary on the data path logic. Limiting the use of resets:

- Limits the overall fanout of the reset net.
- Reduces the amount of interconnect necessary to route the reset.
- Simplifies the timing of the reset paths.
- Results in many cases in overall improvement in performance, area, and power.

**RECOMMENDED:** *Evaluate each synchronous block, and attempt to determine whether a reset is required for proper operation. Do not code the reset by default without ascertaining its real need.*

Functional simulation should easily identify whether a reset is needed or not.

For logic in which no reset is coded, there is much greater flexibility in selecting the FPGA resources to map the logic.

The synthesis tool can then pick the best resource for that code in order to arrive at a potentially superior result by considering, for example:

- Requested functionality
- Performance requirements
- Available device resources
- Power

### Synchronous Reset vs. Asynchronous Reset

If a reset is needed, Xilinx recommends code synchronous resets. Synchronous resets have many advantages over asynchronous resets.

- Synchronous resets can directly map to more resource elements in the FPGA device architecture.

- Asynchronous resets also impact the performance of the general logic structures. As all Xilinx FPGA general-purpose registers can program the set/reset as either asynchronous or synchronous, it can be perceived that there is no penalty in using asynchronous resets. That assumption is often wrong. If a global asynchronous reset is used, it does not increase the control sets. However, the need to route this reset signal to all register elements increases timing complexity.
- If using asynchronous reset, remember to synchronize the deassertion of the asynchronous reset.
- Synchronous resets give more flexibility for control set remapping when higher density or fine tuned placement is needed. A synchronous reset may be remapped to the data path of the register if an incompatible reset is found in the more optimally placed Slice. This can reduce wire length and increase density where needed to allow proper fitting and improved performance.
- Asynchronous resets might require multi-cycle assertion to ensure a circuit is properly reset and stable. When properly timed, synchronous resets do not have this requirement.
- Use synchronous resets if asynchronous resets have a greater probability of upsetting memory contents to block RAMs, LUTRAMs, and SRLs during reset assertion.
- Some resources such as the DSP48 and block RAM have only synchronous resets for the register elements within the block. When asynchronous resets are used on register elements associated with these elements, those registers may not be inferred directly into those blocks without impacting functionality.

**Reset Coding Example One: Multiplier with Asynchronous Reset**

The following example illustrates the importance of using registers with synchronous resets for the logic targeting the dedicated DSP resources. Figure 3-2 shows a 16x16 bits DSP48-based multiplier using pipeline registers with asynchronous reset. Synthesis must use regular fabric registers for the input stages, as well as an external register and 32 LUT2s (red markers) to emulate the asynchronous reset on the DSP output (DSP48 P registers are enabled but not connected to reset). This costs an extra 65 registers and 32 LUTs, and the DSP48 ends up with the configuration (AREG/BREG=0, MREG=0, PREG=1).

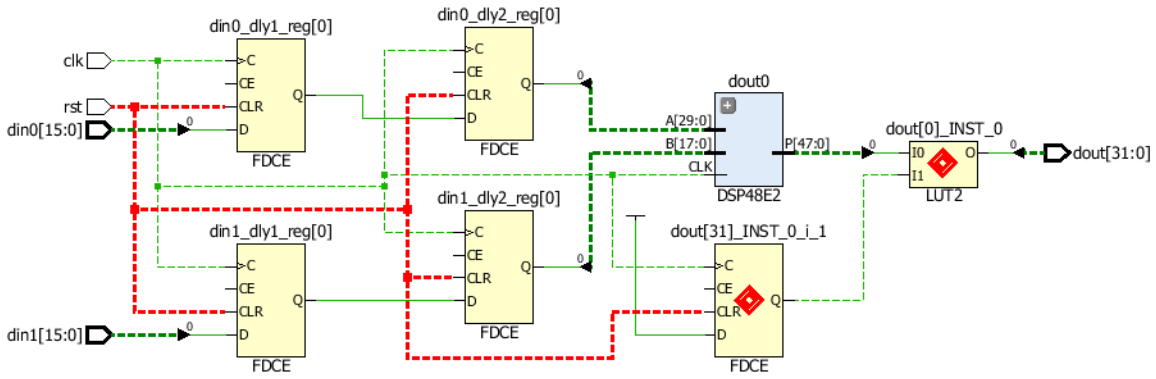



*Figure 3-2:* **Multiplier with Pipeline Registers Using Asynchronous Resets**

By simply changing the reset definition as shown in the following figure, such that the multiplier pipeline registers use a synchronous reset, synthesis can take advantage of the DSP48 internal registers (AREG/BREG=1, MREG=1, PREG=1).

```
always @ (posedge clk or posedge rst) begin
  if (rst) begin
    din0_dly1 <= 16'h0;
    din0_dly2 <= 16'h0;
    din1_dly1 <= 16'h0;
    din1_dly2 <= 16'h0;
    dout      <= 32'h0;
  end else begin
    din0_dly1 <= din0;
    din0_dly2 <= din0_dly1;
    din1_dly1 <= din1;
    din1_dly2 <= din1_dly1;
    dout      <= din0_dly2 * din1_dly2;
  end
end
```

```
always @ (posedge clk) begin
  if (rst) begin
    din0_dly1 <= 16'h0;
    din0_dly2 <= 16'h0;
    din1_dly1 <= 16'h0;
    din1_dly2 <= 16'h0;
    dout      <= 32'h0;
  end else begin
    din0_dly1 <= din0;
    din0_dly2 <= din0_dly1;
    din1_dly1 <= din1;
    din1_dly2 <= din1_dly1;
    dout      <= din0_dly2 * din1_dly2;
  end
end
```

*Figure 3-3:* **Changing Asynchronous Reset into Synchronous Reset on Multiplier**

Due to saving fabric resources and taking advantage of all DSP48 internal registers, the design performance and power efficiency are optimal.

### Reset Coding Example Two: Multiplier with Synchronous Reset

To take advantage of the existing DSP primitive features, the preceding example can be rewritten with a change from asynchronous reset to synchronous reset as follows.

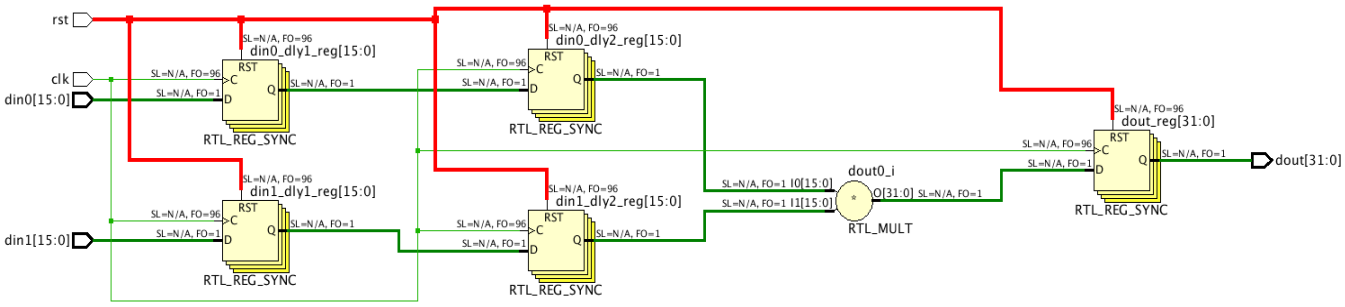



*Figure 3-4:* **Multiplier with Pipeline Registers (Synchronous Reset)**

In this circuit, the DSP48 primitive is inferred with all pipeline registers packed within the DSP primitive (AREG/BREG=1, MREG=1, PREG=1).

The implementation of the second coding example has the following advantages over the first coding example:

- Optimal resource usage
- Better performance and lower power
- Lower number of endpoints

In addition, the second coding example is more concise.

### Issues When Trying to Eliminate Reset in HDL Code

When optimizing the code to eliminate reset, commenting out the conditions within the reset declaration does not create the desired structures and instead creates issues. For example, the following figure shows three pipeline stages with asynchronous reset used for each. If you attempt to eliminate the reset condition for two of the pipeline stages by commenting out the code with the reset condition, the asynchronous reset becomes enabled (inverted logic of rst).

```
always@(posedge clk or posedge rst)
begin
    if(rst)
    begin
        //din_dly1 <= 16'b0;
        //din_dly2 <= 16'b0;
        dout     <= 16'b0;
    end
    else
    begin
      din_dly1 <= din;
      din_dly2 <= din_dly1;
      dout     <= din_dly2;
    end
end
```

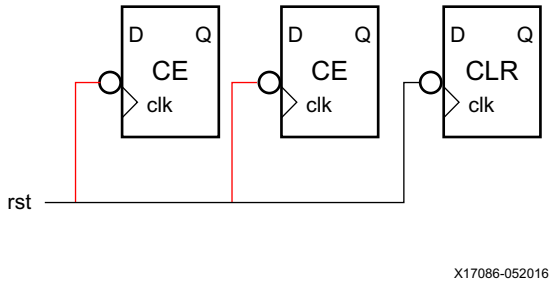



*Figure 3-5:* **Commenting Out Code with Reset Conditions**

The optimal way to remove the resets is to create separate sequential logic procedures with one for reset conditions and the other for non-reset conditions, as shown in the following figure.

```
always@(posedge clk)
begin
    din_dly1 <= din;
    din_dly2 <= din_dly1;
end

always@(posedge clk or posedge rst)
begin
    if(rst)
        dout <= 16'd0;
    else
        dout <= din_dly2;
end
```

*Figure 3-6:* **Separate Procedural Statements for Registers With and Without Reset**

**TIP:** *When using a reset, make sure that all registers in the procedural statement are reset.*

## *Clock Enables*

When used properly, clock enables can significantly reduce design power with little impact on area or performance. However, when clock enables are used improperly, they can lead to:

- Increased area
- Decreased density
- Increased power
- Reduced performance

In many designs with a large number of control sets, low fanout clock enables might be the main contributor to the number of control sets.

### Creating Clock Enables

Clock enables are created when an incomplete conditional statement is coded into a synchronous block. A clock enable is inferred to retain the last value when the prior conditions are not met. When this is the desired functionality, it is valid to code in this manner. However, in some cases when the prior conditional values are not met, the output is a don't care. In that case, Xilinx recommends closing off the conditional (that is, use an `else` clause), with a defined constant (that is, assign the signal to a one or a zero).

In most implementations, this does not result in added logic, and avoids the need for a clock enable. The exception to this rule is in the case of a large bus when inferring a clock enable in which the value is held can help in power reduction. The basic premise is that when small numbers of registers are inferred, a clock enable can be detrimental because it increases control set count. However, in larger groups, it can become more beneficial and is recommended.

### Reset and Clock Enable Precedence

In Xilinx FPGA devices, all registers are built to have set/reset take precedence over clock enable, whether an asynchronous or synchronous set/reset is described. In order to obtain the most optimal result, Xilinx recommends that you always code the set/reset before the enable (if deemed necessary) in the `if/else` constructs within a synchronous block. Coding a clock enable first forces the reset into the data path and creates additional logic.

For information on clocking, see Clocking Guidelines.

## *Controlling Enable/Reset Extraction with Synthesis Attributes*

You can force control set mapping by applying the DIRECT_RESET / DIRECT_ENABLE / EXTRACT_RESET / EXTRACT_ENABLE attributes as needed to handle the mapping of control sets for a given structure.

When the design includes a synchronous reset/enable, synthesis creates a logic cone mapped through the CE/R/S pins when the load is equal to or above the threshold set by the `-control_set_opt_threshold` synthesis switch, or creates a logic cone that maps through the D pin if below the threshold. The default thresholds are:

- 7 series devices: 4
- UltraScale devices: 2

### Using DIRECT_ENABLE and DIRECT_RESET

To use control set mapping you can apply attributes to the nets connected to enable/reset signals, which will force synthesis to use the CE/R pin.

In the following figure, the enable signal (en) is only connected to one flip-flop. Therefore, the synthesis engine connected the en signal to the FDRE/D pin cone of logic. Note that the CE pin is tied to logic 1.

```
module test
(
input clk,
input en,
input din,
output reg dout
);

always@(posedge clk)
begin
    if(en)
    begin
        dout <= din;
    end
end

endmodule
```

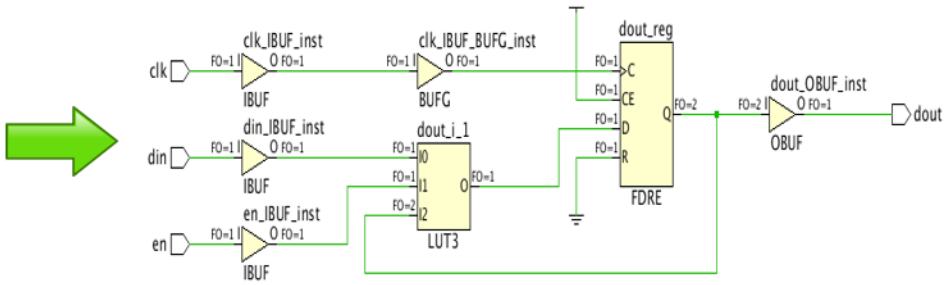



*Figure 3-7:* **Clock Enable Implementation Using Datapath Logic**

To override this default behavior, you can use the DIRECT_ENABLE attribute. For example, the following figure shows how to connect the enable signal (en) to the CE pin of the register by adding the DIRECT_ENABLE attribute to the port/signal.

```
module test
(
input clk,
(* direct_enable = "true" *) input en,
input din,
output reg dout
);

always@(posedge clk)
begin
    if(en)
    begin
        dout <= din;
    end
end

endmodule
```

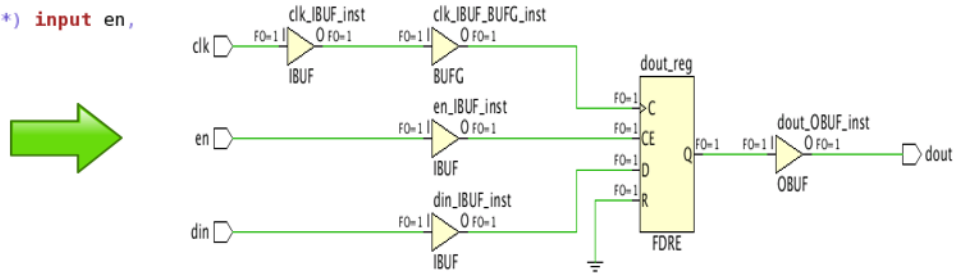



*Figure 3-8:* **Dedicated Clock Enable Implementation Using direct_enable**

The following figure shows RTL code in which either `global_rst` or `int_rst` can reset the register. By default, both are mapped to the reset pin cone of logic.

```
module test (
             input clk,
             input global_rst,
             input [1:0] conf,
             input din,
             output reg dout
            );

reg [1:0] conf_reg;

assign int_rst = &conf_reg;

always@(posedge clk)
begin
    conf_reg <= conf;
    if(global_rst || int_rst)
        dout <= 1'b0;
    else
        dout <= din;
end

endmodule
```

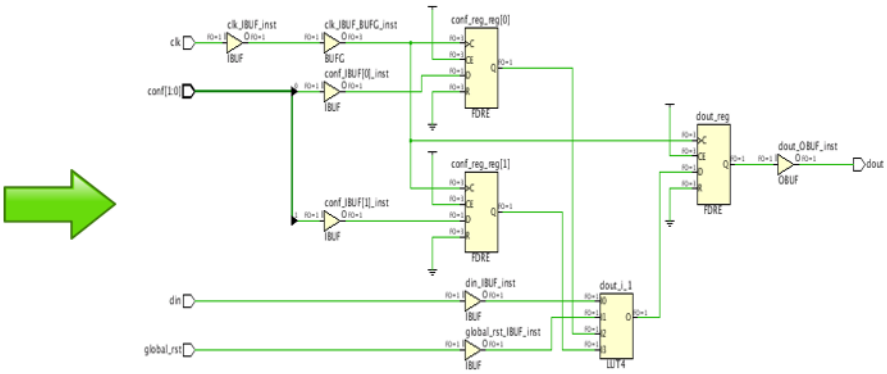


*Figure 3-9:* **Multiple Reset Conditions Mapped Through Datapath Logic**

You can use the DIRECT_RESET attribute to specify which reset signal to connect to the register reset pin. For example, the following figure shows how to use the DIRECT_RESET attribute to connect only the `global_rst` signal to the register FDRE/R pin and connect the `int_rst` signal to the FDRE/D cone of logic.

```
module test (
             input clk,
             (* direct_reset = "true" *) input global_rst,
             input [1:0] conf,
             input din,
             output reg dout
            );

reg [1:0] conf_reg;

assign int_rst = &conf_reg;

always@(posedge clk)
begin
    conf_reg <= conf;
    if(global_rst || int_rst)
        dout <= 1'b0;
    else
        dout <= din;
end

endmodule
```

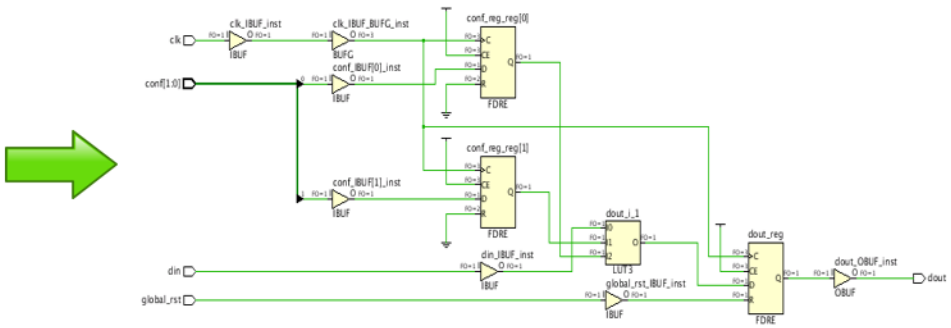


*Figure 3-10:* **Dedicated Reset Pin Usage Using DIRECT_RESET Attribute**

### Pushing the Logic from the Control Pin to the Data Pin

During analysis of critical paths, you might find multiple paths ending at control pins. You must analyze these paths to determine if there is a way to push the logic into the datapath without incurring penalties, such as extra logic levels. There is less delay in a path to the D pin than CE/R/S pins given the same levels of logic because there is a direct connection from the output of the last LUT to the D input of the FF. The following coding examples show how to push the logic from the control pin to the data pin of a register.

In the following example, the enable pin of dout_reg[0] has 2 logic levels, and the data pin has 0 logic levels. In this situation, you can improve timing by moving the enable logic to the D pin by setting the EXTRACT_ENABLE attribute to "no" on the dout register definition in the RTL file.

```
module test
(
input clk,
input [9:0] en,
input [7:0] din,
output reg [7:0] dout
);

wire en_tmp;
reg [7:0] din_reg;
reg [9:0] en_reg;

assign en_tmp = &en_reg;

always@(posedge clk)
begin
    en_reg <= en;
    din_reg <= din;
    if(en_tmp)
      dout <= din_reg;
end

endmodule
```

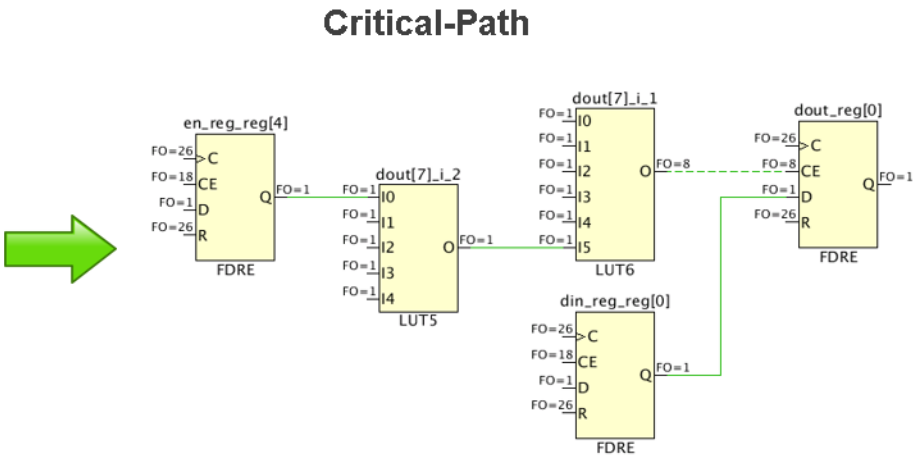



*Figure 3-11:* **Critical Path Ending at Control Pin (Enable) of a Register**

The following example shows how to separate the combinational and sequential logic and map the complete logic in to the datapath. This pushes the logic into the D pin, which still has 2 logic levels.

You can achieve the same structure by setting the EXTRACT_ENABLE attribute to "no." For more information on the EXTRACT_ENABLE attribute, see the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 16].

```
module test
(
input clk,
input [9:0] en,
input [7:0] din,
output reg [7:0] dout
);

wire en_tmp;
reg [7:0] din_reg;
reg [9:0] en_reg;
(* KEEP = "true" *) reg [7:0] dout_nxt;

assign en_tmp = &en_reg;

always@*
begin
    dout_nxt = dout;
    if(en_tmp)
      dout_nxt = din_reg;
end

always@(posedge clk)
begin
    en_reg <= en;
    din_reg <= din;
    dout <= dout_nxt;
end

endmodule
```

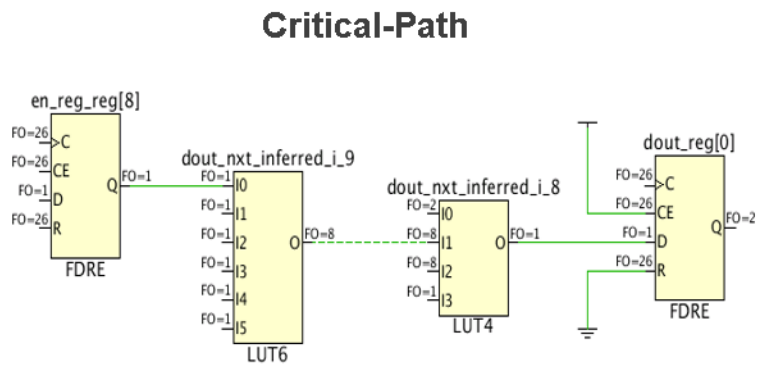



*Figure 3-12:* **Critical Path Ending at Data Pin of a Register (Disabling Enable Extraction)**

## *Tips for Control Signals*

- Check whether a global reset is really needed.
- Avoid asynchronous control signals.
- Keep clock, enable, and reset polarities consistent.
- Do not code a set and reset into the same register element.
- If an asynchronous reset is absolutely needed, remember to synchronize its deassertion.

## Know What You Infer

Your code finally has to map onto the resources present on the device. Make an effort to understand the key arithmetic, storage, and logic elements in the architecture you are targeting. Then, as you code the functionality of the design, anticipate the hardware resources to which the code will map. Understanding this mapping gives you an early insight into any potential problem.

The following examples demonstrate how understanding the hardware resources and mapping can help make certain design decisions:

- For larger than 4-bit addition, subtraction and add-sub, a carry chain is generally used and one LUT per 2-bit addition is used (that is, an 8-bit by 8-bit adder uses 8 LUTs and the associated carry chain). For ternary addition or in the case where the result of an adder is added to another value without the use of a register in between, one LUT per 3-bit addition is used (that is, an 8-bit by 8-bit by 8-bit addition also uses 8 LUTs and the associated carry chain).

  If more than one addition is needed, it may be advantageous to specify registers after every two levels of addition to cut device utilization in half by allowing a ternary implementation to be generated.

- In general, multiplication is targeted to DSP blocks. Signed bit widths less than 18x25 (18x27 in UltraScale devices) map into a single DSP Block. Multiplication requiring larger products may map into more than one DSP block. DSP blocks have pipelining resources inside them.

  Pipelining properly for logic inferred into the DSP block can greatly improve performance and power. When a multiplication is described, three levels of pipelining around it generates best setup, clock-to-out, and power characteristics. Extremely light pipelining (one-level or none) may lead to timing issues and increased power for those blocks, while the pipelining registers within the DSP lie unused.

- Two SRLs with depths of 16 bits or less can be mapped into a single LUT, and single SRLs up to 32 bits can also be mapped into a single LUT.
- For conditional code resulting in standard MUX components:
  - A 4-to-1 MUX can be implemented into a single LUT, resulting in one logic level.
  - An 8-to-1 MUX can be implemented into two LUTs and a MUXF7 component, still resulting in effectively one logic (LUT) level.
  - A 16-to-1 MUX can be implemented into four LUTs and a combination of MUXF7 and MUXF8 resources, still resulting in effectively one logic (LUT) level.

A combination of LUTs, MUXF7, and MUXF8 within the same CLB/slice structure results in a very small combinational delay. Hence, these combinations are considered as equivalent to only one logic level. Understanding this code can lead to better resource management, and can help in better appreciating and controlling logic levels for the data paths.

For general logic, take into account the number of unique inputs for a given register. From that number, an estimation of LUTs and logic levels can be achieved. In general, 6 inputs or fewer always results in a single logic level. Theoretically, two levels of logic can manage up to 36 inputs. However, for all practical purposes, you should assume that approximately 20 inputs is the maximum that can be managed with two levels of logic. In general, the larger the number of inputs and the more complex the logic equation, the more LUTs and logic levels are required.

**IMPORTANT:** *Check the availability of hardware resources and how efficiently they are being utilized early in the design cycle to enable easier modifications. This approach yields better results than waiting until late in the design cycle during timing closure.*

### *Inferring RAM and ROM*

RAM and ROM may be specified in multiple ways. Each has advantages and disadvantages.

- Inference

  Advantages:

  - Highly portable
  - Easy to read and understand
  - Self-documenting
  - Fast simulation

  Disadvantages:

  - Might not have access to all RAM configurations available
  - Might produce less optimal results

  Because inference usually gives good results, it is the recommended method, unless a given use is not supported, or it is not producing adequate results in performance, area, or power. In that case, explore other methods.

  When inferring RAM, Xilinx recommends that you use the HDL Templates provided in the Vivado tools. As mentioned earlier, using asynchronous reset impacts RAM inference, and should be avoided. See Using Vivado Design Suite HDL Templates.

- Xilinx Parameterizable Macros (XPMs)

  Advantages:

  - Portable between Xilinx device families
  - Fast simulation
  - Support for asymmetric width
  - Predictable QoR

  Disadvantages:

  - Limited to supported XPM options

  XPMs are built on inference using fixed templates that you cannot modify. Therefore, they can guarantee QoR and can support features that standard inference does not. When standard inference does not support the features required, Xilinx recommends you use XPMs instead.

  ***Note:*** When you compile simulation libraries using `compile_simlib`, XPMs are automatically compiled. For more information, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 12].

- Direct Instantiation of RAM Primitives

  Advantages:

  - Highest level control over implementation
  - Access to all capabilities of the block

  Disadvantages:

  - Less portable code
  - Wordier and more difficult to understand functionality and intent

- Core from IP Catalog

  Advantages:

  - Generally more optimized result when using multiple components
  - Simple to specify and configure

  Disadvantages:

  - Less portable code
  - Core management

### Performance Considerations When Implementing RAM

In order to efficiently infer memory elements, consider these factors affecting performance:

- Using Dedicated Blocks or Distributed RAMs

  RAMs can be implemented in either the dedicated block RAM or within LUTs using distributed RAM. The choice not only impacts resource selection, but can also significantly impact performance and power.

  In general, the required depth of the RAM is the first criterion. Memory arrays described up to 64 bits deep are generally implemented in LUTRAMs, where depths of 32 bits and less are mapped 2 bits per LUT and depths up to 64-bits can be mapped one bit per LUT. Deeper RAMs can also be implemented in LUTRAM depending on available resources and synthesis tool assignment.

  Memory arrays deeper than 256 bits are generally implemented in block memory. Xilinx FPGA devices have the flexibility to map such structures in different width and depth combinations. You should be familiar with these configurations in order to understand the number and structure of block RAMs used for larger memory array declarations in the code.

- Using the Output Pipeline Register

  Using an output register is required for high performance designs, and is recommended for all designs. This improves the clock to output timing of the block RAM. Additionally, a second output register is beneficial, as slice output registers have faster clock to out timing than a block RAM register. Having both registers has a total read latency of 3. When inferring these registers, they should be in the same level of hierarchy as the RAM array. This allows the tools to merge the block RAM output register into the primitive.

- Using the Input Pipeline Register

  When RAM arrays are large and mapped across many primitives, they can span a considerable area of the die. This can lead to performance issues on address and control lines. Consider adding an extra register after the generation of these signals and before the RAMs. To further improve timing, use `phys_opt_design` later in the flow to replicate this register. Registers without logic on the input will replicate more easily.

### Scenarios Preventing Block RAM Output Register Inference

Xilinx recommends that the memory and the output registers are all inferred in a single level of hierarchy, because this is the easiest method to ensure inference is as intended. There are two scenarios that will infer a block RAM output register. The first one is when an extra register exists on the output, and the second is when the read address register is retimed across the memory array. This can only happen using single port RAM. This is illustrated below:

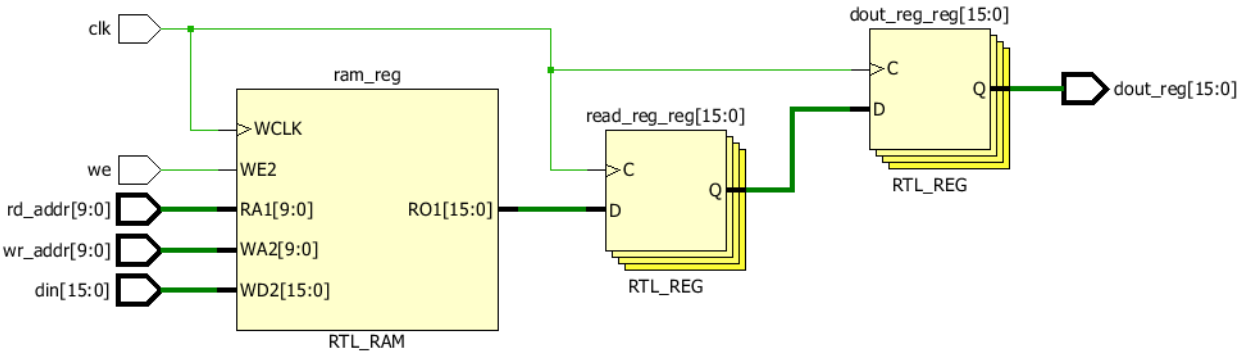



*Figure 3-13:* **RAM with Extra Read Register for Block RAM Output Register Inference**

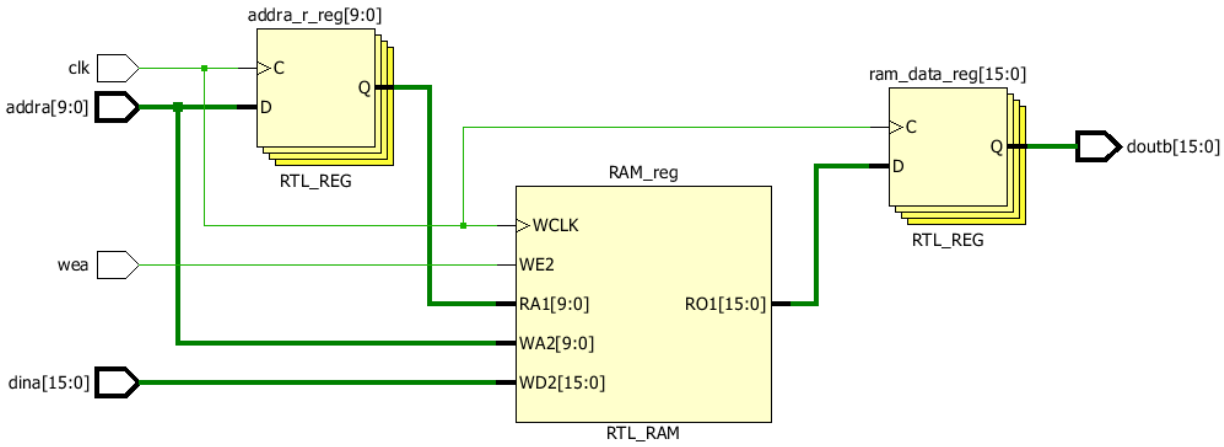



*Figure 3-14:* **View of RAM Before Address Register Retiming**

Certain deviations from these examples can prevent the inference of the output register.

### Checking for Multi-Fanout on the Output of Read Data Registers

The fanout of the data output bits from the memory array must be 1 for the second register to be absorbed by the RAM primitive. This is illustrated in the following figure.

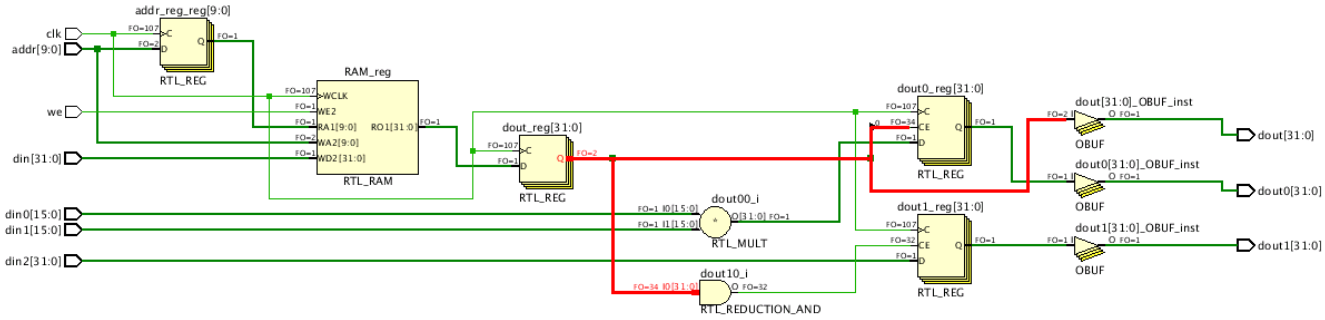



*Figure 3-15:* **Multiple Fanout Preventing Block RAM Output Register Inference**

### Checking for Reset Signals on the Address/Read Data Registers

Memory arrays should not be reset. Only the output of the RAM can tolerate a reset. The reset must be synchronous in order for inference of the output register into the RAM primitive. An asynchronous reset will cause the register to not be inferred into the RAM primitive. Additionally, the output signal can only be reset to 0.

The following figure highlights an example of what to avoid in order to ensure correct inference of RAMs and output registers.

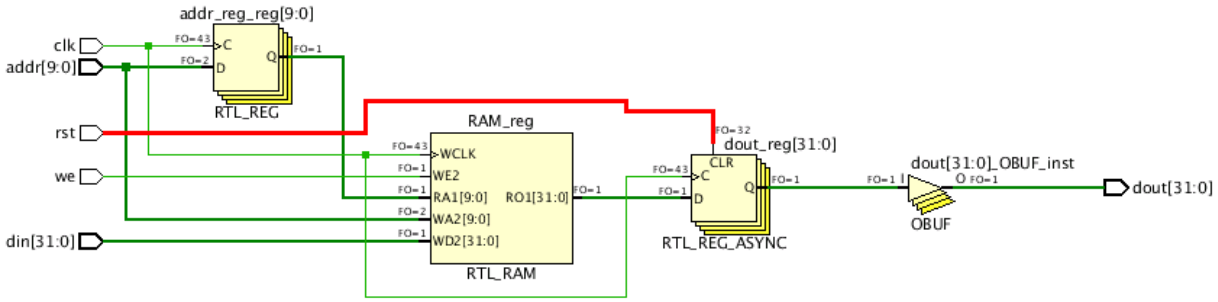



*Figure 3-16:* **Checking for Reset On Address/Read Data Registers**

### Checking for Feedback Structures in Registers

Make sure that registers do not have feedback logic, in the example below, since the adder requires the current value of register, this logic cannot be retimed and packed in to a block RAM. The resultant circuit is a block RAM without output registers (DOA_REG and DOB_REG set to '0').

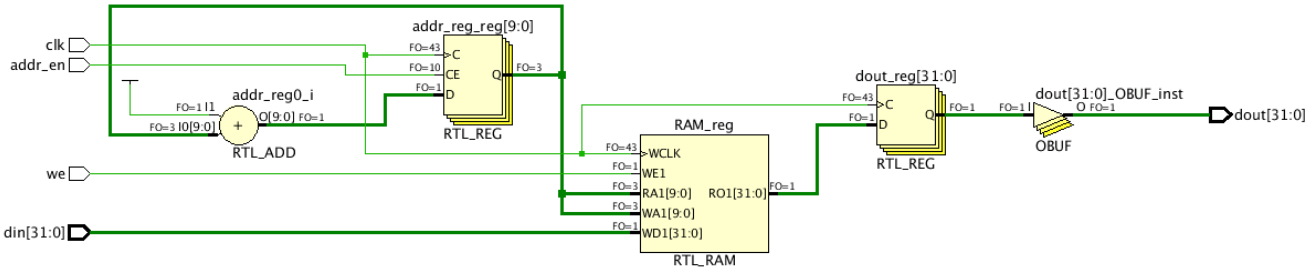



*Figure 3-17:* **Check the Presence of Feedback on Registers Around the RAM Block**

### Mapping Memories to UltraRAM Blocks

UltraRAM is a 4Kx72 memory block with two ports using a single clock. This primitive is only available in certain UltraScale+™ devices. In these devices, UltraRAM is included in addition to block RAM resources.

UltraRAM can be used in your design using one of the following methods:

- Rely on synthesis to infer UltraRAMs by setting the `ram_style = "ultra"` attribute on a memory declaration in HDL.
- Instantiate Xilinx XPM_MEMORY primitives.
- Instantiate UltraRAM UNISIM primitives.

The following code example shows the instantiation of XPM memory and is available in the HDL Language templates. Highlighted parameters `MEMORY_PRIMITIVE` and `READ_LATENCY` are the key parameters to infer memory as UltraRAM for high performance.

- `MEMORY_PRIMITIVE = "ultra"` specifies the memory is to be inferred as UltraRAM.
- `READ_LATENCY` defines the number of pipeline registers present on the output of the memory.

Larger memories are mapped to an UltraRAM matrix consisting of multiple UltraRAM cells configured as row x column structures.

A matrix can be created with single or multiple columns based on the depth. The current default threshold for URAM column height is 8 and it can be controlled with the attribute CASCADE_HEIGHT.

The difference between single column and multiple column UltraRAM matrix is as follows:

- Single column UltraRAM matrix uses the built-in hardware cascade without fabric logic.
- Multiple column UltraRAM matrix uses built-in hardware cascade within each column, plus some fabric logic for connecting the columns. Extra pipelining may be required to maintain performance. This is inferred by increasing the read latency. Vivado automatically packs these registers into UltraRAM as required.

```
xpm_memory_spram # (

  // Common module parameters
  .MEMORY_SIZE        (8*(4096*72)),     //positive integer
  .MEMORY_PRIMITIVE   ("ultra"),         //string; "auto", "distributed", "block" or "ultra";
  .MEMORY_INIT_FILE   ("none"),          //string; "none" or "<filename>.mem"
  .MEMORY_INIT_PARAM  (""    ),          //string;
  .USE_MEM_INIT       (0),               //integer; 0,1
  .WAKEUP_TIME        ("disable_sleep"),//string; "disable_sleep" or "use_sleep_pin"
  .MESSAGE_CONTROL    (0),               //integer; 0,1

  // Port A module parameters
  .WRITE_DATA_WIDTH_A (72),              //positive integer
  .READ_DATA_WIDTH_A  (72),              //positive integer
  .BYTE_WRITE_WIDTH_A (72),              //integer; 8, 9, or WRITE_DATA_WIDTH_A value
  .ADDR_WIDTH_A       (16),               //positive integer
  .READ_RESET_VALUE_A ("0"),             //string
  .READ_LATENCY_A     (9),               //non-negative integer
  .WRITE_MODE_A       ("read_first")     //string; "write_first", "read_first", "no_change"

) xpm_memory_spram_inst (

  // Common module ports
  .sleep          (1'b0),

  // Port A module ports
  .clka           (clka),
  .rsta           (rsta),
  .ena            (ena),
  .regcea         (regcea),
  .wea            (wea),
  .addra          (addra),
  .dina           (dina),
  .injectsbiterra (1'b0),  //do not change
  .injectdbiterra (1'b0),  //do not change
  .douta          (douta),
  .sbiterra       (),      //do not change
  .dbiterra       ()       //do not change

);
```

*Figure 3-18:* **Specifying UltraRAM in RTL Code (via XPM)**

The example above uses a 32 K x 72 memory configuration, which uses eight URAMs. To increase performance of the UltraRAM, more pipelining registers should be added to the cascade chain. This is achieved by increasing the read latency integer.

For more information on inferring UltraRAM in Vivado synthesis, see this link in the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 16].

### *Coding for Optimal DSP and Arithmetic Inference*

The DSP blocks within the Xilinx FPGA devices can perform many different functions, including:

- Multiplication
- Addition and subtraction
- Comparators
- Counters
- General logic

The DSP blocks are highly pipelined blocks with multiple register stages allowing for high-speed operation while reducing the overall power footprint of the resource. Xilinx recommends that you fully pipeline the code intended to map into the DSP48, so that all pipeline stages are utilized. To allow the flexibility of use of this additional resource, a set condition cannot exist in the function for it to properly map to this resource.

DSP48 slice registers within Xilinx devices contain only resets, and not sets. Accordingly, unless necessary, do not code a set (value equals logic 1 upon an applied signal) around multipliers, adders, counters, or other logic that can be implemented within a DSP48 slice. Additionally, avoid asynchronous resets, since the DSP slice only supports synchronous reset operations. Code resulting in sets or asynchronous resets may produce sub-optimal results in terms of area, performance, or power.

Many DSP designs are well-suited for the Xilinx architecture. To obtain best use of the architecture, you must be familiar with the underlying features and capabilities so that design entry code can take advantage of these resources.

The DSP48 blocks use a signed arithmetic implementation. Xilinx recommends code using signed values in the HDL source to best match the resource capabilities and, in general, obtain the most efficient mapping. If unsigned bus values are used in the code, the synthesis tools may still be able to use this resource, but might not obtain the full bit precision of the component due to the unsigned-to-signed conversion.

If the target design is expected to contain a large number of adders, Xilinx recommends that you evaluate the design to make greater use of the DSP48 slice pre-adders and post-adders. For example, with FIR filters, the adder cascade can be used to build a systolic filter rather than using multiple successive add functions (adder trees). If the filter is symmetric, you can evaluate using the dedicated pre-adder to further consolidate the function into both fewer LUTs and flip-flops and also fewer DSP slices as well (in most cases, half the resources).

If adder trees are necessary, the 6-input LUT architecture can efficiently create ternary addition (A + B + C = D) using the same amount of resources as a simple 2-input addition. This can help save and conserve carry logic resources. In many cases, there is no need to use these techniques.

By knowing these capabilities, the proper trade-offs can be acknowledged up front and accounted for in the RTL code to allow for a smoother and more efficient implementation from the start. In most cases, Xilinx recommends inferring DSP resources

For more information about the features and capabilities of the DSP48 slice, and how to best leverage this resource for your design needs, see the *7 Series DSP48E1 Slice User Guide* (UG479) [Ref 44] and *UltraScale Architecture DSP Slice User Guide* (UG579) [Ref 45].

### *Coding Shift Registers and Delay Lines*

In general, a shift register is characterized by some or all of the following control and data signals:

- Clock
- Serial input
- Asynchronous set/reset
- Synchronous set/reset
- Synchronous/asynchronous parallel load
- Clock enable
- Serial or parallel output

Xilinx FPGA devices contain dedicated SRL16 and SRL32 resources (integrated in LUTs). These allow efficiently implemented shift registers without using flip-flop resources. However, these elements support only LEFT shift operations, and have a limited number of I/O signals:

- Clock
- Clock Enable
- Serial Data In
- Serial Data Out

In addition, SRLs have address inputs (A3, A2, A1, A0 inputs for SRL16) determining the length of the shift register. The shift register may be of a fixed static length, or it may be dynamically adjusted.

In dynamic mode each time a new address is applied to the address pins, the new bit position value is available on the Q output after the time delay to access the LUT. Synchronous and asynchronous set/reset control signals are not available in the SRL primitives. However, if your RTL code includes a reset, the Xilinx synthesis tool infers additional logic around the SRL to provide the reset functionality.

To obtain the best performance when using SRLs, Xilinx recommends that you implement the last stage of the shift register in the dedicated Slice register. The Slice registers have a

better clock-to-out time than SRLs. This allows some additional slack for the paths sourced by the shift register logic. Synthesis tools will automatically infer this register unless this resource is instantiated or the synthesis tool is prevented from inferring such a register because of attributes or cross hierarchy boundary optimization restrictions.

Xilinx recommends that you use the HDL coding styles represented in the Vivado Design Suite HDL Templates.

When using registers to obtain placement flexibility in the chip, turn off SRL inference using the attribute:

```
SHREG_EXTRACT = "no"
```

For more information about synthesis attributes and how to specify those attributes in the HDL code, see the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 16].

### *Initialization of All Inferred Registers, SRLs, and Memories*

The GSR net initializes all registers to the specified initial value in the HDL code. If no initial value is supplied, the synthesis tool is at liberty to assign the initial state to either zero or one. Vivado synthesis generally defaults to zero with a few exceptions such as one-hot state machine encodings.

Any inferred SRL, memory, or other synchronous element may also have an initial state defined that will be programmed into the associated element upon configuration.

Xilinx highly recommends that you initialize all synchronous elements accordingly. Initialization of registers is completely inferable by all major FPGA synthesis tools. This lessens the need to add a reset for the sole purpose of initialization, and makes the RTL code more closely match the implemented design in functional simulation, as all synchronous element start with a known value in the FPGA device after configuration.

Initial state of the registers and latches VHDL coding example one:

```
signal reg1 : std_logic := '0'; -- specifying register1 to start as a zero
signal reg2 : std_logic := '1'; -- specifying register2 to start as a one
signal reg3 : std_logic_vector(3 downto 0):="1011"; -- specifying INIT value for
4-bit register
```

Initial state of the registers and latches Verilog coding example one:

```
reg register1 = 1'b0; // specifying regsiter1 to start as a zero
reg register2 = 1'b1; // specifying register2 to start as a one
reg [3:0] register3 = 4'b1011; //specifying INIT value for 4-bit register
```

Initial state of the registers and latches Verilog coding example two

Another possibility in Verilog is to use an `initial` statement:

```
reg [3:0] register3;
initial begin
  register3= 4'b1011;
end
```

## *Deciding When to Instantiate or Infer*

Xilinx recommends that you have an RTL description of your design; and that you let the synthesis tool do the mapping of the code into the resources available in the FPGA device. In addition to making the code more portable, all inferred logic is visible to the synthesis tool, allowing the tool to perform optimizations between functions. These optimizations include logic replications; restructuring and merging; and retiming to balance logic delay between registers.

### Synthesis Tool Optimization

When device library cells are instantiated, synthesis tools do not optimize them by default. Even when instructed to optimize the device library cells, synthesis tools generally cannot perform the same level of optimization as with the RTL. Therefore, synthesis tools typically only perform optimizations on the paths to and from these cells but not through the cells.

For example, if an SRL is instantiated and is part of a long path, this path might become a bottleneck. The SRL has a longer clock-to-out delay than a regular register. To preserve the area reduction provided by the SRL while improving its clock-to-out performance, an SRL of one delay less than the actual desired delay is created, with the last stage implemented in a regular flip-flop.

### When Instantiation Is Desirable

Instantiation may be desirable when the synthesis tool mapping does not meet the timing, power, or area constraints; or when a particular feature within an FPGA device cannot be inferred.

With instantiation, you have total control over the synthesis tool. For example, to achieve better performance, you can implement a comparator using only LUTs, instead of the combination of LUT and carry chain elements usually chosen by the synthesis tool.

Sometimes instantiation may be the only way to make use of the complex resources available in the device. This can be due to:

- HDL Language Restrictions

  For example, it is not possible to describe double data rate (DDR) outputs in VHDL because it requires two separate processes to drive the same signal.

- Hardware Complexity

  It is easier to instantiate the I/O SerDes elements than to create synthesizable description.

- Synthesis Tools Inference Limitations

  For example, synthesis tools currently do not have the capability to infer the hard FIFOs from RTL descriptions. Therefore, you must instantiate them.

  If you decide to instantiate a Xilinx primitive, see the appropriate User Guide and Libraries Guide for the target architecture to fully understand the component functionality, configuration, and connectivity.

  In case of both inference as well as instantiation, Xilinx recommends that you use the instantiation and language templates from the Vivado Design Suite language templates.

**TIPS:**
- *Infer functionality whenever possible.*
- *When synthesized RTL code does not meet requirements, review the requirements before replacing the code with device library component instantiations.*
- *Consider the Vivado Design Suite language templates when writing common Verilog and VHDL behavioral constructs or if necessary instantiating the desired primitives.*

## Coding Styles to Improve Performance

For high performance designs, the coding techniques discussed in this section (Coding Styles to Improve Performance) can mitigate possible timing hazards.

### *High Fanouts in Critical Paths*

High fanout nets are much easier to deal with early in the design process. What constitutes too high of a fanout is often dictated by performance requirements and the construction of the paths. You can use the following techniques to address issues with high fanout nets.

**RECOMMENDED:** *Identify high fanout nets using the* `report_high_fanout_nets` *Tcl command after synthesis. Monitor the impact of these nets on design performance as you progress through the implementation process.*

#### Reduce Loads in Portions of the Design That Do Not Require It

For high fanout control signals, evaluate whether all coded portions of the design require that net. Reducing the number of loads can greatly reduce timing problems.

### Use Register Replication

Register replication can increase the speed of critical paths by making copies of registers to reduce the fanout of a given signal. This gives the implementation tools more flexibility in placing and routing the different loads and associated logic. Synthesis tools use this technique extensively.

Most synthesis tools use a fanout threshold limit to automatically determine whether to duplicate a register. Lowering this global threshold allows automatic duplication of high fanout nets. However, it does not allow control over which registers are duplicated or how their loads are grouped. In addition, the global replication mechanism does not assess timing slack accurately, which can lead to unnecessary replicated cells, logic utilization increase, and potentially higher power consumption.

Often, a better approach to reducing fanout is to use a balanced tree for the high fanout signals. Consider manually replicating registers based on the design hierarchy, because the cells included in a hierarchy are often placed together. For example, in the balanced reset tree shown in the following figure, the high fanout reset FF RST2 is replicated in RTL to balance the fanout across the different modules. If required, physical synthesis can perform further replication to improve WNS based on placement information.

**TIP:** *To preserve the duplicate registers in synthesis, use a KEEP attribute instead of DONT_TOUCH. A DONT_TOUCH attribute prevents further optimization during physical optimization later in the implementation flow.*

***Note:*** If a LUT1 rather than a register is replicated, it indicates that an attribute or constraint is applied incorrectly.

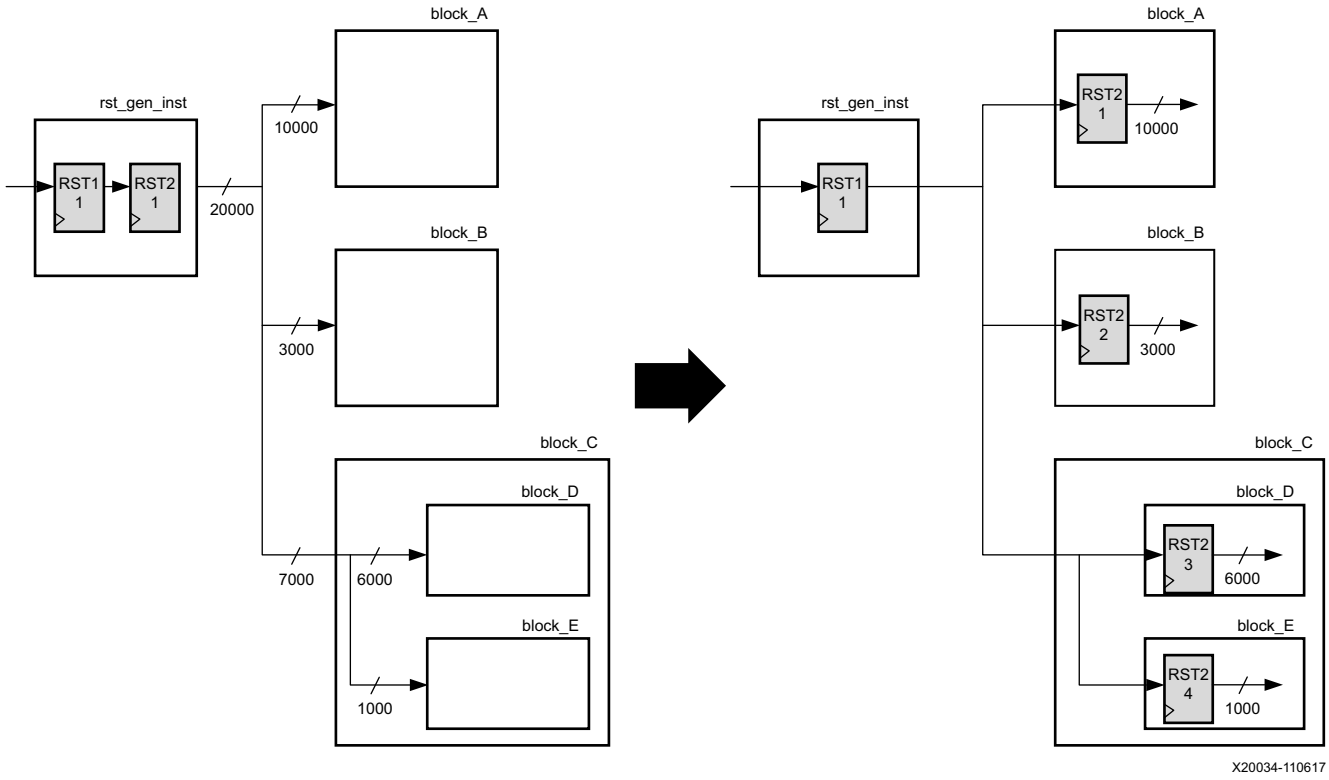



*Figure 3-19:* **High Fanout Reset Transformed to Balanced Reset Tree**

**RECOMMENDED:** *Using MAX_FANOUT attributes on global high fanout signals leads to sub-optimal replication similar to when the global fanout limit is lowered in synthesis. For this reason, Xilinx recommends only using MAX_FANOUT inside the hierarchies on local signals with medium to low fanout.*

Do not replicate registers used for synchronizing signals that cross clock domains. The presence of the ASYNC_REG attribute on these registers prevents the tool from replicating these registers. If the synchronizing chain has a very high fanout and replication must meet

timing, add an extra register after the synchronization chain that does not have the ASYNC_REG constraint.

The following table provides guidelines on the number of fanouts that might be acceptable for your design.

*Table 3-1:* **Fanout Guidelines for Medium Performance 7 Series Devices**

| Condition | Fanout > 5000 | Fanout > 200 | Fanout > 100 |
|---|---|---|---|
| **Low Frequency 1 to 125 MHZ** | Few logic levels between synchronous logic<br><br><13 levels of logic at maximum frequency | N/A | N/A |
| **Medium Frequency 125 to 250 MHz** | If the design does not meet timing, you might need to reduce fanout and/or logic levels. | <6 levels of logic at maximum frequency. (Driver and load types impact performance.) | N/A |
| **High Frequency > 250 MHz** | Not recommended for most designs. | Small number of logic levels is typically necessary for higher speeds. | Advance pipelining methods required. Careful logic replication. Compact functions. Low logic levels required. (Driver and load types impact performance.) |

**TIP:** *If the timing reports indicate that high-fanout signals are limiting the design performance, consider replicating the signals using the implementation tool options, such as* `opt_design -hier_fanout_limit`, `place_design -fanout_opt`, *and* `phys_opt_design`.

**TIP:** *When replicating registers, consider using a naming convention for the registers, such as* `<original_name>_a`, `<original_name>_b`, *etc., to make it easier to understand intent of the replication and easier to maintain the RTL code.*

## *Pipelining Considerations*

Another way to increase performance is to restructure long datapaths with several levels of logic and distribute them over multiple clock cycles. This method allows for a faster clock cycle and increased data throughput at the expense of latency and pipeline overhead logic management.

Because FPGA devices contain many registers, the additional registers and overhead logic are usually not an issue. However, the datapath spans multiple cycles, and you must make special considerations for the rest of the design to account for the added path latency.

### Consider Pipelining for SSI Devices

When designing high performance register-to-register connections for SLR boundary crossings, the appropriate pipelining must be described in the HDL code and controlled at synthesis. This ensures that the Shift Register LUT (SRL) inference and other optimizations do not occur in the logic path that must cross an SLR boundary. Modifying the code in this manner along with appropriate use of Pblocks defines where the SLR boundary crossing occurs.

### Consider Pipelining Up Front

Considering pipelining up front rather than later on can improve timing closure. Adding pipelining at a later stage to certain paths often propagates latency differences across the circuit. This can make one seemingly small change require a major redesign of portions of the code.

Identifying pipelining opportunities early in the design can often significantly improve timing closure, implementation runtime (due to easier-to-solve timing problems), and device power (due to reduced switching of logic).

### Check Inferred Logic

As you code your design, be aware of the logic being inferred. Monitor the following conditions for additional pipelining considerations:

- Cones of logic with large fanin

  For example, code that requires large buses or several combinational signals to compute an output

- Blocks with restricted placement or slow clock-to-out or large setup requirements

  For example, block RAMs without output registers or arithmetic code that is not appropriately pipelined

- Forced placement that causes long routes

  For example, a pinout that forces a route across the chip might require pipelining to allow for high-speed operation

- Logic comprised of large XOR functions

  Large XOR functions often have high switch rates that can generate large dynamic power dissipation. Pipelining these functions can reduce switching, which positively impacts power consumption of the described circuit.

In the following figure the clock speed is limited by:

- Clock-to out-time of the source flip-flop
- Logic delay through four levels of logic
- Routing associated with the four function generators
- Setup time of the destination register

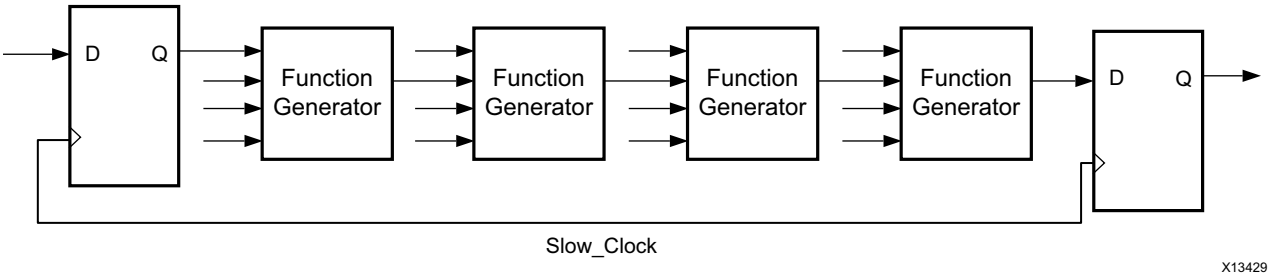



*Figure 3-20:* **Before Pipelining Diagram**

The following figure is an example of the same data path shown in Figure 3-20. Because the flip-flop is contained in the same slice as the function generator, the clock speed is limited by the clock-to-out time of the source flip-flop, the logic delay through one level of logic, one routing delay, and the setup time of the destination register. In this example, the system clock runs faster after pipelining than before pipelining.

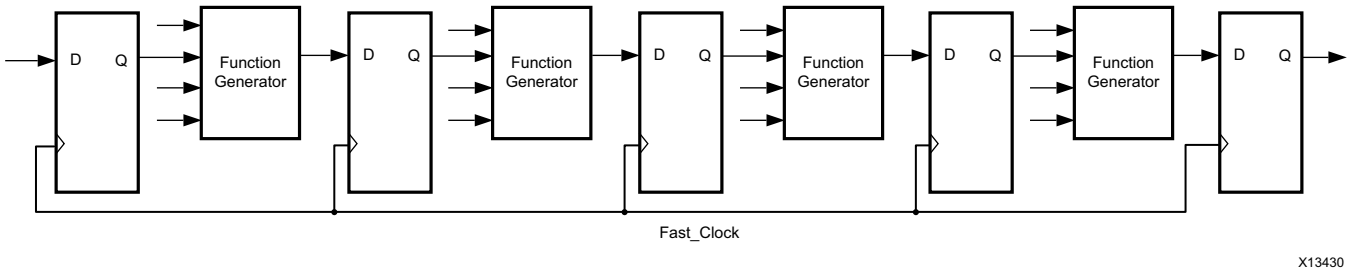



*Figure 3-21:* **After Pipelining Diagram**

### Determine Whether Pipelining is Needed

A commonly used pipelining technique is to identify a large combinatorial logic path, break it into smaller paths, and introduce a register stage between these paths, ideally balancing each pipeline stage.

To determine whether a design requires pipelining, identify the frequency of the clocks and the amount of logic distributed across each of the clock groups. You can use the `report_design_analysis` Tcl command with the `-logic_level_distribution` option to determine the logic-level distribution for each of the clock groups.

**TIP:** *The design analysis report also highlights the number of paths with zero logic levels, which you can use to determine where to make modifications in your code.*

### Balance Latency

To balance the latency by adding pipeline stages, add the stage to the control path and not the data path. The data path includes wider buses, which increases the number of flip-flop and register resources used.

For example, if you have a 128-bit data path, 2 stages of registers, and a requirement of 5 cycles of latency, inserting 3 register stages results in an extra 3 x 128 = 384 flip-flops. Alternatively, you can use registers to control logic to enable the data path. Use 5 stages of single-bit registers to control the enable signal of datapath flip-flops and multicycle path timing exceptions accordingly.

***Note:*** This example is only possible for certain designs. For example, in cases where there is a fanout from the intermediate data path flip-flops, having only 2 stages does not work.

**RECOMMENDED:** *The optimal LUT:FF ratio in an FPGA is 1:1. Designs with significantly more FFs will increase unrelated logic packing into slices, which will increase routing complexity and can degrade QoR.*

### Balance Pipeline Depth and SRL Usage

When there are deep register pipelines, map as many registers as possible into the SRLs to avoid significant increases in register utilization. For example, a 9-deep pipeline for a data width of 32 results in 9 registers for each bit, which uses 32 x 9 = 288 registers. Mapping the same structure to SRLs uses 32 SRLs. Each SRL has address pins A4 through A0 connected to 5'b01000 to implement a depth of 9 stages.

There are multiple ways to infer SRLs during synthesis, including the following:

- SRL
- REG -> SRL
- SRL -> REG
- REG -> SRL -> REG

You can create these structures using the `srl_style` attribute in the RTL code as follows:

- `(* srl_style = "srl" *)`
- `(* srl_style = "reg_srl" *)`
- `(* srl_style = "srl_reg" *)`
- `(* srl_style = "reg_srl_reg" *)`

A common mistake is to use different enable/reset control signals in deeper pipeline stages. Following is an example of a reset used in a 9-deep pipeline stage with the reset connected to the third, fifth, and eighth pipeline stages. With this structure, the tools map the pipeline stages to registers only, because there is a reset pin on the SRL primitive.

```
FF->FF->FF(reset) -> FF->FF(reset)->FF->FF->FF(reset)->FF
```

To take advantage of SRL inference:

- Ensure there are no resets for the pipeline stages.
- Analyze whether the reset is really required.
- Use the reset on one flip-flop (for example, on the first or last stage of the pipeline).

**Avoid Unnecessary Pipelining**

For highly utilized designs, too much pipelining can lead to sub-optimal results. For example, unnecessary pipeline stages increase the number of flip-flops and routing resources, which might limit the place and route algorithms if the utilization is high.

***Note:*** If there are many paths with 0/1 levels of logic, check to make sure this is intentional.

**Consider Pipelining Macro Primitives**

Based on the target architecture, dedicated primitives such as block RAMs and DSPs can work at over 500 MHz if enough pipelining is used. For high frequency designs, Xilinx recommends using all of the pipelines within these blocks.

## Coding Styles to Improve Power

Coding styles to improve power include:

### *Gate Clock or Data Paths*

Gating the clock or data paths is a common technique to stop transition when the results of these paths are not used. Gating a clock stops all driven synchronous loads and prevents data path signal switching and glitches from continuing to propagate.

Power optimization (`power_opt_design`) can automatically generate signal gating logic to reduce switching activity. However, you have information about the application, data flow, and dependencies that is not available to the tool, which only you can specify.

### *Maximize Gating Elements*

Maximize the number of elements affected by the gating signal. For example, it is more power efficient to gate a clock domain at its driving source than to gate each load with a clock enable signal.

### *Use Clock Enable Pins of Dedicated Clock Buffers*

When gating or multiplexing clocks to minimize activity or clock tree usage, use the clock enable ports of dedicated clock buffers. Inserting LUTs or using other methods to gate-off clock signals is not efficient for power and timing.

### *Use Case Block When Priority Encoder Not Needed*

When a priority encoding is not needed, use a case block instead of an if-then-else block or ternary operator.

Inefficient coding example

```
if (reg1)
  val = reg_in1;
else if (reg2)
   val = reg_in2;
else if (reg3)
   val = reg_in3;
else val = reg_in4;
```

Correct coding example

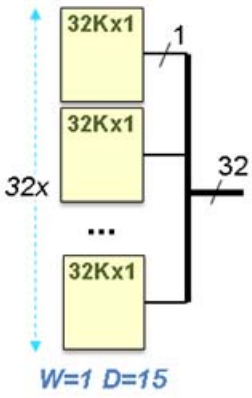
```
(* parallel_case *) casex  ({reg1, reg2, reg3})
1xx: val = reg_in1 ;
01x: val = reg_in2 ;
001: val = reg_in3 ;
default: val = reg_in4 ;
endcase
```

## Performance/Power Trade-off for Block RAMs

There are multiple ways of breaking a memory configuration to serve a particular requirement. The requirement for a particular design can be performance, power, or a mixture of both.

The following example highlights the different structures that can be generated to achieve your requirements. Synthesis can limit the cascading of the block RAM for the performance/power trade-off using the CASCADE_HEIGHT attribute, for UltraScale and later devices. The usage and arguments for the attribute are described in the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 16].

The following figure shows an example of 32Kx32 memory configuration for higher performance (timing).

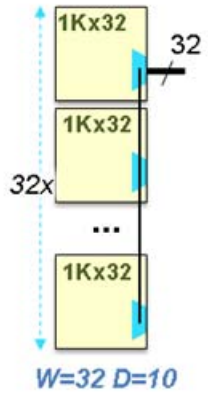



```
module test
(
input  clk,
input  we,
input  [14:0] addr,
input  [31:0] din,
output reg [31:0] dout
);

(* ram_style = "block", cascade_height = 1 *)
reg  [31:0] mem [(2**15)-1:0];
reg  [14:0] addr_reg;

always @(posedge clk)
begin
    addr_reg <= addr;
    dout <= mem[addr_reg];
    if (we)
        mem[addr_reg] <= din;
end

endmodule
```

*Figure 3-22:* **RTL Representation of 32Kx32 Using 32Kx1 and CASCADE_HEIGHT=1**

In this implementation, all block RAMs are always enabled (for each read or write) and consume more power.

The following figure shows an example of cascading all the block RAMs for low power.

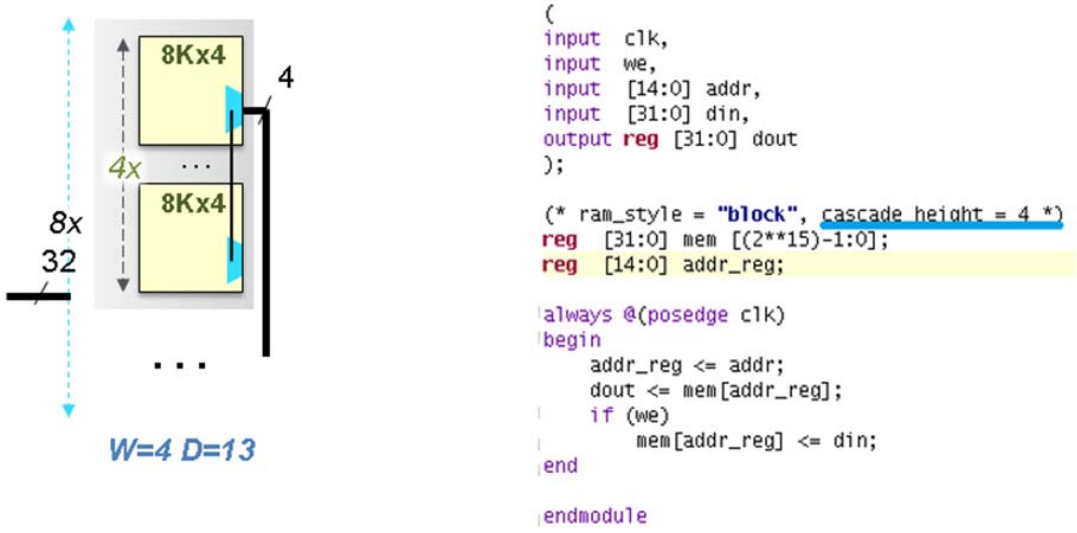



```
module test
(
input  clk,
input  we,
input  [14:0] addr,
input  [31:0] din,
output reg [31:0] dout
);

(* ram_style = "block", cascade_height = 32 *)
reg  [31:0] mem [(2**15)-1:0];
reg  [14:0] addr_reg;

always @(posedge clk)
begin
    addr_reg <= addr;
    dout <= mem[addr_reg];
    if (we)
        mem[addr_reg] <= din;
end

endmodule
```

*Figure 3-23:* **RTL Representation of 32Kx32 Using 1Kx32 and CASCADE_HEIGHT=32**

In this implementation, because one block RAM at a time is selected (from each unit), the dynamic power contribution is almost half. UltraScale device block RAMs have a dedicated cascade MUX and routing structure that allows the construction of wide, deep memories requiring more than one block RAM primitive to be built in a very power efficient configuration.

The following figure shows an example of how to limit the cascading and gain both power and performance at the same time, often with no trade-off in performance.




```
module test
(
input  clk,
input  we,
input  [14:0] addr,
input  [31:0] din,
output reg [31:0] dout
);

(* ram_style = "block", cascade_height = 4 *)
reg  [31:0] mem [(2**15)-1:0];
reg  [14:0] addr_reg;

always @(posedge clk)
begin
     addr_reg <= addr;
     dout <= mem[addr_reg];
     if (we)
          mem[addr_reg] <= din;
end

endmodule
```

*Figure 3-24:* **RTL Representation of 32Kx32 Using 8Kx4 and CASCADE_HEIGHT=4**

Because 8 block RAMs are selected at a time in this implementation, the dynamic power contribution is better than for the high performance structure, but not as good as for the low power structure. The advantage with this structure compared to a low power structure is that it uses only 4 block RAMs in the cascaded path, which has impact on the target frequency when compared to 32 block RAMs in the critical path for the low power structure.

## Decomposing Deeper Memory Configurations for Balanced Power and Performance

When working with deeper memory configurations, you can use the RAM_DECOMP synthesis attribute in the RTL to reduce power by improving memory composition. When the RAM_DECOMP attribute is applied to a memory array, the memory logic is mapped to a wider array of block RAM primitives. To balance power and performance, you can control cascading using the CASCADE_HEIGHT attribute along with the RAM_DECOMP attribute. This approach requires more address decoding logic but helps to reduce the number of block RAMs that are enabled for each read operation, which helps to reduce power.

For example, the following figure shows a 32x16K memory configuration.

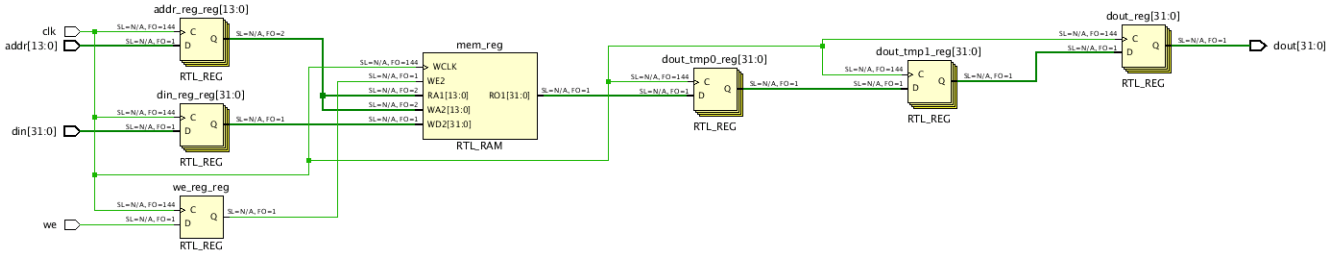



*Figure 3-25:* **32x16K Memory Configuration**

If you apply the following attributes:

```
ram_decomp = "power"
cascade_height = 4
```

16 RAMB36E2 is inferred and the memory is decomposed as follows:

- The base primitive is 32x1K.
- 4 block RAMs are cascaded to create a 32x4K configuration.
- 4 parallel structures create a 16K deep memory.
- The outputs are multiplexed to generate the output data.

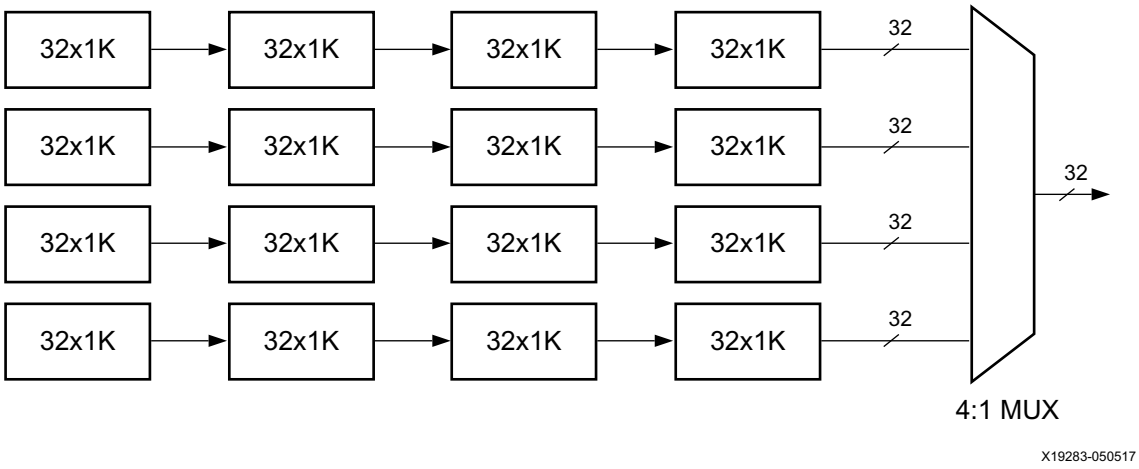



*Figure 3-26:* **Generated Structure for 32x16K Memory Configuration Example Using CASCADE_HEIGHT and RAM_DECOMP Attributes**

The following RTL code example shows the use of the CASCADE_HEIGHT and RAM_DECOMP attributes.

```
module test
(
input  clk,
input  we,
input  [13:0] addr,
input  [31:0] din,
output reg [31:0] dout
);

(* ram_style = "block", ram_decomp = "power", cascade_height = 4 *) reg  [31:0] mem [(16*1024)-1:0];
reg  [13:0] addr_reg;
reg  [31:0] dout_tmp0;
reg  [31:0] dout_tmp1;
reg  [31:0] din_reg;
reg         we_reg;

always @(posedge clk)
begin
    addr_reg <= addr;
    din_reg  <= din;
    we_reg   <= we;
    dout_tmp0 <= mem[addr_reg];
    dout_tmp1 <= dout_tmp0;
    dout <= dout_tmp1;
    if (we_reg)
        mem[addr_reg] <= din_reg;
end

endmodule
```

*Figure 3-27:* **RTL Code for 32x16K Memory Configuration Using the CASCADE_HEIGHT and RAM_DECOMP Attributes**

If you apply only the `ram_decomp = "power"` attribute, 16 RAMB36E2 is inferred and the memory is decomposed as follows:

- The base primitive is 32x1K.
- 8 block RAMs are cascaded to create a 32x8K configuration.
- 2 parallel structures create a 16K deep memory.
- The outputs are multiplexed into a 2:1 MUX to generate the output data.

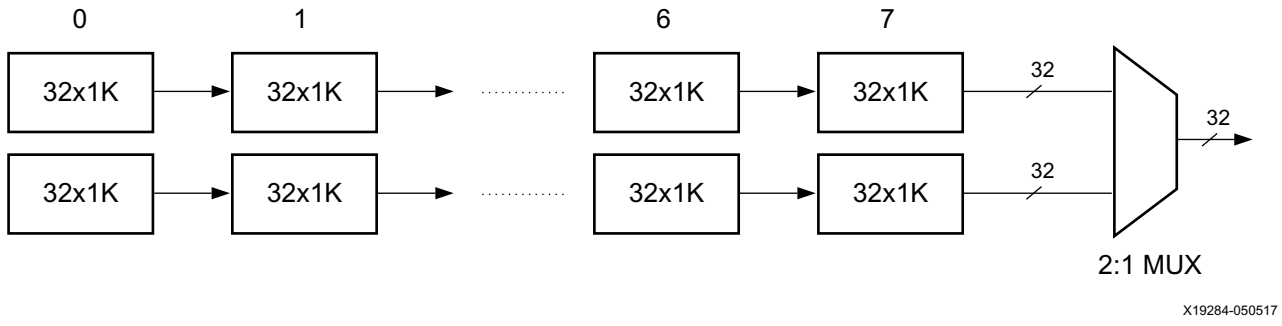



*Figure 3-28:* **Generated Structure for 32x16K Memory Configuration Using the RAM_DECOMP Attribute**

The following RTL code example shows the use of the RAM_DECOMP attribute.

```
module test
(
input  clk,
input  we,
input  [13:0] addr,
input  [31:0] din,
output reg [31:0] dout
);

(* ram_style = "block", ram_decomp = "power"*) reg  [31:0] mem [(16*1024)-1:0];
reg  [13:0] addr_reg;
reg  [31:0] dout_tmp0;
reg  [31:0] dout_tmp1;
reg  [31:0] din_reg;
reg         we_reg;

always @(posedge clk)
begin
     addr_reg <= addr;
     din_reg  <= din;
     we_reg   <= we;
     dout_tmp0 <= mem[addr_reg];
     dout_tmp1 <= dout_tmp0;
     dout <= dout_tmp1;
     if (we_reg)
         mem[addr_reg] <= din_reg;
end

endmodule
```

*Figure 3-29:* **RTL Code for 32x16K Memory Configuration Using the RAM_DECOMP Attribute**

If you use only the RAM_DECOMP attribute, the overall power savings is similar to using both the RAM_DECOMP and CASCADE_HEIGHT attributes together, because only one block RAM is active at a time. Creating a 4-deep cascaded block RAM chain is better for performance when compared to an 8-deep cascaded block RAM chain.

For more information, see this link in the *Vivado Design Suite User Guide: Synthesis* (UG901)[Ref 16].

## Running RTL DRCs

A set of RTL DRC rules identify potential coding issues with your HDL. You can perform these checks on the elaborated views, which you can open by clicking **Open Elaborated Design** in the Flow Navigator. You can run these DRC checks by selecting **RTL Analysis > Report Methodology** in the Flow Navigator or by executing `report_methodology` at the Tcl command prompt.

# Clocking Guidelines

Each FPGA architecture has some dedicated resources for clocking. Understanding the clocking resources for your FPGA architecture can allow you to plan your clocking to best utilize those resources. Most designs might not need you to be aware of these details. However, if you can control the placement and have a good idea of the fanout on each of the clocking domains, you can explore alternatives based on the following clocking details. If you decide to exploit any of these clocking resources, you need to explicitly instantiate the corresponding clocking element.

## UltraScale Device Clocking

UltraScale devices have a different clocking structure from previous device architectures, which blurs the line between global versus regional clocking. UltraScale devices do not have regional clock buffers like 7 series devices and instead use a common buffer and clock routing structure whether the loads are local/regional or global.

UltraScale devices feature smaller clock regions of a fixed size across devices, and the clock regions no longer span half of the device width in the horizontal direction. The number of clock regions per row varies per UltraScale device. Each clock region contains a clock network routing that is divided into 24 vertical and horizontal routing tracks and 24 vertical and horizontal distribution tracks. The following figure shows a device with 36 clock regions (6 columns x 6 rows). The equivalent 7 Series device has 12 clock regions (2 columns x 6 rows).

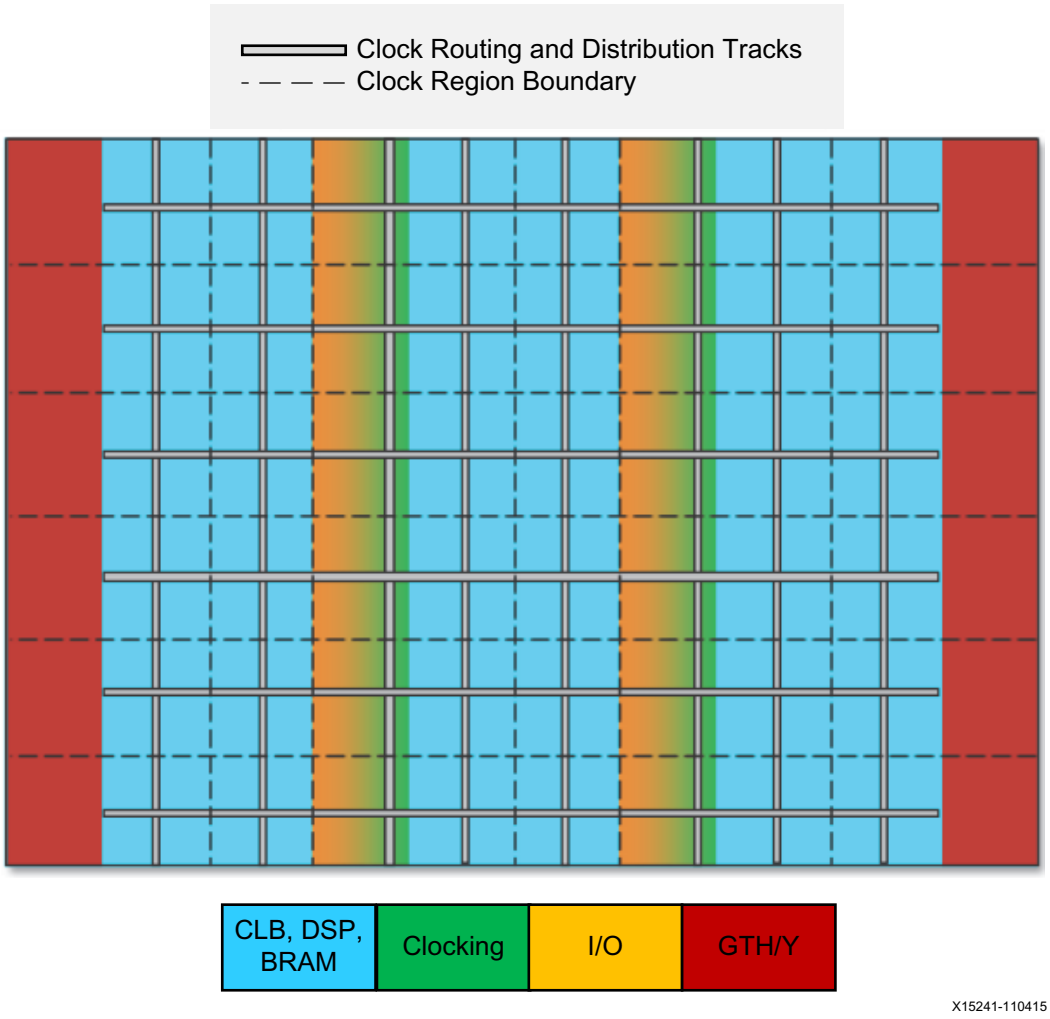



*Figure 3-30:* **UltraScale Device Clock Region Tiles**

The clocking architecture is designed so that only the clock resources necessary to connect clock buffers and loads for a given placement are used, and no resource is wasted in clock regions with no loads. The efficient clock resource utilization enables support for more design clocks in the architecture while improving clock characteristics for performance and power. Following are the main categories of clock types and associated clock structures grouped by their driver and use:

- High-Speed I/O Clocks

  These clocks are associated with the high-speed SelectIO™ interface bit slice logic, generated by the PLL, and routed via dedicated, low-jitter resources to the bit slice logic for high-speed I/O interfaces. In general, this clocking structure is created and controlled by Xilinx IP, such as memory IP or the High Speed SelectIO Wizard, and is not user specified.

- General Clocks

  These clocks are used in most clock tree structures and can be sourced by a GCIO package pin, an MMCM/PLL, or fabric logic cells (not generally suggested). The general clocking network must be driven by BUFGCE/BUFGCE_DIV/BUFGCTRL buffers, which are available in any clock region that contains an I/O column. Any given clock region can support up to 24 unique clocks, and most UltraScale devices can support over 100 clock trees depending on their topology, fanout, and load placement.

- Gigabit Transceiver (GT) Clocks

  Transmit, receive, and reference clocks of gigabit transceivers (GTH or GTY) use dedicated clocking in the clock regions that include the GTs. You can use GT clocks to achieve the following:

  - Drive the general clocking network using the BUFG_GT buffers to connect any loads in the fabric
  - Share clocks across several transceivers in the same or different Quad

### *Clock Primitives*

Most clocks enter the device through a global clock-capable I/O (GCIO) pin. These clocks directly drive the clock network via a clock buffer or are transformed by a PLL or MMCM located in the clock management tile (CMT) adjacent to the I/O column.

The CMT contains the following clocking resources:

- Clock generation blocks
  - 2 PLLs
  - 1 MMCM
- Global clock buffers
  - 24 BUFGCEs
  - 8 BUFGCTRLs
  - 4 BUFGCE_DIVs

***Note:*** Clocking resources in CMTs that are adjacent to I/O columns with unbonded I/Os are available for use.

The GT user clocks drive the global clock network via BUFG_GT buffers. There are 24 BUFG_GT buffers per clock region adjacent to the GTH/GTY columns.

Following is summary information for each of the UltraScale device clock buffers:

- BUFGCE

  The most commonly used buffer is the BUFGCE. This is a general clock buffer with a clock enable/disable feature equivalent to the 7 series BUFHCE.

- BUFGCE_DIV

  The BUFGCE_DIV is useful when a simple division of the clock is required. It is considered easier to use and more power efficient than using an MMCM or PLL for simple clock division. When used properly, it can also show less skew between clock domains as compared to an MMCM or PLL when crossing clock domains. The BUFGCE_DIV is often used as replacement for the BUFR function in 7 series devices. However, because the BUFGCE_DIV can drive the global clock network, it is considered more capable than the BUFR component.

- BUFGCTRL (also BUFGMUX)

  The BUFGCTRL can be instantiated as a BUFGMUX and is generally used when multiplexing two or more clock sources to a single clock network. As with the BUFGCE and BUFGCE_DIV, it can drive the clock network for either regional or global clocking.

- BUFG_GT

  When using clocks generated by GTs, the BUFG_GT clock buffer allows connectivity to the global clock network. In most cases, the BUFG_GT is used as a regional buffer with its loads placed in one or two adjacent clock regions. The BUFG_GT has built-in dynamic clock division capability that you can use in place of an MMCM for clock rate changes.

You can use the Clock Utilization Report in the Vivado IDE to visually analyze clocking resource utilization and clock routing. The following figure shows the clock resource utilization per clock region overlaid in the Device window. For more information on this report, see the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21].

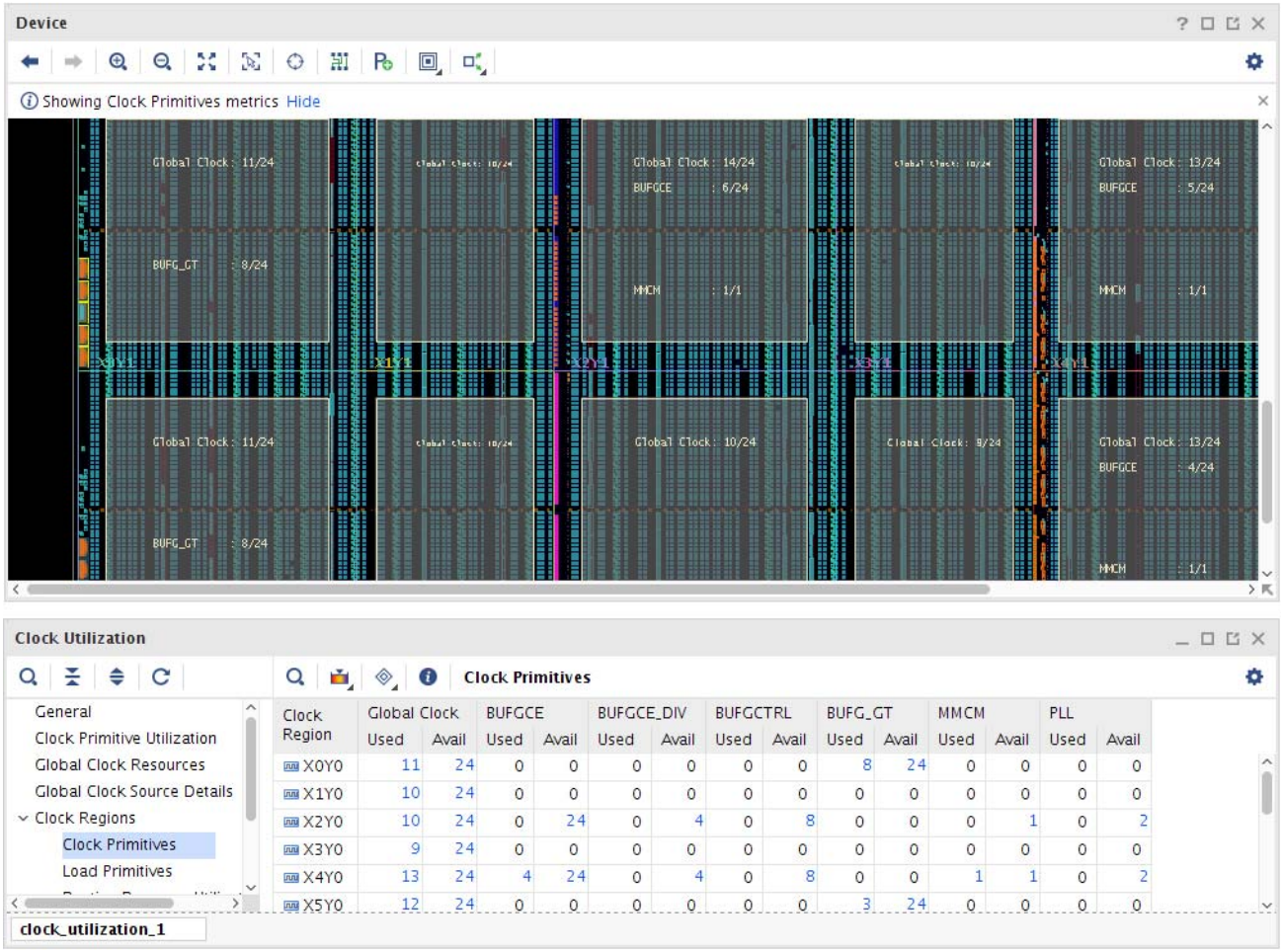



*Figure 3-31:* **Clock Utilization Report**

For more information on the BUFGCE, BUFGCE_DIV, and BUFGCTRL buffers, see the *UltraScale Architecture Clocking Resources User Guide* (UG572) [Ref 40]. For details on connectivity and use of the BUFG_GT buffer, see the appropriate *UltraScale Architecture Transceiver User Guide* [Ref 41].

### *Global Clock Buffer Connectivity and Routing Tracks*

Each of the 24 BUFGCE buffers in a clock region can only drive a specific clock routing track. However, the BUFGCTRL and BUFGCE_DIV outputs can use any of the 24 tracks by going through a MUX structure. Each BUFGCE_DIV shares the input connectivity with a specific BUFGCE site, and each BUFGCTRL shares input connectivity with two specific BUFGCE sites.

Consequently, when BUFGCE_DIV or BUFGCTRL buffers are used in the clock region, use of the BUFGCE buffers is limited. The following figure shows the bottom 6 BUFGCE in a clock region, which are replicated 4 times within a clock region.

***Note:*** A global clock net is assigned to a specific track ID in the device for all the vertical, horizontal routing, and distribution resources the clock uses. A clock *cannot* change track IDs unless the clock goes through another clock buffer.

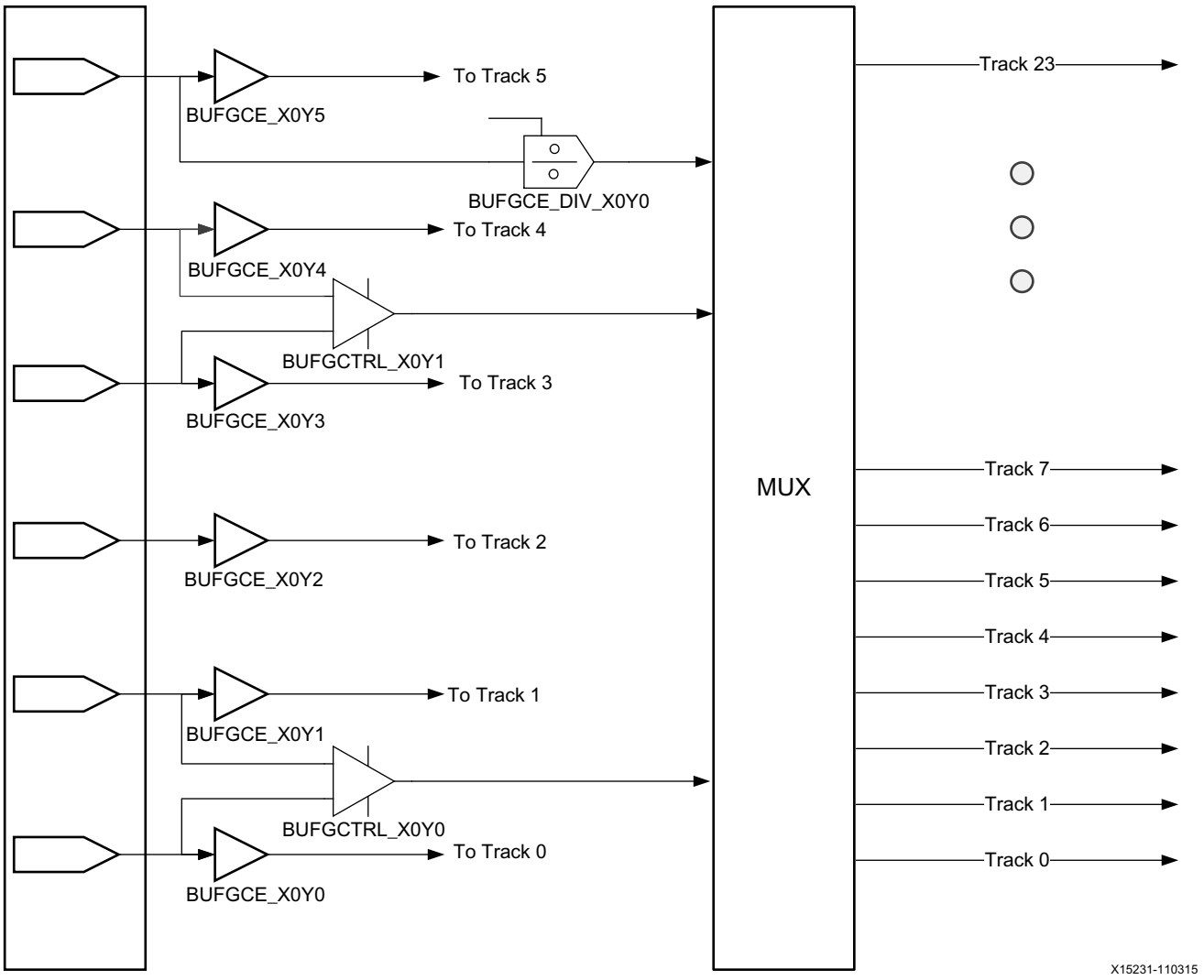



*Figure 3-32:* **BUFGCE, BUFGCE_DIV, and BUFGCTRL Shared Inputs and Output Muxing**

### *Clock Routing, Root, and Distribution*

To properly understand the clocking capacity of an UltraScale device and the clocking utilization of a design, it is important to know how the clock routes use the dedicated routing resources:

- From the clock buffer to the clock root, the clock signal goes through one or several segments of vertical and horizontal routing. Each segment must use the same track ID (between 0 and 23).
- At the clock root, the clock signal transitions from the routing track to the distribution track with the same track ID. To reduce skew, the clock root is usually in the clock region located in the center of the clock window. The clock window is the rectangular area that includes all the clock regions where the clock net loads are placed. For skew optimization reasons, the Vivado IDE might move the clock root to off center.
- From the clock root to the CLB columns where the loads are located, the clock signal travels on the vertical distribution (both up and down the device as needed) and then onto the horizontal distribution (both to the left and right as needed).
- The CLB columns are split into two halves, which are located above and below the horizontal distribution resources. Each half of the CLB column contains several leaf clock routing resources that can be reached by any of the horizontal distribution tracks.

In some cases, a clock buffer can directly drive onto the clock distribution track. This usually happens when the clock root is located in the same clock region as the clock buffer or when the clock buffer only drives non-clock pins (for example, high fanout nets).

Because clock routing resources are segmented, only the routing and distribution segments used to traverse a clock region or to reach a load in a clock region are consumed.

The following figure shows how a clock buffer located in clock region X2Y1 reaches its loads placed inside the clock window, which is formed by a rectangle of clock regions from X1Y3 to X5Y5.

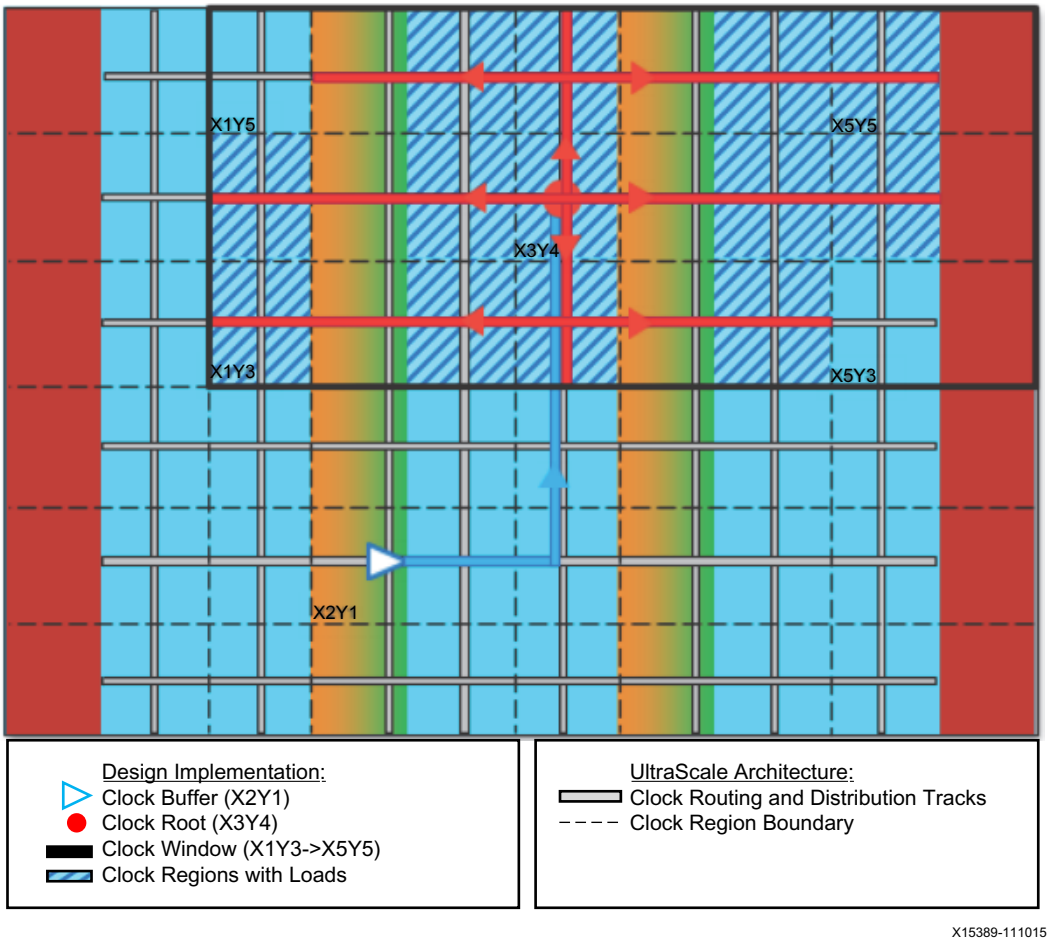



*Figure 3-33:* **UltraScale Device Clock Routing from Driver to Loads**

In the following figure, a routed device view shows an example of a global clock that spans most of the device. The clock buffer driving the network is highlighted in blue in clock region X2Y0 and drives onto the horizontal routing in that clock region. The net then transitions from the horizontal routing onto the vertical routing in clock region X2Y0 reaching the clock root in clock region X2Y5. All clock routing is highlighted in blue. The clock root is highlighted in red in the clock region X2Y5. From the clock root in X2Y5, the net transitions onto the vertical distribution and then the horizontal distribution to the clock leaf pins. The distribution layer and the leaf clock routing resources in the CLB columns are highlighted in red.

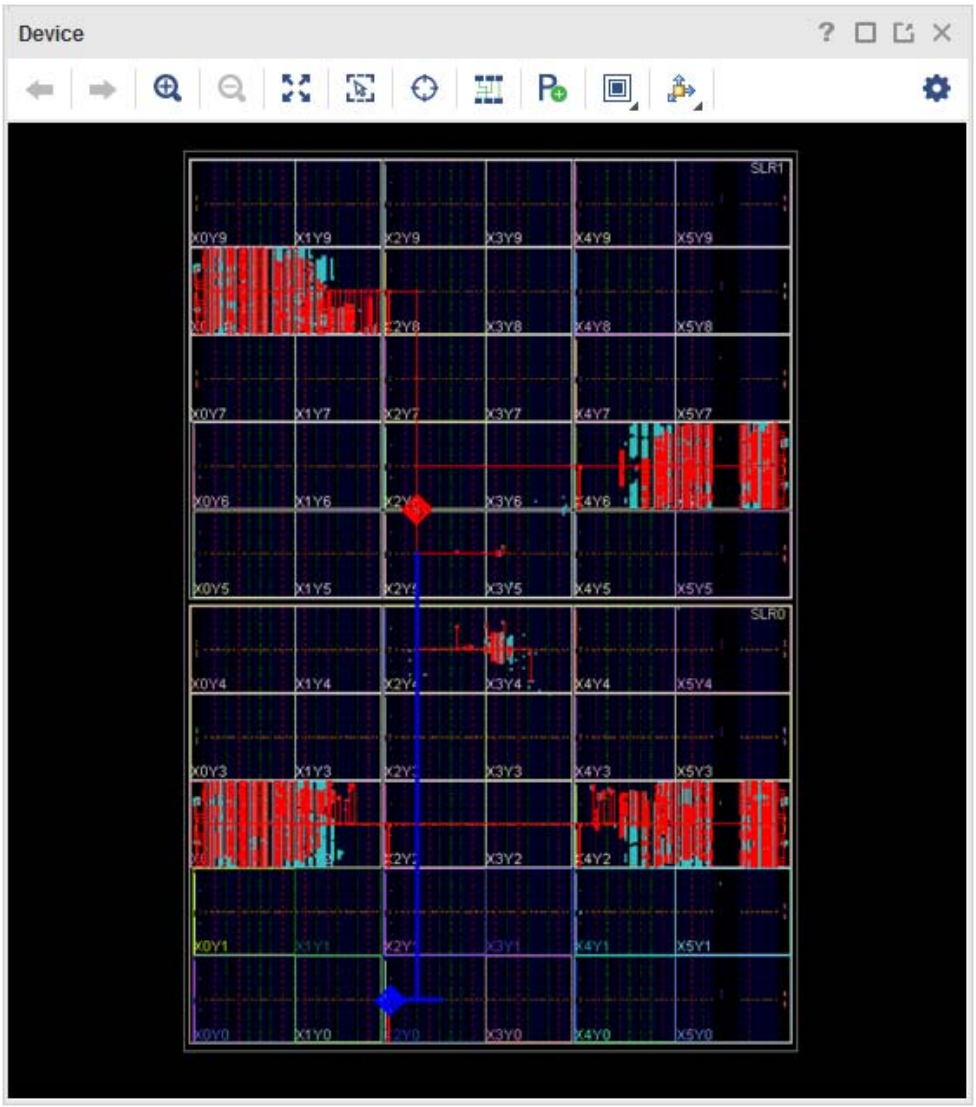



*Figure 3-34:* **Routed Device View of a Routed Clock Network**

### *Clock Tree Placement and Routing*

During the following phases, the Vivado placer determines the placement of MMCM/PLLs, global clock buffers, and the clock root while honoring the physical XDC constraints:

1. I/O and clock placement

   The placer places I/O buffers and MMCM/PLLs based on connectivity rules and user constraints. The placer assigns clock buffers to clock regions but not to individual sites unless constrained using the LOC property. For details, see Table 3-2. Only the clock buffers that only drive non-clock loads can move to a different clock region later in the flow based on the placement of their driver and loads.

   Any placer error at this phase is due to conflicting connectivity rules, user constraints, or both. The log file shows extensive information about the possible root cause of the error, which you must review in detail to make the appropriate design or constraint change.

2. SLR partitioning (SSI technology devices only) and global placement

   The placer performs the initial clock tree implementation based on early driver and load placements. Each clock net is associated with a clock window. The excessive overlap of clock windows can lead to placer errors due to anticipated clock routing contention.

   When a clock partitioning error occurs, the log file shows the last clock budgeting solution for each clock net as well as the number of unique clock nets present in each clock region. Review the log file in detail to determine which clocks to remove from the overutilized clock regions. You can remove clocks using the following methods:

   - Reduce the number of clocks in the design by combining identical synchronous clocks, removing unnecessary MMCM feedback clocks, or consolidating lower fanout clocks with high fanout clocks.
   - Move clock primitives to different clock regions, especially those without connectivity-based placement rules.
   - Add floorplanning constraints on clock loads to keep clocks with smaller fanout closer to their driver or away from the highly utilized clock regions.

   The placer refines the clock tree implementation several times to help improve timing QoR. For example, during the later placement optimization phases, the placer analyzes each challenging clock to determine a better clock root location.

3. Clock tree pre-routing

   The placer guides the subsequent implementation steps and provides accurate delay estimates for post-place timing analysis.

After placement, the Vivado tools can modify the clock tree implementation as follows:

- The Vivado physical optimizer can replicate and move cells to clock regions without associated clocks.
- The Vivado router can make adjustments to improve timing QoR and legalize the clock routing. The Vivado router can also modify the clock root location to improve timing QoR when you use the `Explore` routing directive.

The following table summarizes the placement rules for the main clock topologies and how constraints affect these rules.

*Table 3-2:* **Topologies with and without Placement Rules**

| Constrained Source | Unconstrained Destination | Behavior |
|---|---|---|
| GCIO | BUFGCE, BUFGCTRL, BUFGCE_DIV, PLL/MMCM | Automatically placed in same clock region. |
| PLL/MMCM | BUFGCE, BUFGCTRL, BUFGCE_DIV | Automatically placed in same clock region. |
| GT*_CHANNEL | BUFG_GT | Automatically placed in same clock region. |
| BUFGCTRL | BUFGCTRL | Automatically placed in same clock region.<br>***Note:*** You can override placement within same clock region using the CLOCK_REGION constraint. |
| BUFG* | BUFG* | Unpredictable placement of unconstrained destination BUFG.<br>Recommend constraining destination BUFG* using the CLOCK_REGION constraint.<br>***Note:*** This excludes BUFGCTRL -> BUFGCTRL. |
| BUFG* | MMCM/PLL | Unpredictable placement of unconstrained destination MMCM/PLL.<br>Recommend constraining MMCM/PLL using a LOC constraint.<br>Recommend CLOCK_DEDICATED_ROUTE constraint when the route spans adjacent or multiple clock regions. |

## *Clocking Capability*

Clock planning must be based on the total number of high fanout clocks and low fanout clocks in the target device.

### High Fanout Clocks

A high fanout clock spans almost an entire SLR of an SSI technology device or almost all clock regions of a monolithic device. The following figure shows a high fanout clock that spans almost an entire SLR with the BUFGCE driver shown in red.

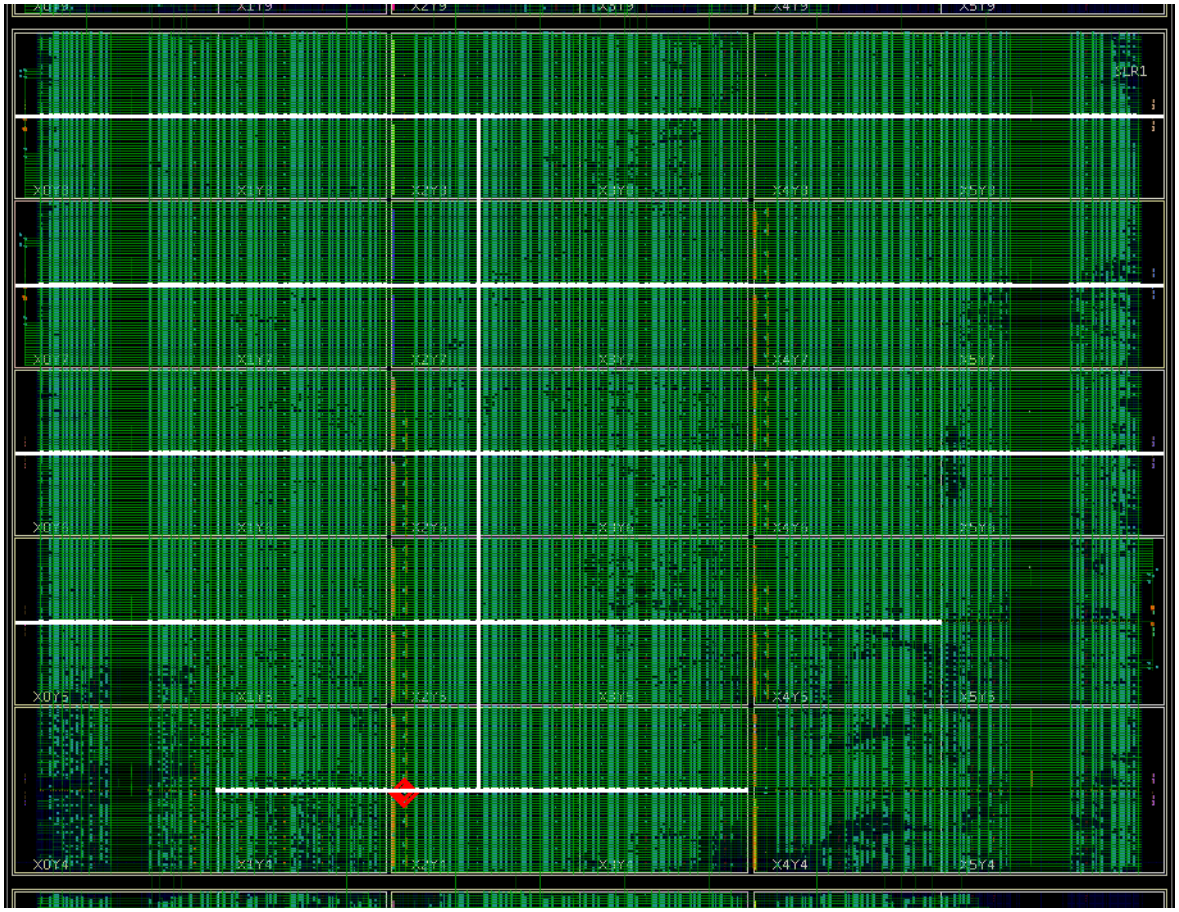



*Figure 3-35:* **High Fanout Clock Spanning an SLR**

***Note:*** Using more than 24 clocks in a design might cause issues that require special design considerations or other up-front planning.

**IMPORTANT:** *In ZHOLD and BUF_IN compensation modes, the MMCM feedback clock path matches the CLKOUT0 clock path in terms of routing track, clock root location, and distribution tracks. Therefore, the feedback clock can be considered a high fanout clock when the clock buffer and clock root are far apart. For more information on MMCM compensation mechanism, see I/O Timing with MMCM ZHOLD/BUF_IN Compensation.*

### Low Fanout Clocks

In most cases, a low fanout clock is a clock net that is connected to less than 5,000 clock pins, which are placed in 3 or fewer horizontally adjacent clock regions. The clock routing, clock root, and clock distribution are all contained within the localized area.

In some cases, the placer is expected to identify a low fanout clock but fails. This can be caused by design size, device size, or physical XDC constraints, such as a LOC constraint or Pblock, which prevent the placer from placing the loads in a local area. To address this issue, you might need to guide the tool by manually creating a Pblock or modifying the existing physical constraints.

Clocks driven by BUFG_GTs are an example of a low fanout clock. The Vivado placer automatically identifies these clock nets and contains the loads to the clock regions adjacent to the GT interface. The following figure shows a low fanout clock contained in two clock regions with the BUFG_GT driver shown in red.

**TIP:** *To contain a low fanout clock to a single clock region, you can use the CLOCK_LOW_FANOUT XDC constraint. For more information, see Using the CLOCK_LOW_FANOUT Constraint.*

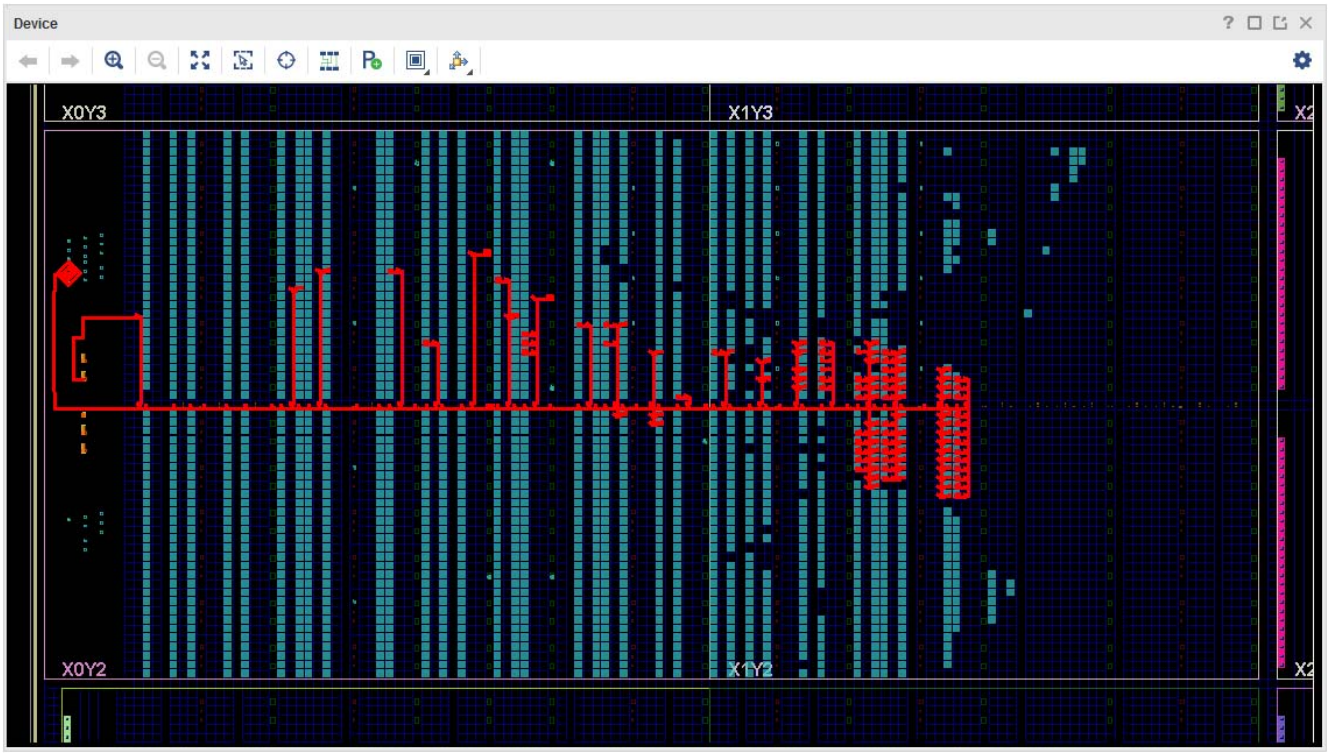



*Figure 3-36:* **Low Fanout Clock Contained in Two Clock Regions**

**Balanced Utilization of High and Low Fanout Clocks**

UltraScale devices support more clocks than previous Xilinx FPGA families. This enables a wide range of clocking utilization scenarios, such as the following:

- 24 clocks or less

  Unless conflicting user constraints exist, all clocks can be treated as high fanout clocks without risking placement or routing contention.

- Almost 300 clocks

  For a design that targets a device with 6 clock region rows and includes only low fanout clocks with each clock included in 3 clock regions at most, the following clocks are required: 6 rows x 2 clock windows per row x 24 clocks per region = 288 clocks.

Low fanout clock windows do not have a fixed size but are usually between 1 and 3 clock regions. High fanout clocks rarely span the entire device or an entire SLR.

The following method shows how to balance high fanout clocks and low fanout clocks, assuming that a few low fanout clocks come from I/O interfaces and most from GT interfaces. You can apply the same method for each SSI technology device SLR.

- High fanout clocks
  - Up to 12 for monolithic devices
  - Up to 24 for SSI technology devices (assuming some high fanout clocks are only present in 1 SLR)
- Low fanout clocks
  - Up to 12 plus 8 per GT utilized Quad
  - Alternatively, up to 12 plus 6 per GT interface (group of GT channels that share the RXUSRCLK and TXUSRCLK)

## *Clock Constraints*

Physical XDC constraints drive the implementation of clock trees and control the use of high fanout clocking resources. Because UltraScale device clocking is more flexible than clocking with previous architectures and includes additional architectural constraints, it is important to understand how to properly constrain your clocks for implementation.

### Using LOC Constraints for IO/MMCM/PLL/GT

To constrain clocks, you can assign placement constraints as follows:

- On a clock input at the I/O port

  Assigning a PACKAGE_PIN constraint for a clock on a GCIO or assigning a LOC to an IOB affects the clock network. The MMCM/PLL and clock buffers directly connected to the input port must be placed in the same clock region.

- On an MMCM or PLL

  The clock buffers directly connected to the MMCM or PLL outputs and the input clock ports connected to the MMCM or PLL inputs are automatically placed in the same clock region. If an input clock port and an MMCM or PLL are directly connected and constrained to different clock regions, you must manually insert a clock buffer and set a CLOCK_DEDICATED_ROUTE constraint on the net connected to the MMCM or PLL.

- On a GT*_CHANNEL or IBUFDS_GTE3 cell

  The BUFG_GTs driven by the cell are placed in the same clock region.

**CAUTION!** *Xilinx does not recommended using LOC constraints on the clock buffer cells. This method forces the clock onto a specific track ID, which can result in placement that cannot be legally routed. Only use LOC constraints to place high fanout clock buffers in UltraScale devices when you understand the entire clock tree of the design and when placement is consistent in the design. Even after taking these precautions, collisions might occur during implementation due to design or constraint changes.*

### Using the CLOCK_REGION Property on Clock Buffers

You can use the CLOCK_REGION constraint to assign a clock buffer to a clock region without specifying a site. This gives the placer more flexibility when optimizing all the clock trees and when determining the appropriate buffer sites to successfully route all clocks.

You can also use a CLOCK_REGION constraint to provide guidance on the placement of cascaded clock buffers or clock buffers driven by non-clocking primitives, such as fabric logic.

In the following example, the XDC constraint assigns the `clkgen/clkout2_buf` clock buffer to the `CLOCK_REGION X2Y2`.

```
set_property CLOCK_REGION X2Y2 [get_cells clkgen/clkout2_buf]
```

***Note:*** In most cases, the clock buffers are directly driven by input clock ports, MMCMs, PLLs, or GT*_CHANNELs that are already constrained to a clock region. If this is the case, the clock buffers are automatically placed in the same clock region, and you do not need to use the CLOCK_REGION constraint.

### Using a Pblock to Restrict Clock Buffer Placement

When a clock buffer does not need to be placed in a specific clock region, you can use a Pblock to specify a range of clock regions. For example, use a Pblock when a BUFGCTRL is needed to multiplex two clocks that are located in different areas. You can assign the BUFGTRL to a Pblock that includes the clock regions between the two clock drivers and let the placer identify a valid placement.

***Note:*** Xilinx does *not* recommend using a Pblock for a single clock region.

### Using the USER_CLOCK_ROOT Property on a Clock Net

You can use the USER_CLOCK_ROOT property to force the clock root location of a clock driven by a clock buffer. Specifying the USER_CLOCK_ROOT property influences the design placement, because it impacts both insertion delay and skew by modifying the clock routing. The USER_CLOCK_ROOT value corresponds to a clock region, and you must set the property on the net segment directly driven by the high fanout clock buffer. Following is an example:

```
set_property USER_CLOCK_ROOT X2Y3 [get_nets clkgen/wbClk_o]
```

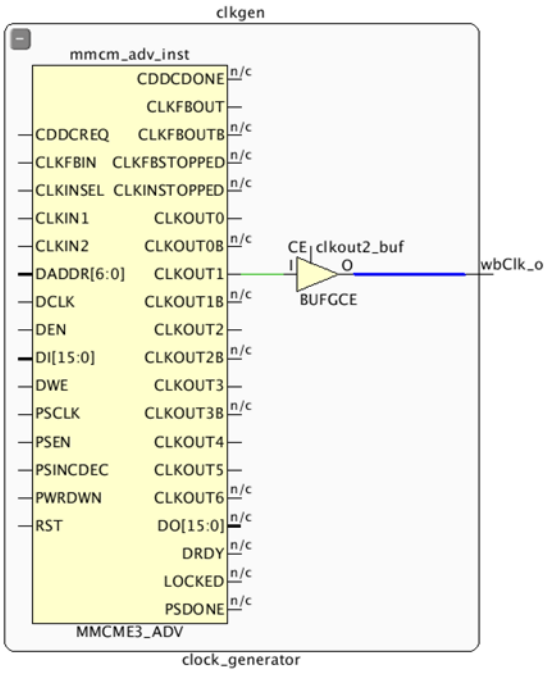



*Figure 3-37:* **USER_CLOCK_ROOT Applied on the Net Segment Driven by the Clock Buffer**

After placement, you can use the CLOCK_ROOT property to query the actual clock root as shown in the following example. The CLOCK_ROOT reports the assigned root whether it was user assigned or automatically assigned by the Vivado tools.

```
get_property CLOCK_ROOT [get_nets clkgen/wbClk_o]
=> X2Y3
```

Another way to review the clock root assignments of your implemented design is to use the `report_clock_utilization` Tcl command. For example:

```
report_clock_utilization [-clock_roots_only]
```

The following figure shows this report.

| Index | Clock Net | Root Clock Region |
|---|---|---|
| 1 | clkgen/clkfbout_buf | X4Y1 |
| 2 | clkgen/cpuClk_o | X4Y1 |
| 3 | clkgen/fftClk_o | X3Y2 |
| 4 | clkgen/phyClk0_o | X3Y3 |
| 5 | clkgen/phyClk1_o | X3Y2 |
| 6 | clkgen/usbClk_o | X3Y3 |

*Figure 3-38:* **report_clock_utilization Clock Root Assignments**

### Using the CLOCK_DELAY_GROUP Constraint on Several Clock Nets

You can use the CLOCK_DELAY_GROUP constraint to match the insertion delay of multiple, related clock networks driven by different clock buffers. This constraint is commonly used to minimize skew on synchronous CDC timing paths between clocks originating from the same MMCM or PLL source. You must set the CLOCK_DELAY_GROUP constraint on the net segment directly connected to the clock buffer. The following example shows the `clk1_net` and `clk2_net` clock nets, which are directly driven by the clock buffers:

```
set_property CLOCK_DELAY_GROUP grp12 [get_nets {clk1_net clk2_net}]
```

For more information on using this constraint on paths between clocks, see Synchronous CDC.

### Using the CLOCK_DEDICATED_ROUTE Constraint

The CLOCK_DEDICATED_ROUTE constraint is typically used when driving from a clock buffer in one clock region to an MMCM or PLL in another clock region. By default, the CLOCK_DEDICATED_ROUTE constraint is set to TRUE, and the buffer/MMCM or PLL pair must be placed in the same clock region.

The following table summarizes the different CLOCK_DEDICATED_ROUTE constraint values, use, and behavior.

*Table 3-3:* **UltraScale Device CLOCK_DEDICATED_ROUTE Constraint Summary**

| Value | Use | Behavior |
|---|---|---|
| TRUE | Default value on clock nets | • Global clock buffer and MMCM/PLLs must be placed in the same clock region.<br>• This value ensures the net is routed using only global clock resources. |
| SAME_CMT_COLUMN | Net driven by a global clock buffer or the output of an IBUF<br><br>Examples:<br>`set_property CLOCK_DEDICATED_ROUTE [get_nets -of [get_pins BUFGCE_inst/O]]`<br>`set_property CLOCK_DEDICATED_ROUTE [get_nets -of [get_pins IBUF_inst/O]]` | • MMCM/PLLs must be placed in a clock region in the same vertical column.<br>• This value ensures the net is routed using only global clock resources.<br>• For optimal results, Xilinx recommends using a LOC constraint on the MMCM/PLL to control placement of the MMCM/PLL within in the same vertical column. |
| ANY_CMT_COLUMN | Net driven by a global clock buffer<br><br>Examples:<br>`set_property CLOCK_DEDICATED_ROUTE [get_nets -of [get_pins BUFGCE_inst/O]]`<br>`set_property CLOCK_DEDICATED_ROUTE [get_nets -of [get_pins BUFGCE_DIV_inst/O]]`<br>`set_property CLOCK_DEDICATED_ROUTE [get_nets -of [get_pins BUFGCTRL_inst/O]]` | • MMCM/PLLs can be placed in any clock region with available resources.<br>• This value ensures the net is routed using only global clock resources.<br>• For optimal results, Xilinx recommends using a LOC constraint on the MMCM/PLL to control placement of the MMCM/PLL within the device. |
| FALSE | Clock net not driven by a global clock buffer but part of the clock network (for example, nets driven by the output of an IBUF or nets directly connected to output clock pins of an MMCM)<br><br>Examples:<br>`set_property CLOCK_DEDICATED_ROUTE [get_nets -of [get_pins MMCME4_ADV_inst/CLKOUT0]]`<br>`set_property CLOCK_DEDICATED_ROUTE [get_nets -of [get_pins IBUF_inst/O]]` | • Net is routed using fabric and global clock resources.<br>• This can adversely affect the timing and performance of the clock network.<br>**IMPORTANT**: For UltraScale devices, CLOCK_DEDICATED_ROUTE=FALSE must only be used when a clock normally routed with global clock resources needs to be routed with fabric resources for special design reasons. |

***Note:*** When working with UltraScale devices, do *not* apply the CLOCK_DEDICATED_ROUTE property to the net driven directly by a port. Instead, apply the CLOCK_DEDICATED_ROUTE property to the output of the IBUF.

When driving from a clock buffer in one clock region to a MMCM or PLL in a vertically adjacent clock region, you must set the CLOCK_DEDICATED_ROUTE to BACKBONE for 7 series devices or to SAME_CMT_COLUMN for UltraScale devices. This prevents

implementation errors and ensures that the clock is routed with global clock resources only. The following example and figure show a clock buffer driving two PLLs in vertically adjacent clock regions.

```
set_property CLOCK_DEDICATED_ROUTE SAME_CMT_COLUMN [get_nets -of [get_pins BUFG_inst_0/O]]
set_property LOC PLLE3_ADV_X0Y0 [get_cells PLLE3_ADV_inst_0]
set_property LOC PLLE3_ADV_X0Y4 [get_cells PLLE3_ADV_inst_1]
```

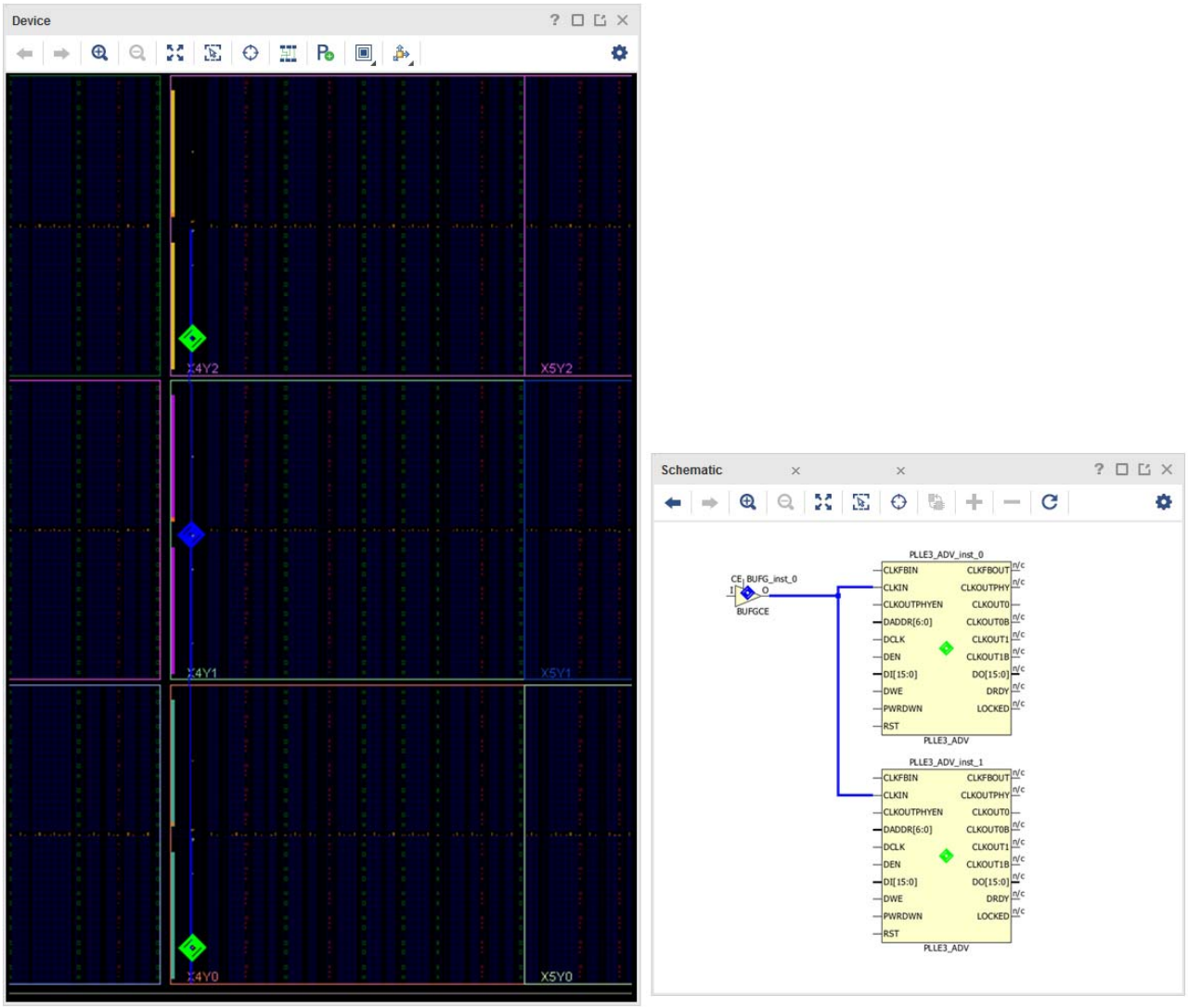



*Figure 3-39:* **CLOCK_DEDICATED_ROUTE Constraint Set to SAME_CMT_COLUMN**

When driving from a clock buffer to other clock regions that are not vertically adjacent, you must set the CLOCK_DEDICATED_ROUTE to FALSE for 7 series devices or to ANY_CMT_COLUMN for UltraScale devices. This prevents implementation errors and ensures that the clock is routed with global clock resources only. The following example and figure show a BUFGCE driving two PLLs that are not located on the same clock region column as the input buffer.

```
set_property CLOCK_DEDICATED_ROUTE ANY_CMT_COLUMN [get_nets -of [get_pins BUFG_inst_0/O]]
set_property LOC PLLE3_ADV_X1Y0 [get_cells PLLE3_ADV_inst_0]
set_property LOC PLLE3_ADV_X1Y4 [get_cells PLLE3_ADV_inst_1]
```

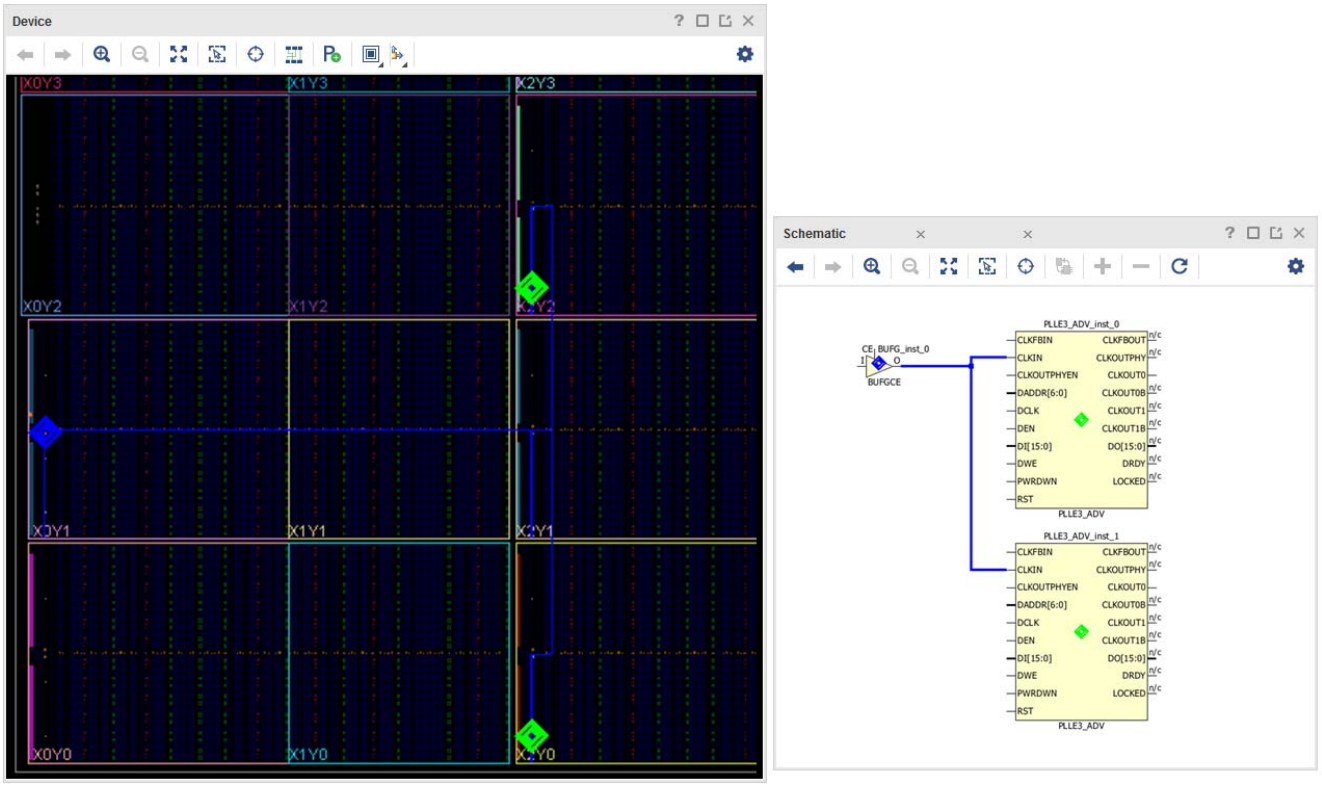



*Figure 3-40:* **CLOCK_DEDICATED_ROUTE Set to ANY_CMT_COLUMN**

### Using the CLOCK_LOW_FANOUT Constraint

You can use the CLOCK_LOW_FANOUT constraint to contain the loads of a clock buffer in a single clock region. CLOCK_LOW_FANOUT is set on the clock net segment directly driven by the global clock buffer, and the fanout of the global clock buffer must be less than 2,000 loads.

***Note:*** CLOCK_LOW_FANOUT takes lower precedence when used in conjunction with other clocking constraints. If CLOCK_LOW_FANOUT is in conflict with other clock constraints, such as USER_CLOCK_ROOT CLOCK_DELAY_GROUP or CLOCK_DEDICATED_ROUTE, CLOCK_LOW_FANOUT is not obeyed.

The following example shows the CLOCK_LOW_FANOUT constraint used to drive a clock network with less than 2,000 loads and contain it in a single clock region. The input clock port, clkIn has a PACKAGE_PIN assignment to a GCIO located in the CLOCK_REGION X2Y0 and drives a PLLE3_ADV. The PLLE3_ADV drives a global clock buffer that subsequently drives the clock network with 1379 loads. The loads of the global clock buffer are all placed in the CLOCK_REGION X2Y0.

```
# PACKAGE_PIN AF9 - IOBank 64 - CLOCK_REGION X2Y0
set_property PACKAGE_PIN AF9 [get_ports clkIn]
set_property IOSTANDARD LVCMOS18 [get_ports clkIn]
set_property CLOCK_LOW_FANOUT TRUE [get_nets -of [get_pins clkOut0_bufg_inst/O]]
```

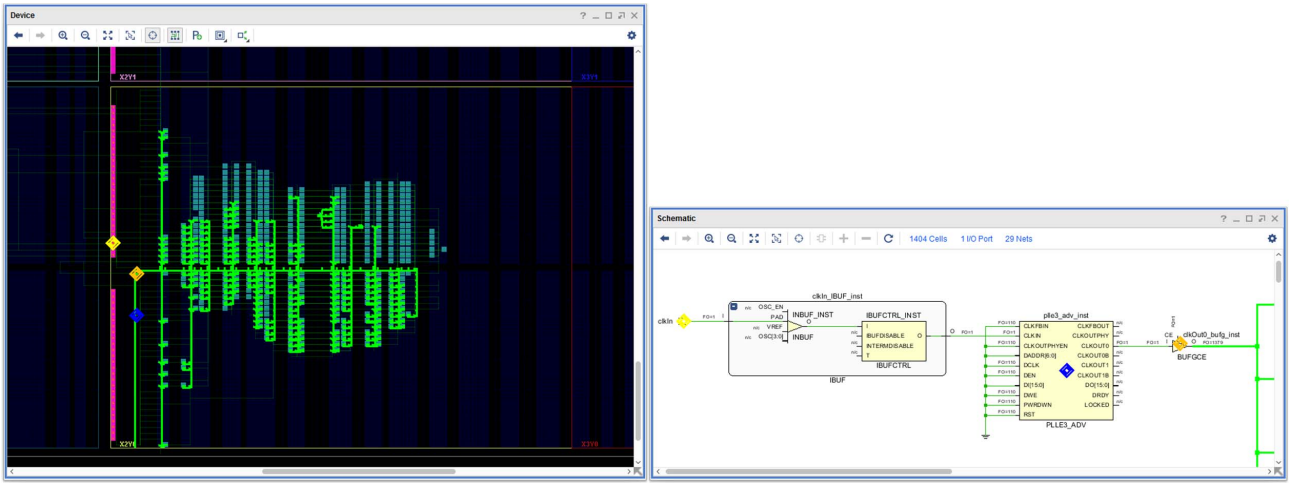



*Figure 3-41:* **CLOCK_LOW_FANOUT Example in Device View and Schematic View**

## *Clocking Topology Recommendations*

Xilinx recommends using simple clock tree topologies with the minimum number of clock buffers required for the design. Using extra clock buffers requires more routing tracks, which can lead to placement errors or routing conflicts in clock regions where the clock routing requirement is high and is close to the maximum capacity.

Following are clocking topology recommendations for BUFGCE/BUFGCTRL/BUFGCE_DIV connectivity.

### Parallel Clock Buffers

Use parallel clock buffers to achieve the following:

- Ensure predictable placement across implementation runs

  When the parallel clock buffers are directly driven by the same input clock port, MMCM, PLL, or GT*_CHANNEL, the buffers are always placed in the same clock region as their driver regardless of the netlist changes or logic placement variation.

- Match the insertion delays between parallel branches of the clock tree

  Xilinx recommends parallel buffers over cascaded clock buffers, especially when there are synchronous paths between the branches. When using cascaded buffers, the clock insertion delay is *not* matched between the branches of the clock trees even when using the CLOCK_DELAY_GROUP or USER_CLOCK_ROOT constraints. This can result in high clock skew, which makes timing closure challenging if not impossible.

The following figure shows three parallel BUFGCE buffers driven by the MMCM CLKOUT0 port.

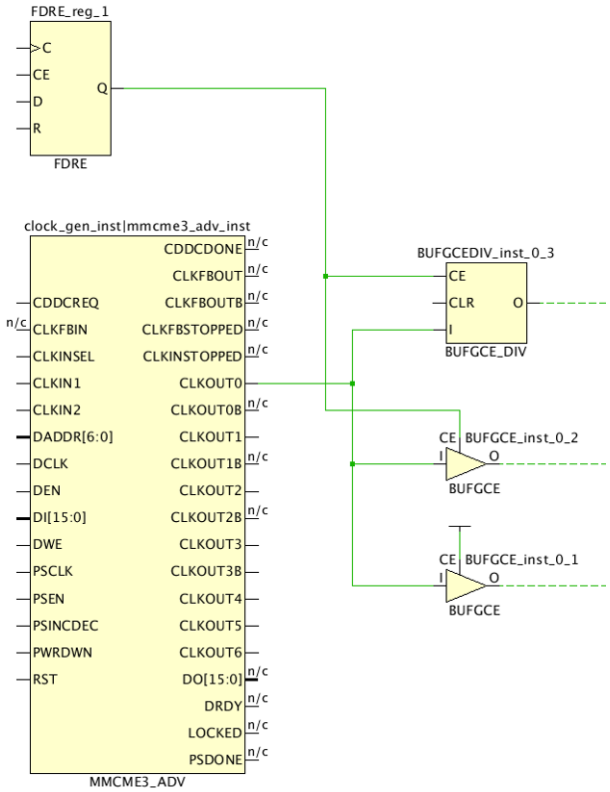



*Figure 3-42:* **Parallel BUFGCE on MMCM Output**

### Cascaded Clock Buffers

In general, Xilinx does not recommend using cascaded buffers to artificially increase the delay and reduce the skew between unrelated clock trees branches. Unlike connections between BUFGCTRLs, other clock buffer connections do not have a dedicated path in the architecture. Therefore, the relative placement of clock buffers is not predictable, and all placement rules take precedence over placing unconstrained cascaded buffers.

However, you can use cascaded clock buffers to achieve the following:

- Route the clock to another clock buffer located in a different clock region

  This method is typical when using a clock multiplexer for clocks generated by MMCMs located in different clock regions. Although one of the MMCMs can directly drive the BUFGCTRL (BUFGMUX), the other MMCM requires an intermediate clock buffer to route the clock signal to the other region. The following figure shows an example.

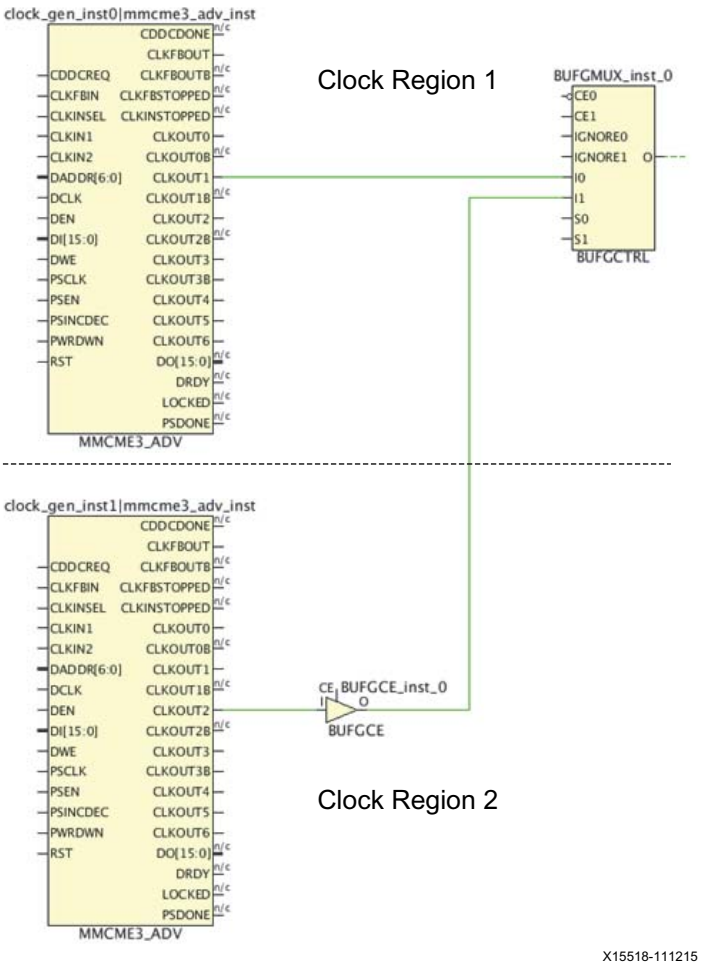



*Figure 3-43:* **Routing the Clock to Another Clock Region**

- Balance the number of clock buffer levels across the clock tree branches when there is a synchronous path between those branches

  For example, consider an MMCM clock called clk0 that drives both group A (sequential cells driven via a BUFGCTRL located in a different clock region) and group B (sequential cells). To better match the delay between the branches, insert a BUFGCE for group B and place it in the same clock region as the BUFGCTRL. This ensures that the synchronous paths between group A and group B have a controlled amount of skew. The following figure shows an example.

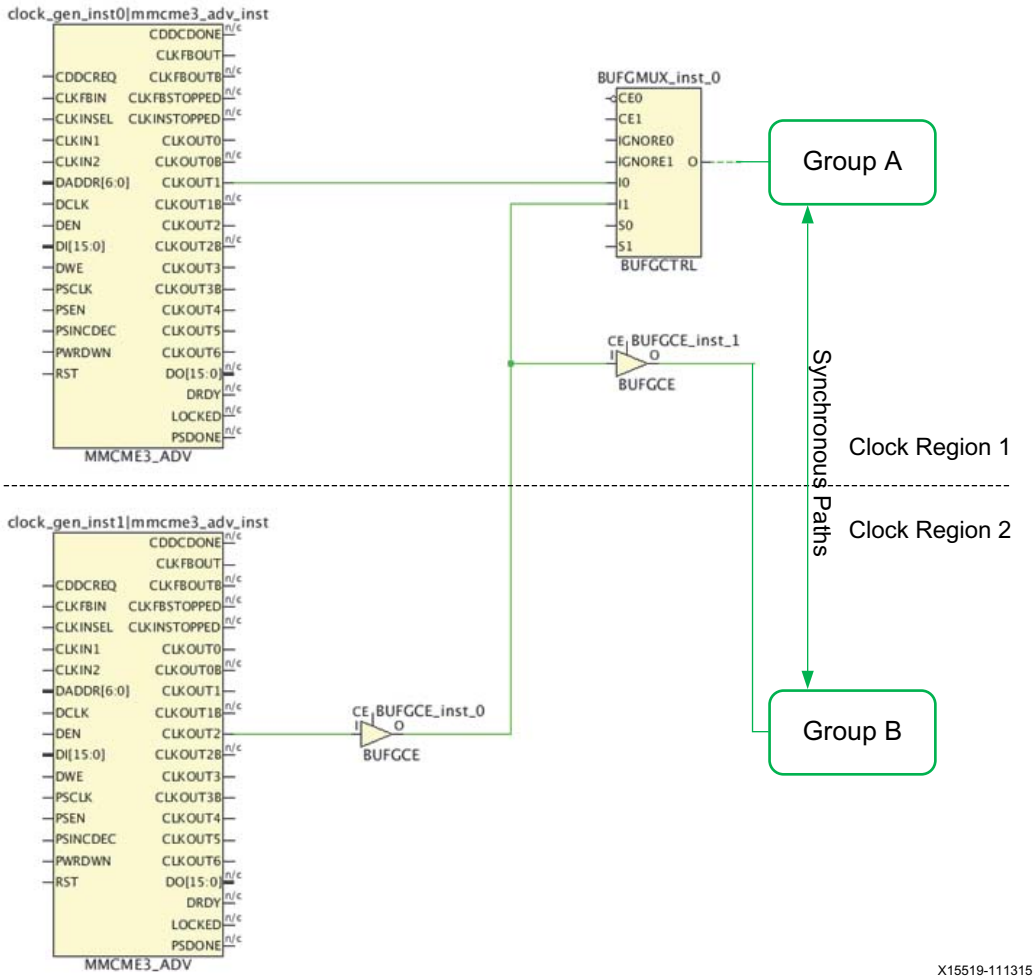



*Figure 3-44:* **Balancing Clock Trees for Synchronous Paths Between Clock Regions**

***Note:*** If there are only asynchronous paths between the clock tree branches, the branches do not need to be balanced as long as there is proper synchronization circuitry on the receiving clock domain.

- Build clock multiplexers as described in Clock Multiplexing.

To reduce the variation of insertion delays and skew, Xilinx recommends the following when using cascaded clock buffers:

- Keep the cascaded buffers in the same or adjacent clock regions.
- When clock tree branches are balanced, assign all the clock buffers of the same level to the same clock region.

***Note:*** If absolutely required, Xilinx recommends using two cascaded BUFGCTRLs instead of cascaded BUFGCEs. Using dedicated routing, you can cascade two adjacent BUFGCTRLs with minimum delay when both BUFGCTRLs are placed inside the same clock region.

### Clock Multiplexing

You can build a clock multiplexer using a combination of parallel and cascaded BUFGCTRLs. The placer finds the optimal placement based on the clock buffer site availability. If possible, the placer places BUFGCTRLs in adjacent sites to take advantage of the dedicated cascade paths. If that is not possible, the placer will attempt to place the BUFGCTRLs from the same level in the adjacent clock regions.

The following figure shows a 4:1 MUX with balanced cascading. The first level of BUFGCTRL buffers are both placed in the directly adjacent sites (X0Y2, X0Y0) of the last BUFGCTRL (X0Y1). This configuration ensures a comparable insertion delay for all the clocks reaching the last BUFGCTRL. You can use an equivalent structure for a 3:1 MUX.

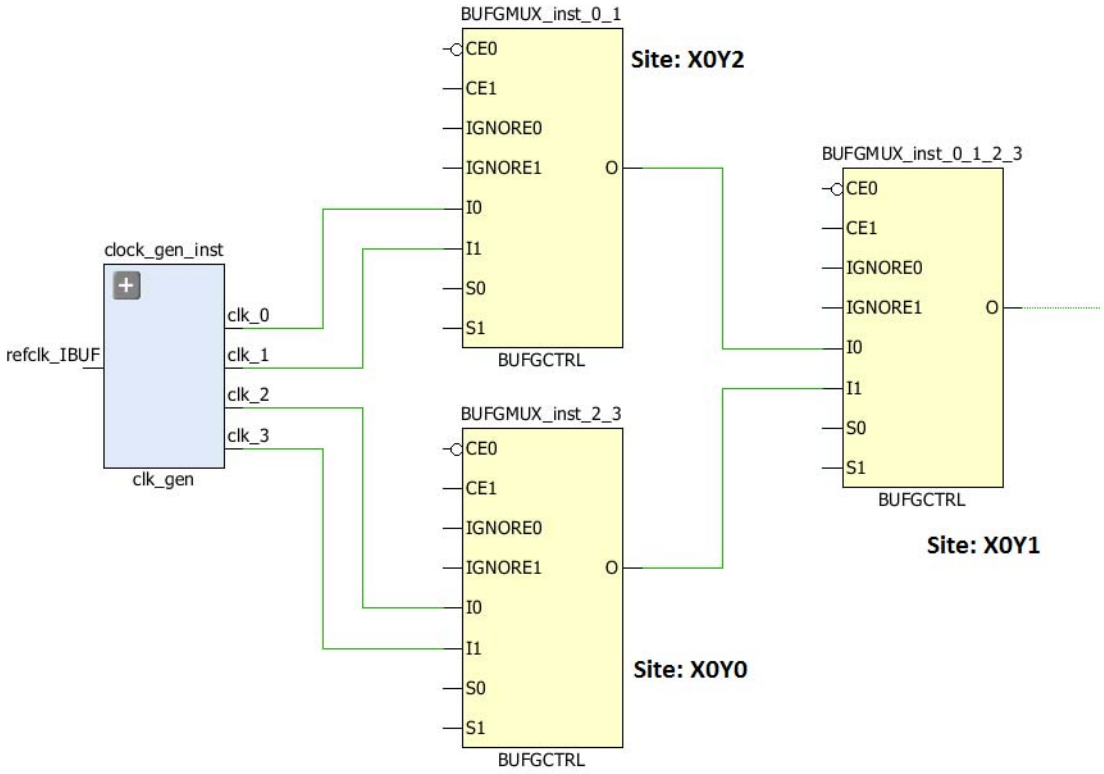



*Figure 3-45:* **4:1 MUX Using Parallel BUFGCTRL**

When creating a 5:1 or larger clock MUX structure, it is common to create a symmetrical clock structure as shown in the following figure. However, this is a sub-optimal solution, because each BUFGCTRL only has one cascade path to the two adjacent BUFGCTRLs, which does not provide minimal delay for all connections between the BUFGCTRLs.

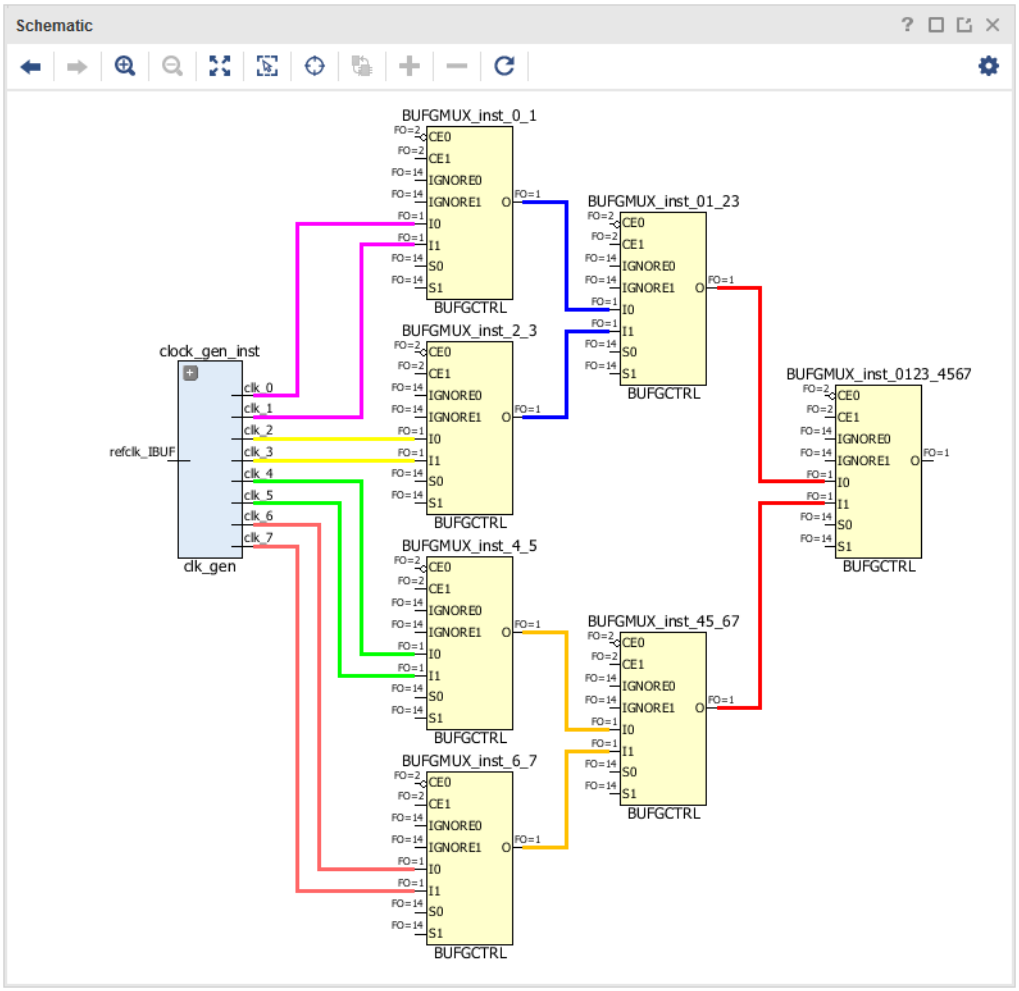



*Figure 3-46:* **Non-Recommended 8:1 Balanced Clock MUX Structure**

To support larger clock multiplexers (from 5:1 to 8:1 MUX), Xilinx recommends using cascaded BUFGCTRL buffers as shown in the following figure. This figure shows an optimal 8:1 MUX that uses 7 BUFGCTRL buffers.

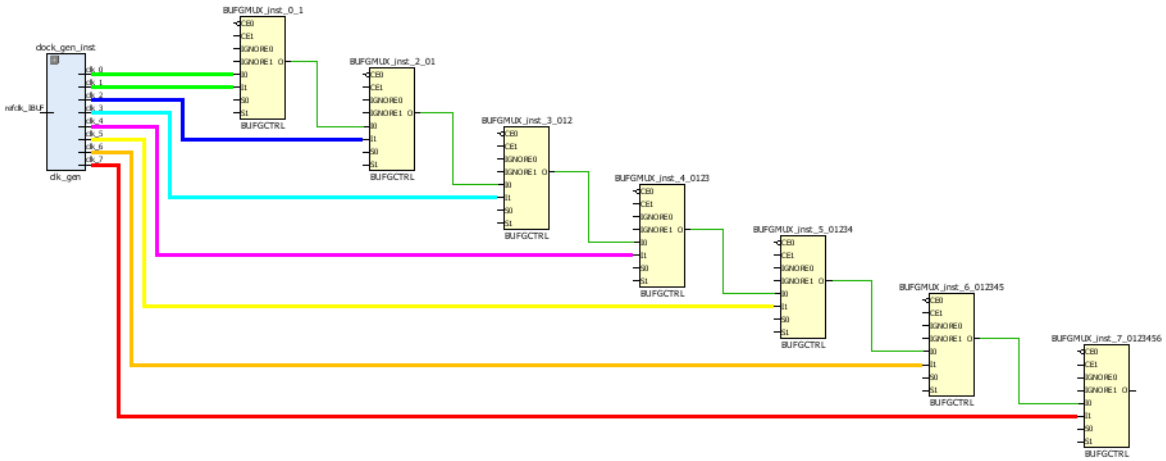



*Figure 3-47:* **8:1 MUX Using Cascaded BUFGCTRL**

***Note:*** When using wide BUFGCTRL-based clock multiplexers, the clock insertion delays cannot be balanced because some paths are longer than other paths in hardware. Therefore, this method is recommended for multiplexing asynchronous clocks only.

### PLL/MMCM Feedback Path and Compensation Mode

PLLs do not support delay compensation and always operate in INTERNAL compensation mode, which means they do not need a feedback path. Similarly, MMCMs set to INTERNAL compensation mode do not need a feedback path. In both cases, the Vivado tools do not always automatically remove unnecessary feedback clock buffers. You must remove the clock buffers manually to reduce the amount of high fanout clock resource utilization. This is especially important for designs with high clocking usage where clock contention might occur.

When the MMCM compensation is set to ZHOLD or BUF_IN, the placer assigns the same clock root to the nets driven by the feedback buffer and by all buffers directly connected to the CLKOUT0 pin. This ensures that the insertion delays are matched so that the I/O ports and the sequential cells connected to CLKOUT0 are phase-aligned and hold time is met at the device interface. The Vivado tools consider all the loads of these nets to optimally define the clock root.

The Vivado tools do not automatically match the insertion delay with the other MMCM outputs. To match the insertion delay for the nets driven by other MMCM output buffers, use the following properties:

- CLOCK_DELAY_GROUP

  Apply the same CLOCK_DELAY_GROUP property value to the nets directly driven by feedback clock buffer, the CLKOUT0 buffers, and the other MMCM output buffers as needed. This is the preferred method.

- USER_CLOCK_ROOT

  If you need to force a specific clock root, use the same USER_CLOCK_ROOT property value on the nets driven by the feedback clock buffer, the CLKOUT0 buffers, and the other MMCM output buffers as needed.

#### BUFG_GT Divider

The BUFG_GT buffers can drive any loads in the fabric and include an optional divider you can use to divide the clock from the GT*_CHANNEL. This eliminates the need to use an extra MMCM or BUFG_DIV to divide the clock.

### *SelectIO Clocking*

The UltraScale device SelectIO primitives have maximum skew requirements between clock pins. Using the optimal clocking topology for the SelectIO primitives prevents maximum skew violations, improves interface timing between the UltraScale device and the fabric logic, and uses fewer clocking resources.

#### ISERDESE3 and IDDRE1 Clocking

For ISERDESE3 and IDDRE1 clocking in UltraScale and UltraScale+ devices, maximum skew requirements exist between the clock and inverted clock pins. To meet the maximum skew requirements, Xilinx recommends using a single net for the clock and inverted clock pins when using the local inversion.

In the following figure, the left side shows a sub-optimal configuration that uses the CLKOUT0B output of the MMCM. The right side of the figure shows the optimal configuration that uses the local inversion on the CLK_B and CB pins of the ISERDESE3 and IDDRE1. Using the optimal configuration guarantees that the maximum skew requirement is met while using fewer global clock resources.

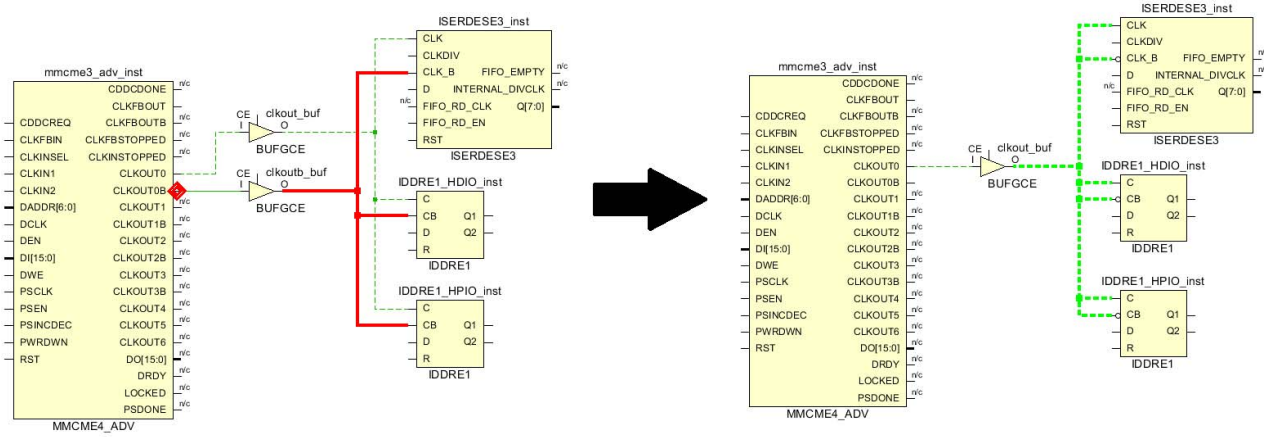



*Figure 3-48:* **Sub-Optimal to Optimal Clocking Topologies for ISERDESE3 and IDDRE1**

### OSERDESE3 Clocking

For OSERDESE3 clocking in UltraScale and UltraScale+ devices, maximum skew requirements exist between the high-speed clock and divided clock pins. To meet the maximum skew requirements, Xilinx recommends using parallel global clock buffers where one of the global clock buffers is a BUFGCE_DIV. This removes the additional clock uncertainty between the two outputs of the MMCM.

In the following figure, the left side shows a sub-optimal configuration that uses two separate outputs of the MMCM. The right side of the figure shows the optimal configuration that uses a single MMCM output and the BUFGCE_DIV cell, which provides the divided clock using the BUFGCE_DIVIDE property.

***Note:*** The high-speed clock does not need to be driven by a BUFGCE. Alternatively, you can use BUFGCE_DIV with a BUFGCE_DIVIDE property setting of 1.

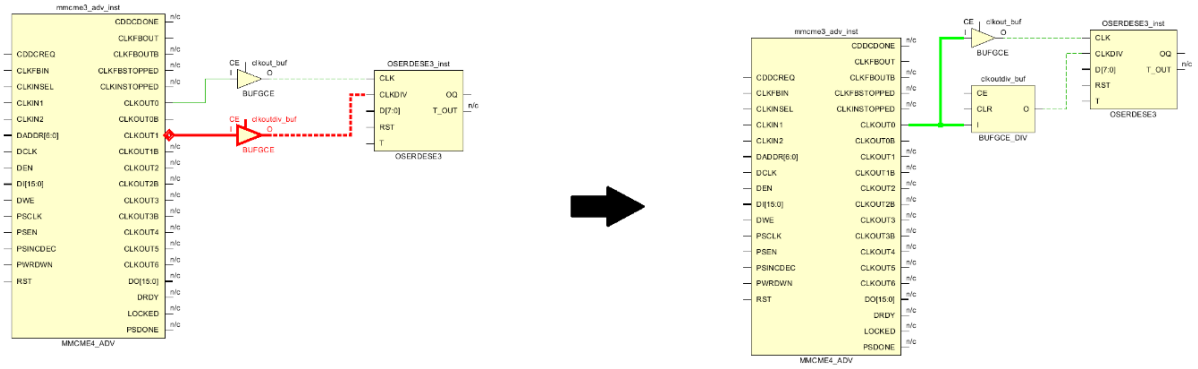



*Figure 3-49:* **Sub-Optimal to Optimal Clocking Topologies for OSERDESE3**

### I/O Timing with MMCM ZHOLD/BUF_IN Compensation

ZHOLD compensation indicates that the MMCM is configured to provide a negative hold for all I/O registers of an entire I/O column. When a clock capable I/O (CCIO) drives a single MMCM that is configured in ZHOLD compensation mode, the placer will attempt to place the MMCM with the CCIO in the same clock region. In this case, the CCIO can drive the MMCM directly without going through a BUFG. This allows the ZHOLD compensation of the MMCM to remain in effect.

However, if a CCIO drives an MMCM configured in ZHOLD mode in addition to another MMCM, logic optimization will attempt to legalize the clock routing to the MMCMs by inserting a BUFG after the CCIO. Because the MMCM with ZHOLD compensation is no longer driven directly by a CCIO, the compensation is changed to BUF_IN. To avoid this, ensure that the CCIO drives the MMCM configured in ZHOLD mode directly and drives the additional MMCM through a BUFG. In addition, set the CLOCK_DEDICATED_ROUTE property for the net driven by the BUFG to ANY_CMT_COLUMN.

Because the clock insertion delay varies with the clock root locations and the clock root placement depends on placement of the loads, there might be variability between runs. This variability affects the timing inside the FPGA as well as the I/O timing.

When dealing with high-frequency I/Os, you might want more control over the I/O timing and less variability between runs. One way to achieve this is to force the clock root placement. You can run the tool in automated mode and look at the clock root region. If the I/O timing is satisfactory, you can force the clock root placement on the buffer nets associated with I/O timing. To determine the placement of the clock roots, use the `report_clock_utilization [-clock_roots_only]` Tcl command.

In the following example, the I/O ports are located in the X0Y0 region. The Vivado placer determined the placement of the clock roots in X1Y2 based on the I/O placement as well as placement of other loads.

| Index | Clock Net | Root Clock Region | Clock Root Node |
|---|---|---|---|
| 1 | mmcm/inst/clk0 | X1Y2 | RCLK_BRAM_L_X30Y209/CLK_VDISTR_B0T0 |
| 2 | mmcm/inst/clkfbout_buf_mmcm_zhold | X1Y2 | RCLK_CLEL_R_L_X25Y209/CLK_VDISTR_B0T |

*Figure 3-50:* **Clock Utilization Summary with Unconstrained Clock Root**

The following summary shows the I/O timing when the clock root is unconstrained.

| Setup | | Hold | |
|---|---|---|---|
| Worst Negative Slack (WNS): | -0.279 ns | Worst Hold Slack (WHS): | 0.036 ns |
| Total Negative Slack (TNS): | -5.394 ns | Total Hold Slack (THS): | 0.000 ns |
| Number of Failing Endpoints: | 47 | Number of Failing Endpoints: | 0 |

*Figure 3-51:* **Timing Summary with Unconstrained Clock Root**

In the following example, the clock roots are moved next to the I/O registers in X0Y0, which reduces the clock insertion delays and timing pessimism and therefore, improves the I/O timing.

| Index | Clock Net | Root Clock Region | Clock Root Node |
|---|---|---|---|
| 1 | mmcm/inst/clk0 | X0Y0 | XIPHY_L_X0Y60/CLK_VDISTR_B0T20 |
| 2 | mmcm/inst/clkfbout_buf_mmcm_zhold | X0Y0 | XIPHY_L_X0Y60/CLK_VDISTR_B0T14 |

*Figure 3-52:* **Clock Utilization Summary with User Constrained Clock Root**

The following summary shows the I/O timing when the clock root is moved.

| Setup | | Hold | |
|---|---|---|---|
| Worst Negative Slack (WNS): | -0.217 ns | Worst Hold Slack (WHS): | 0.060 ns |
| Total Negative Slack (TNS): | -1.488 ns | Total Hold Slack (THS): | 0.000 ns |
| Number of Failing Endpoints: | 16 | Number of Failing Endpoints: | 0 |

*Figure 3-53:* **Timing Summary with User Constrained Clock Root**

### *Synchronous CDC*

When the design includes synchronous CDC paths between clocks that originate from the same MMCM/PLL, you can use the following techniques to better control the clock insertion delays and skew and therefore, the slack on those paths.

**IMPORTANT:** *If the CDC paths are between clocks that originate from different MMCM/PLLs, the clock insertion delays across the MMCMs/PLLs are more difficult to control. In this case, Xilinx recommends that you treat these clock domain crossings as asynchronous and make design changes accordingly.*

When a path is timed between two clocks that originate from different output pins of the same MMCM/PLL, the MMCM/PLL phase error adds to the clock uncertainty for the path. For designs using high clock frequencies, the phase error can cause issues with timing closure both for setup and hold.

The following figure shows an example of paths both with and without the phase error. Path 1 is a CDC path clocked by two buffers connected to the same MMCM output and does not include the phase error. Path 2 is clocked by two clocks that originate from two different MMCM outputs and does include the phase error.

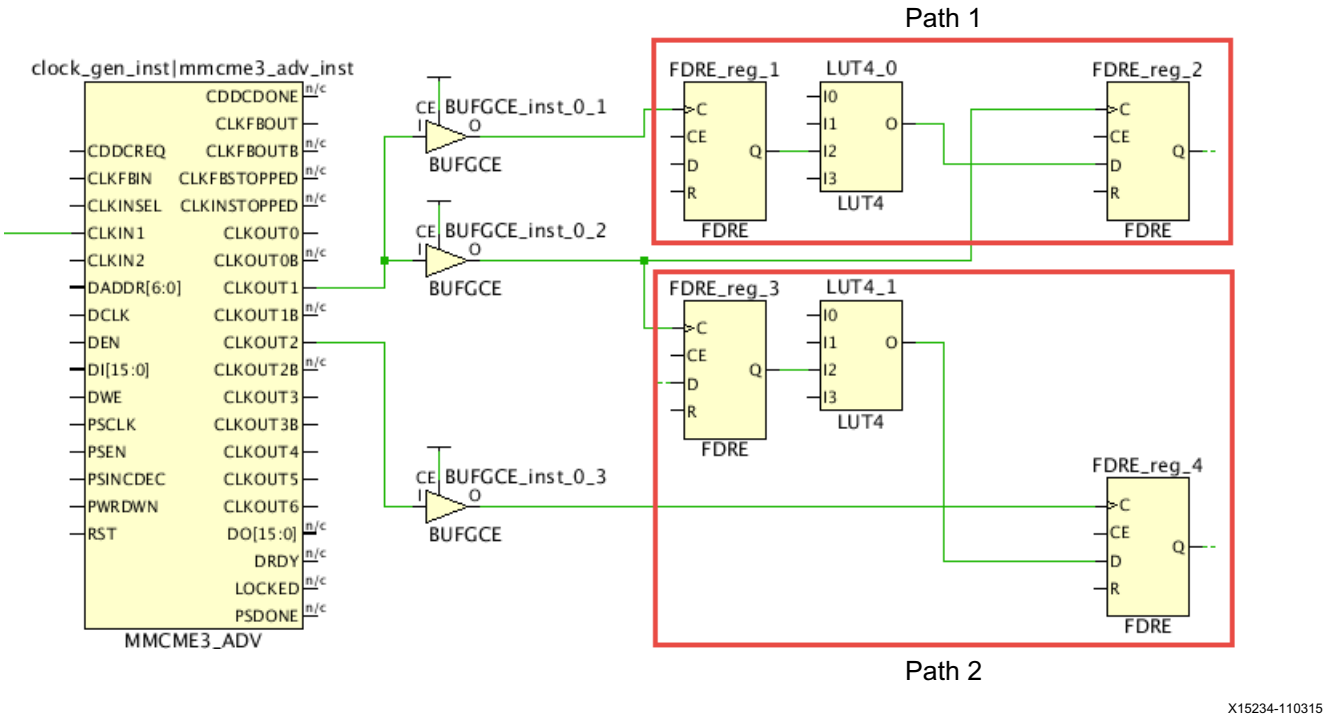



*Figure 3-54:* **MMCM and Phase Error**

When two synchronous clocks from the same MMCM/PLL have a simple period ratio (/2 /4 /8), you can prevent the phase error between the two clock domains using a single MMCM/PLL output connected to two BUFGCE_DIV buffers. The BUFGCE_DIV buffer performs the clock division (/1 /2 /4 /8). Other ratios are possible (/3 /5 /7) but this requires modifying the clock duty cycle and making mixed edge timing paths more challenging.

***Note:*** Because the BUFGCE and BUFGCE_DIV do not have the same cell delays, Xilinx recommends using the same clock buffer for both synchronous clocks (two BUFGCE or two BUFGCE_DIV buffers).

The following figure shows two BUFGCE_DIVs that divide the CLKOUT0 clock by 1 and by 2 respectively.

***Note:*** Although the following figure only uses two BUFGCE_DIVs in parallel, you can use up to four BUFGCE_DIVs in a clock region.

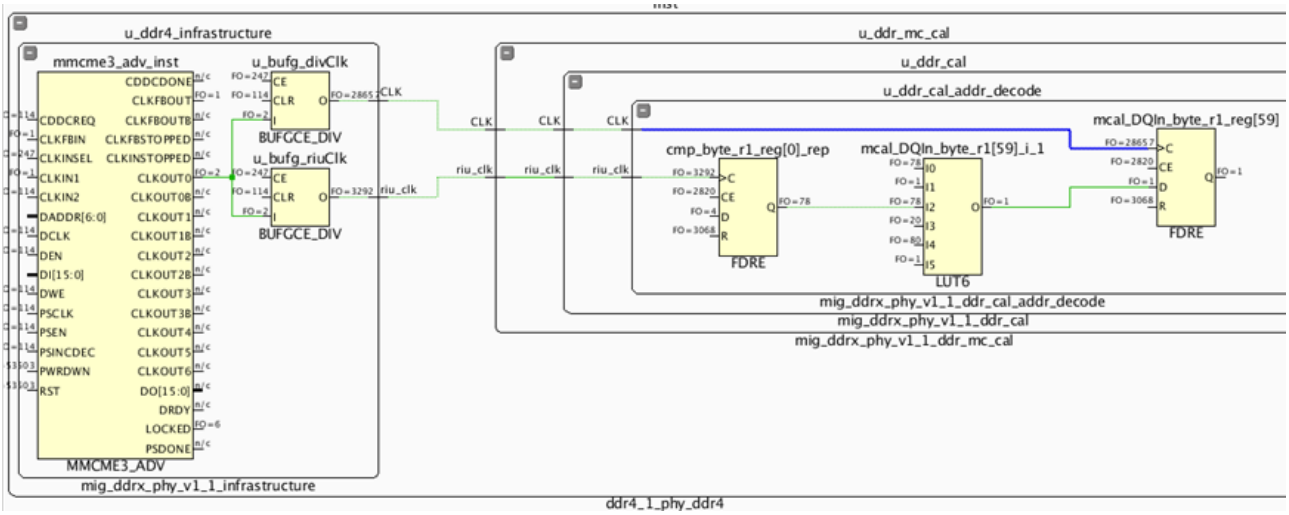



*Figure 3-55:* **MMCM Synchronous CDC with BUFGCE_DIVs Connected to One MMCM Output**

To automatically balance several clocks that originate from the same MMCM or PLL, set the same CLOCK_DELAY_GROUP property value on the nets driven by the clock buffers that need to be balanced. Following are additional recommendation:

- Avoid setting the CLOCK_DELAY_GROUP constraint on too many clocks, because this stresses the clock placer resulting in sub-optimal solutions or errors.
- Review the critical synchronous CDC paths in the Timing Summary Report to determine which clocks must be delay matched to meet timing.
- Limit the use of the CLOCK_DELAY_GROUP on groups of synchronous clocks with tight requirements and with identical clocking topologies.

**IMPORTANT:** *Xilinx recommends using the Clocking Wizard for creating optimal clocking structures, which use a mix of BUFGCEs and BUFGCE_DIVs along with related clock grouping constraints.*

## *GT Interface Clocking*

Each GT interface requires several clocks, including some clocks that are shared across bonded GT*_CHANNEL cells located in one or several GT quads. UltraScale devices provide up to 128 GT*_CHANNEL sites, which can lead to the use of several hundreds of clocks in a design. Most GT clocks have a low fanout with loads placed locally in the clock region next to the associated GT*_CHANNEL. Some GT clocks drive loads across the entire device and require the utilization of clock routing resource in many clock regions. The UltraScale

architecture includes the following enhancements to efficiently support the high number of GT clocks required.

### BUFG_GT with Dynamic Divider

In UltraScale devices, the BUFG_GT buffer simplifies GT clocking. Because the BUFG_GT includes dynamic division capabilities, MMCMs are no longer required to perform simple integer divides on GT output clocks. This saves clocking resources and provides an improved low skew clock path when both a divided GT*_CHANNEL output clock and full-rate clock are required.

You can use the BUFG_GT global clock buffer for GT interfaces where the user logic operates at half the clock frequency of the internal PCS logic or for PCIe® interfaces where the GT*_CHANNEL needs to generate multiple clock frequencies for user_clk, sys_clk, and pipe_clk. The following figure compares clocking requirements between 7 series and UltraScale devices for a single-lane GT interface where the frequency of TXUSRCLK2 is equal to half of the frequency of TXUSRCLK.

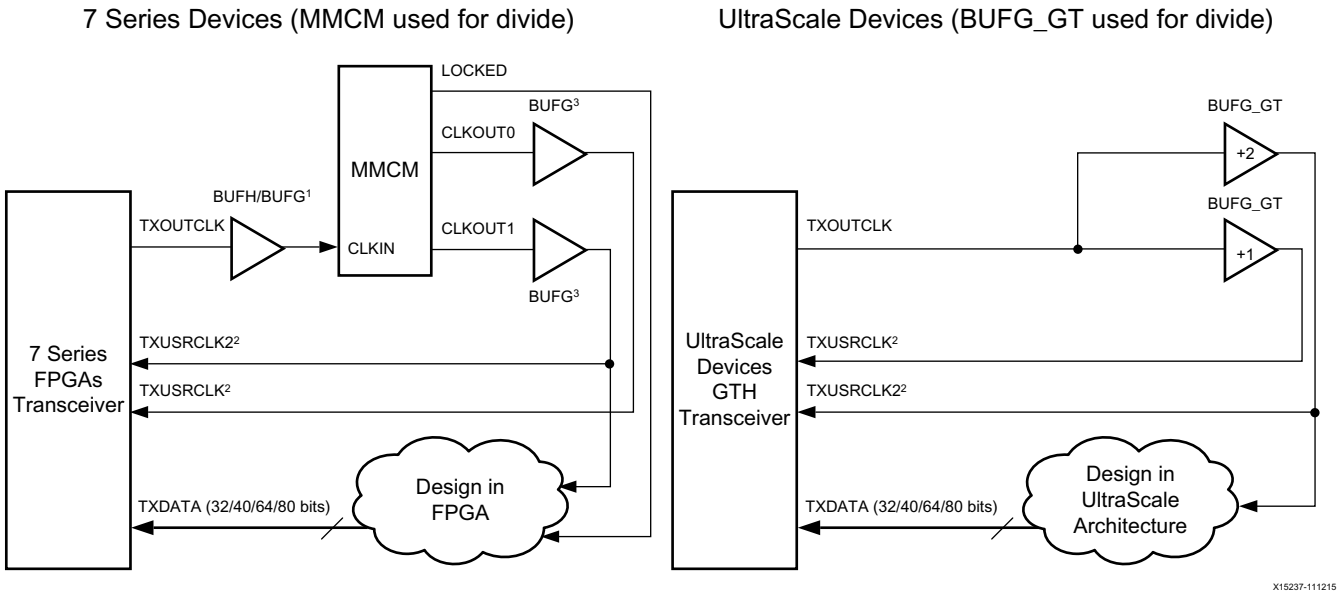



*Figure 3-56:* **Clocking Requirements Comparison**

You can use any output clock of the GT*_CHANNELs within a Quad or any reference clock generated by an IBUFDS_GTE3/ODIV2 pin within a Quad to drive any of the 24 BUFG_GT buffers located in the same clock region. A BUFG_GT_SYNC is always required to synchronize reset and clear of BUFG_GTs driven by a common clock source.

***Note:*** The Vivado tools automatically insert the BUFG_GT_SYNC primitive if it is not present in the design.

Some applications still require the use of an MMCM to generate complex non-integer clock division of the GT output clocks or the IBUFDS_GTE3/ODIV2 reference clock. In these cases, a BUFG_GT must directly drive the MMCM. By default, the placer tries to place the MMCM on the same clock region row as the BUFG_GT. If other MMCMs try to use the same MMCM site, you must verify that the automated MMCM placement is still as close as possible to the BUFG_GT to avoid wasting clocking resources due to long routes.

### Single Quad vs. Multi-Quad Interface

In a multi-channel interface, a master channel can generate [RT]XUSRCLK[2] for all the GT*CHANNELs of the interface. If a multi-channel interface spans multiple quads, the maximum allowed distance for a GT*CHANNEL from the reference clock source is 2 clock regions above and below.

The following figure shows a multi-quad interface. The GT*CHANNELs are marked in yellow, the TXUSRCLK is highlighted in blue, and the TXUSRCLK2 is highlighted in red. The BUFG_GTs driving both TXUSRCLK and TXUSRCLK2 are located in the center quad and are marked in blue and red.

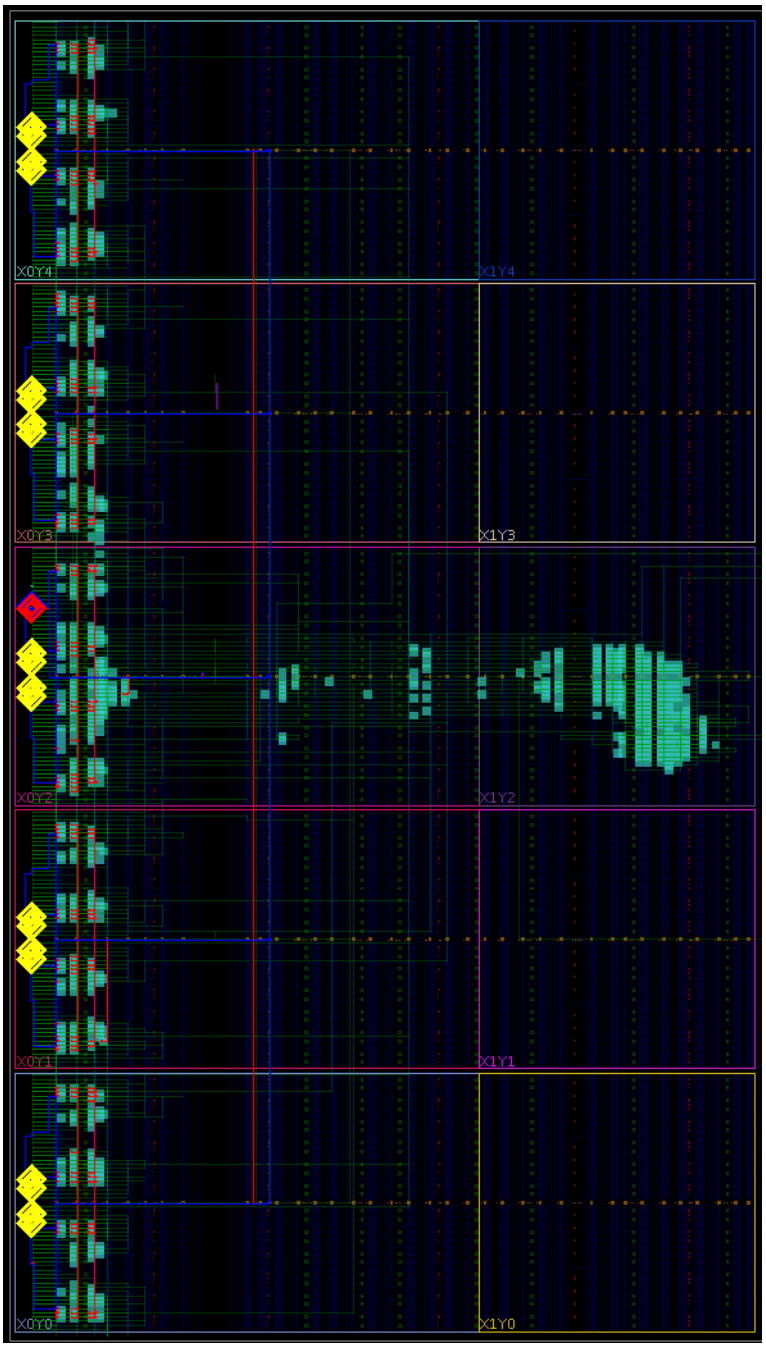



*Figure 3-57:* **TXUSRCLK/TXUSRCLK2 Clock Routing for a Multi-Quad Interface**

If the GT interface is contained within a single Quad, the placer treats the BUFG_GT clocks as local clocks. In this case, the placer attempts to place the BUFG_GT clock loads in the clock regions horizontally adjacent to the BUFG_GT, starting with the clock region that contains the BUFG_GT and potentially using up to half the width of the device.

To override the placer regional clock constraint, assign any of the BUFG_GT clock loads to a Pblock. The following figure shows a single-quad interface. The GT*CHANNELs are marked in yellow, the TXUSRCLK is highlighted in blue, and the TXUSRCLK2 is highlighted in red. All the TXUSRCLK2 loads are placed in the same clock region as the GT*CHANNELs.

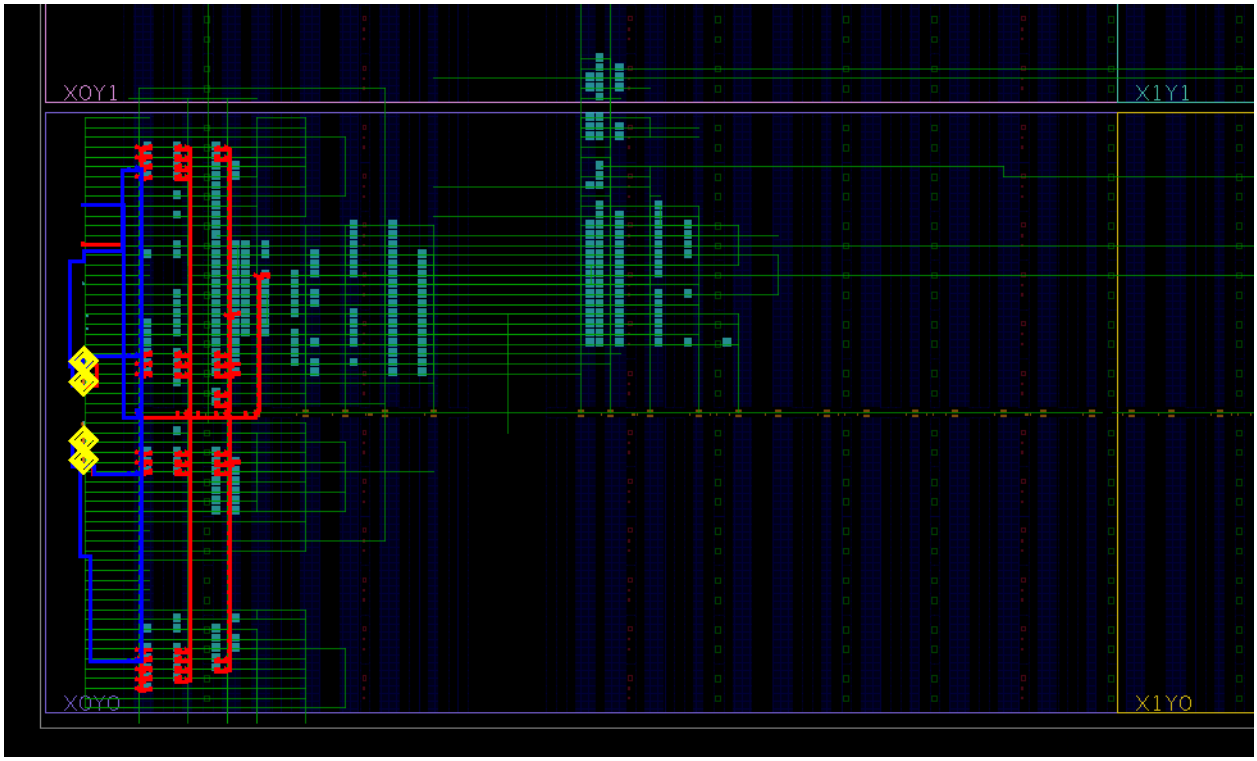



*Figure 3-58:* **TXUSRCLK/TXUSRCLK2 Clock Routing for a Single-Quad Interface**

### [RT]XUSRCLK/[RT]XUSRCLK2 Skew Matching

When [RT]XUSRCLK2 operates at half the frequency of [RT]XUSRCLK (i.e., separate BUFG_GTs with divide by 1 and divide by 2), a tight skew requirement exists between the [RT]XUSRCLK/[RT]XUSRCLK2 pair at each GT*CHANNEL of a GT interface. To meet the skew requirement, GT*CHANNELs can be a maximum of 2 clock regions above or below the master channel that generates the [RT]XUSRCLK/[RT]XUSRCLK2 pair. In addition, the placer tightly controls skew as follows:

- Assigns the BUFG_GT pairs to the upper or lower 12 BUFG_GTs in a Quad
- Assigns the clock root for both clocks to the clock region containing the BUFG_GTs

**RECOMMENDED:** *To avoid skew violations, Xilinx highly recommends following this clocking topology when [RT]XUSRCLK2 operates at half the frequency of [RT]XUSRCLK.*

#### Integrated Block for PCI Express CORECLK/PIPECLK/USERCLK Skew Matching

The UltraScale Integrated Block for PCI Express® requires three clocks: CORECLK, USERCLK, and PIPECLK. The three clocks are sourced by BUFG_GTs driven by the TXOUTCLK pin of one of the GT*_CHANNELs of the physical interface. A tight skew requirement exists between the CORCLK and PIPECLK pins and the CORECLK and USERCLK pins. To meet the skew requirement, the placer tightly controls skew as follows:

- Assigns the BUFG_GTs that drive the three PCIe clocks in groups to the upper or lower 12 BUFG_GTs in a Quad
- Assigns the clock root for all three clocks to the same clock region

  ***Note:*** For more information on PCIe clocking requirements, see the *UltraScale Architecture Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156) [Ref 42].

## 7 Series Device Clocking

***Note:*** This section uses Virtex®-7 clocking resources as an example. The clocking resources for Virtex-6 devices are similar. If you are using a different architecture, see the *Clocking Resources Guide* [Ref 40] for your device.

Virtex-6 and Virtex-7 devices contain thirty-two global clock buffers known as BUFGs. BUFGs can serve most clocking needs for designs with less demanding needs in terms of number of clocks, design performance, and clocking control. Global clocking resources include BUFG, BUFG, BUFGCE, BUFGMUX, and BUFGCTRL primitives, which each have their own features. For more information on the features of these global clock components, see the *Clocking Resources Guide* [Ref 40] and *Libraries Guide* for your device.

**RECOMMENDED:** *If clocking demands exceed the number of BUFGs, or if better overall clocking characteristics are desired, analyze the clocking needs against the available clocking resources, and select the best resource for the task.*

In addition to global clocking resources, regional clocking resources are also available, which allow tighter control of clock networks. Regional clocking resources include the Horizontal Clock Region Buffers (BUFH, BUFHCE), Regional Clock Buffer (BUFR), I/O Clock Buffer (BUFIO), and Multi-Regional Clock Buffer (BUFMR). For more information on the features of these regional clock components, see the see the *Clocking Resources Guide* [Ref 40] and *Libraries Guide* for your device.

### *Using Horizontal Clock Region Buffers for Clock Gating*

You can use the Horizontal Clock Region Buffer (BUFHCE) in conjunction with BUFGs to perform a medium-grained clock gating function. For portions of a clock domain ranging

from a few hundred to a few thousand loads in which you want to stop clocking intermittently, the BUFHCE can be an effective clocking resource. A BUFG can drive multiple BUFHCEs in the same or different clock regions, which allows you to individually control clocking in several low clock skew domains.

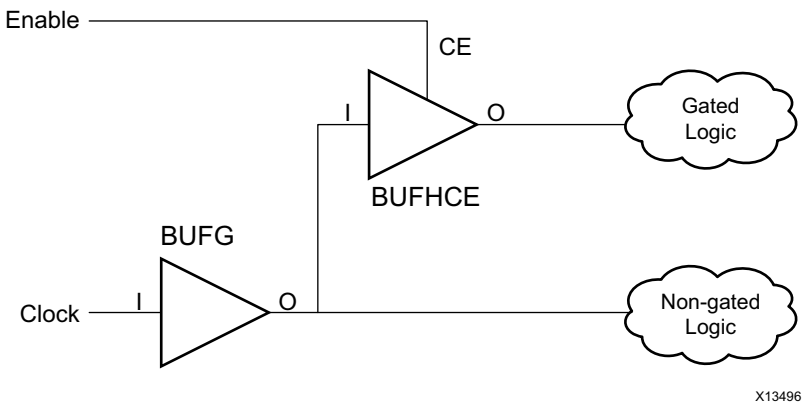



*Figure 3-59:* **Horizontal Clock Region Buffers**

When used independently, all loads connected to the BUFH must reside in the same clock region. This makes it well-suited for very high-speed, more fine-grained (fewer loads) clocking needs. BUFHCE can be used to achieve medium-grained clock-gating within the specific clock region. You must ensure that the resources driven by the BUFH do not exceed the available resources in the clock region and that no other conflicts exist.

The phase relationship might be different between the BUFH and clock domains driven by BUFGs, other BUFHs, or any other clocking resource. The single exception is when two BUFHs are driven to horizontally adjacent regions. In this case, the skew between left and right clock regions when both BUFHs driven by the same clock source should have a very controlled phase relationship in which data may safely cross the two BUFH clock domains. BUFHs can be used to gain access to MMCMs or PLLs in opposite regions to a clock input or GT. However, care must be taken in this approach to ensure that the MMCM or PLL is available.

### *Additional Clocking Considerations for SSI Devices*

In general, all clocking considerations mentioned above also apply to SSI technology devices. However, there are additional considerations when targeting these devices due to their construction. When using a BUFMR, it cannot drive clocking resources across an SLR boundary. Accordingly, Xilinx recommends that you place the clocks driving BUFMRs into the bank or clocking region in the center clock region within an SLR. This gives access to all three clock regions on the left or right side of the SLR.

In terms of global clocking, for designs requiring sixteen or fewer global clocks (BUFGs), no additional considerations are necessary. The tools automatically assign BUFGs in a way to avoid any possible contention. When more than sixteen (but fewer than thirty-two) BUFGs

are required, some consideration to pin selection and placement must be done in order to avoid any chance of contention of resources based on global clocking line contention and/or placement of clock loads.

As in all other Xilinx 7 series devices, Clock-Capable I/Os (CCIOs) and their associated Clock Management Tile (CMT) have restrictions on the BUFGs they can drive within the given SLR. CCIOs in the top or bottom half of the SLR can drive BUFGs only in the top or bottom half of the SLR (respectively). For this reason, pin and associated CMT selection should be done in a way in which no more that sixteen BUFGs are required in either the top or bottom half of all SLRs collectively. In doing so, the tools can automatically assign all BUFGs in a way to allow all clocks to be driven to all SLRs without contention.

For designs that require more than thirty-two global clocks, Xilinx recommends that you explore using BUFRs and BUFHs for smaller clock domains to reduce the number of needed global clock domains. BUFRs with the use of a BUFMR to drive resources within three clock regions that encompasses one-half of an SLR (approximately 250,000 logic cells in a Virtex-7 class SLR). Horizontally adjacent clock regions may have both left and right BUFH buffers driven in a low-skew manner enabling a clocking domain of one-third of an SLR (approximately 167,000 logic cells).

Using these resources when possible not only leads to fewer considerations for clocking resource contention, but many times improves overall placement, resulting in improved performance and power.

If more than thirty-two global clocks are needed that must drive more than half of an SLR or to multiple SLRs, it is possible to segment the BUFG global clocking spines. Isolation buffers exist on the vertical global clock lines at the periphery of the SLRs that allow use of two BUFGs in different SLRs that occupy the same vertical global clocking track without contention. To make use of this feature, more user control and intervention is required. In the figure below, BUFG0 through BUFG2 in the three SLRs have been isolated, and hence have independent clocks within their respective SLRs. On the other hand, the BUFG31 line has not been isolated. Hence, the same BUFG31 (located in SLR2 in the figure) drives the clock lines in all the 3 SLRs - and BUFG31 located in other SLRs should be disabled.

Careful selection and manual placement (LOCs) must be used for the BUFGs. Additionally, all loads for each clock domain must be manually grouped and placed in the appropriate SLR to avoid clocking contention. If all global clocks are placed and all loads managed in a way to not create any clocking contention and allow the clock to reach all loads, this can allow greater use of the global clocking resources beyond thirty-two.

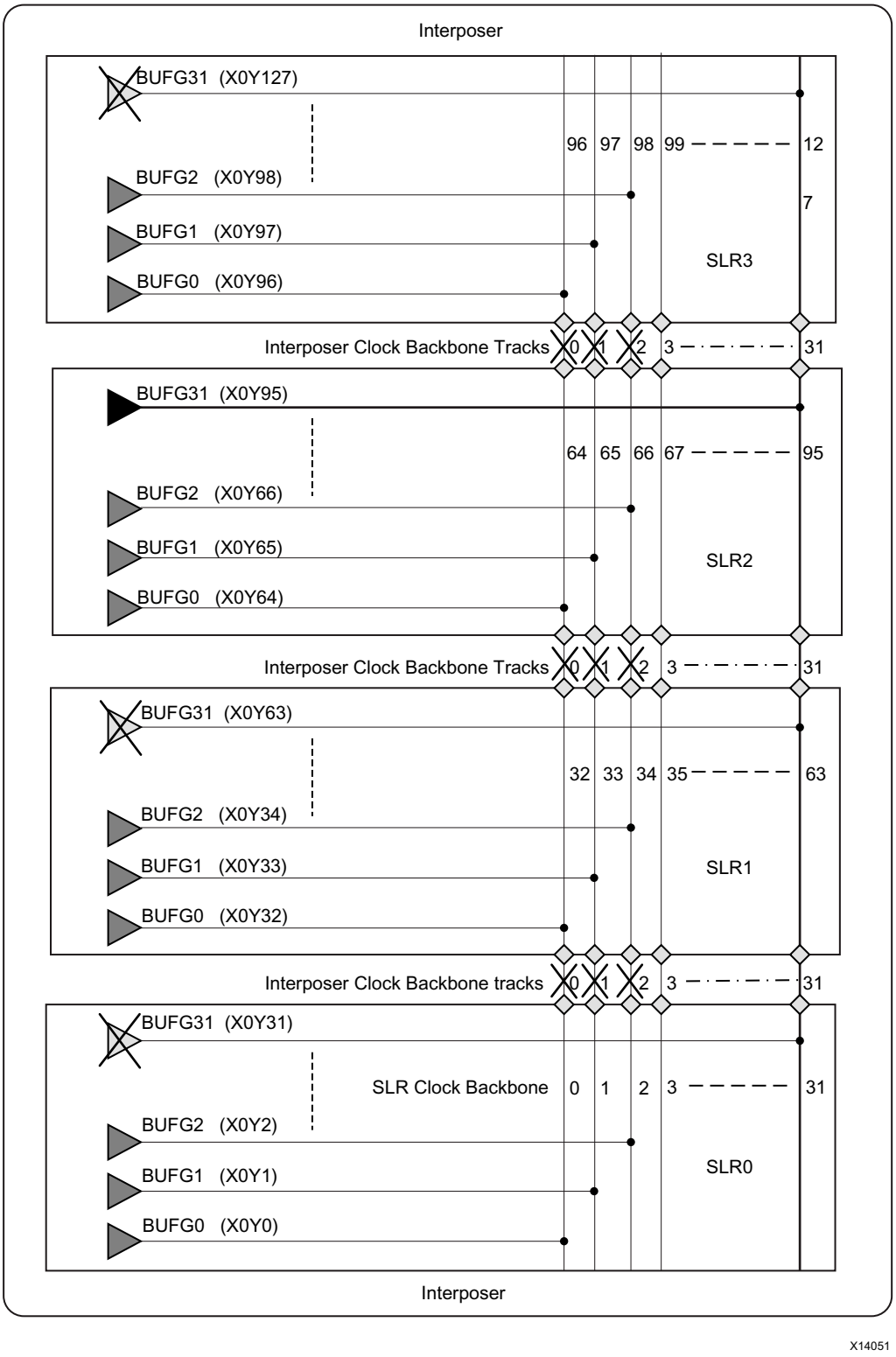



*Figure 3-60:* **Optional Isolation on Clock Lines for SSI Devices**

## Clock Skew for Global Clocking Resources in SSI Technology Devices

Clock skew in any large FPGA device may represent a significant portion of the overall timing budget for a given path. Too much clock skew may not only represent issues with maximum clock speed, but may also manifest itself into stringent hold time requirements. Having multiple die in a device worsens the process portion of the PVT equation, but is managed by the Xilinx assembly process in which only die of similar speed are packaged together.

Even with that extra action, the Xilinx timing tools accounts for these differences as a part of the timing report. During path analysis, these aspects are analyzed as a part of the setup and hold calculations, and are reported as a part of the path delay against the specified requirements. No additional user calculations or consideration are necessary for SSI technology devices, because the timing analysis tools consider these factors in their calculations.

Skew can increase if using the top or bottom SLR as the delay-differential is higher among points farther away from each other. For this reason, Xilinx recommends for global clocks that must drive more than one SLR to be placed into the center SLR. This allows a more even distribution of the overall clocking network across the part resulting in less overall clock skew.

When targeting UltraScale devices, there is less repercussion to clock placement. However, it is still highly suggested to place the clock source as close as possible to the central point of the clock loads to reduce clock insertion delay and improve clock power.

## Designing the Clock Structure

Now that you understand the major considerations for clocking decisions, let us see how you can achieve the desired clocking for your design.

### *Inference*

Without user intervention, Vivado synthesis automatically specifies a global buffer (BUFG) for all clock structures up to the maximum allowed in an architecture (unless otherwise specified or controlled by the synthesis tool). As discussed above, the BUFG provides a well-controlled, low-skew network suitable for most clocking needs. Nothing additional is required unless your design clocking exceeds the number or capabilities of BUFGs in the part.

Applying additional control of the clocking structure, however, may prove to show better characteristics in terms of jitter, skew, placement, power, performance, or other characteristics.

### *Synthesis Constraints and Attributes*

A simple way to control clocking resources is to use the CLOCK_BUFFER_TYPE synthesis constraint or attribute. Synthesis constraints may be used to:

- Prevent BUFG inference.
- Replace a BUFG with an alternative clocking structure.
- Specify a clock buffer where one would not exist otherwise.

Using synthesis constraints allows this type of control without requiring any modification to the code.

Attributes can be placed in either of the following locations:

- Directly in the HDL code, which allows them to persist in the code
- As constraints in the XDC file, which allows this control without any changes needed to the source HDL code

### *Use of IP*

Certain IP assists in the creation of the clocking structures. Clocking Wizard and I/O Wizard specifically can assist in the selection and creation of the clocking resources and structure, including:

- BUFG
- BUFGCE
- BUFGCE_DIV (UltraScale devices)
- BUFGCTRL
- BUFIO (7 series devices)
- BUFR (7 series devices)
- Clock modifying blocks such as:
  - Mixed Mode Clocking Manager (MMCM)
  - Phase Lock Loop (PLL) components

More complex IP such as memory Interface Generator (MIG), PCIe, or Transceiver Wizard may also include clocking structures as part of the overall IP. This may provide additional clocking resources if properly taken into account. If not taken into account, it may limit some clocking options for the remainder of the design.

Xilinx highly recommends that, for any instantiated IP, the clocking requirements, capabilities and resources are well understood and leveraged where possible in other portions of the design.

For more information, see Working With Intellectual Property (IP).

### *Instantiation*

The most low-level and direct method of controlling clocking structures is to instantiate the desired clocking resources into the HDL design. This allows you to access all possible capabilities of the device and exercise absolute control over them. When using BUFGCE, BUFGMUX, BUFHCE, or other clocking structure that requires extra logic and control, instantiation is generally the only option. However, even for simple buffers, sometimes the quickest way to obtain a desired result is to be direct and instantiate it into your design.

An effective style to manage clocking resources (especially when instantiating) is to contain the clocking resources in a separate entity or module instantiated at the top or near the top of the code. By having it at the top-level of code, it may more easily be distributed to multiple modules in your design.

Be aware of where clocking resources can and should be shared. Creating redundant clocking resources is not only a waste of resources, but generally consume more power, create more potential conflicts and placement decisions resulting in longer overall implementation tool runtimes and potentially more complex timing situations. This is another reason why having the clocking resources near the top module is important.

**TIP:** *You can use Vivado HDL templates to instantiate specific clocking primitives. See Using Vivado Design Suite HDL Templates.*

# Controlling the Phase, Frequency, Duty-Cycle, and Jitter of the Clock

This section provides techniques for fine-tuning the clock characteristics.

## *Using Clock Modifying Blocks (MMCM and PLL)*

You can use an MMCM or PLL to change the overall characteristics of an incoming clock. An MMCM is most commonly used to remove the insertion delay of the clock (phase align the clock to the incoming system synchronous data) or for conditioning and controlling the clock characteristics, such as:

- Creating tighter control of phase
- Filtering jitter in the clock
- Changing the clock frequency
- Correcting or changing the clock duty cycle

To use the MMCM or PLL, several attributes must be coordinated to ensure that the MMCM is operating within specifications and delivering the desired clocking characteristics on its output. For this reason, Xilinx highly recommends that you use the Clocking Wizard to properly configure this resource.

You can also directly instantiate the MMCM or PLL, which allows even greater control. However, be sure to use the proper settings to avoid causing the following issues:

- Increasing clock uncertainty due to increased jitter
- Building incorrect phase relationships
- Making timing closure more difficult

**IMPORTANT:** *When using the Clocking Wizard to configure the MMCM or PLL, the Clocking Wizard by default attempts to configure the MMCM for low output jitter using reasonable power characteristics.*

Depending on your goals, you can change the settings in the Clocking Wizard to further minimize jitter and thus, improve timing at the cost of higher power. Alternatively, you can reduce power but increase output jitter.

While using MMCM or PLL, be sure to do the following:

- Do not leave any inputs floating. Relying on synthesis or other optimization tools to tie off the floating values is not recommended, because the values might be different than expected.
- Connect RST to the user logic, so that it can be asserted by logic controlled by a reliable clocking source. Grounding of RST can cause problems if the clock is interrupted.

- Use LOCKED output in the implementation of reset. For example, hold the synchronous logic clocked from the PLL in reset until LOCKED is asserted. The LOCKED signal must be synchronized before it is used in a synchronous portion of the design. Xilinx recommends adding LOCKED to a processor map so it is visible when debugging.
- Confirm the connectivity between CLKFBIN and CLKFBOUT. The BUFG only needs to be included in the feedback path if the PLL/MMCM output clock needs to be phase aligned with the input reference clock, for example, when using ZHOLD compensation mode.
- To avoid the MMCM or PLL phase error timing penalty on synchronous clock domain crossing paths in UltraScale devices, use BUFGCE_DIVs instead of BUFGCE. For details, see Synchronous CDC.

**RECOMMENDED:** *Explore the different settings within the Clocking Wizard to ensure that the most desirable configuration is created based on your overall design goals.*

### *Using IDELAYs on Clocks to Control Phase*

For 7 series devices, if only minor phase adjustments are necessary, you can use IDELAY or ODELAY (instead of MMCM or PLL) to add additional delay. This increases the phase offset of the clock in relation to any associated data. When using UltraScale devices, you cannot use an IDELAY on an input clock source. Therefore, if phase manipulation is necessary, Xilinx recommends using an MMCM.

## Using Gated Clocks

Xilinx devices include dedicated clock networks that can provide a large-fanout, low-skew clocking resource. Fine-grained clock gating techniques included in the HDL code can disrupt the functionality and prevent efficient use of the dedicated clocking resources. Therefore, when writing HDL to target a device, Xilinx does not recommend that you code clock gating constructs into the clock path. Instead, control clocking by using coding techniques to infer clock enables in order to stop portions of the design, either for functionality or power reasons.

### *Converting Clock Gating to Clock Enable*

If the code already contains clock gating constructs, or if it is intended for a different technology that requires such coding styles, Xilinx recommends that you use a synthesis tool that can remap gates placed within the clock path to clock enables in the data path. Doing so allows for a better mapping to the clocking resources; and simplifies the timing analysis of the circuit for data entering and exiting the gated domain. For example, use the `-gated_clock_conversion auto` option with Vivado synthesis to attempt automatic conversion to register clock enable logic. For the complex gated clock structures, use the GATED_CLOCK attribute in the RTL code to guide Vivado synthesis.

### *Gating the Clock Buffer*

When larger portions of the clock network can be shut down for periods of time, you can enable or disable the clock network using a BUFGCE or BUFGCTRL. In addition, when targeting UltraScale devices, you can gate the BUFGCE_DIV and BUFG_GT. For 7 series devices, you can also use the BUFHCE, BUFR, and BUFMRCE to gate the clock.

When a clock can be slowed down during periods of time, you can also use these buffers with additional logic to periodically enable the clock net. Alternatively, you can use a BUFGMUX or BUFGCTRL to switch the clock source from a faster clock signal to a slower clock.

Any of these techniques can effectively reduce dynamic power. However, depending on the requirements and clock topology, one technique may prove more effective than another. For example, in 7 series devices:

- A BUFR might work best if it is an externally generated clock (under 450 MHz) that is only needed to source up to three clock regions.
- For Virtex-7 devices, a BUFMRCE might also be needed to use this technique with more than one clock region (but only up to three vertically adjacent regions).
- A BUFHCE is better suited for higher-speed clocks that can be contained in a single clock region. Although a BUFGCE may span the device and is the most flexible approach, it might not be the best choice for the greatest power savings.

### *Controlling and Synchronizing Device Startup*

After the device completes configuration, a sequence of events occurs in which the device completes the configuration state and enters into general operation. In most configuration sequences, one of the last steps is the deassertion of the Global Set Reset (GSR), followed by the deassertion of the Global Enable (GWE) signal. When this happens, the design is in a known initial state and is then released for operation.

If this release point is not synchronized to the given clock domain or if the clock is operating at a faster time than the GWE can safely be released, portions of the design can go into an unknown state. For some designs, this does not matter. In other designs, this can cause the design to become unstable or to incorrectly process the initial data set.

If the design must start up in a known state, Xilinx recommends that you take action to control the start-up synchronization process using any of the following methods:

- Use instantiated clock buffer components with clock enable capability. Delay the reset release by as many cycles as needed before enabling the design clock. The following example shows how to delay the first design clock edge after the reset is released in an UltraScale device. By setting ASYNC_REG=TRUE on the synchronizer registers, all registers are placed in a single SLICE and therefore, do not need to be driven by a global clock resource. To prevent clock buffer insertion on the synchronizer clock, use the CLOCK_BUFFER_TYPE=NONE property on the input clock port.

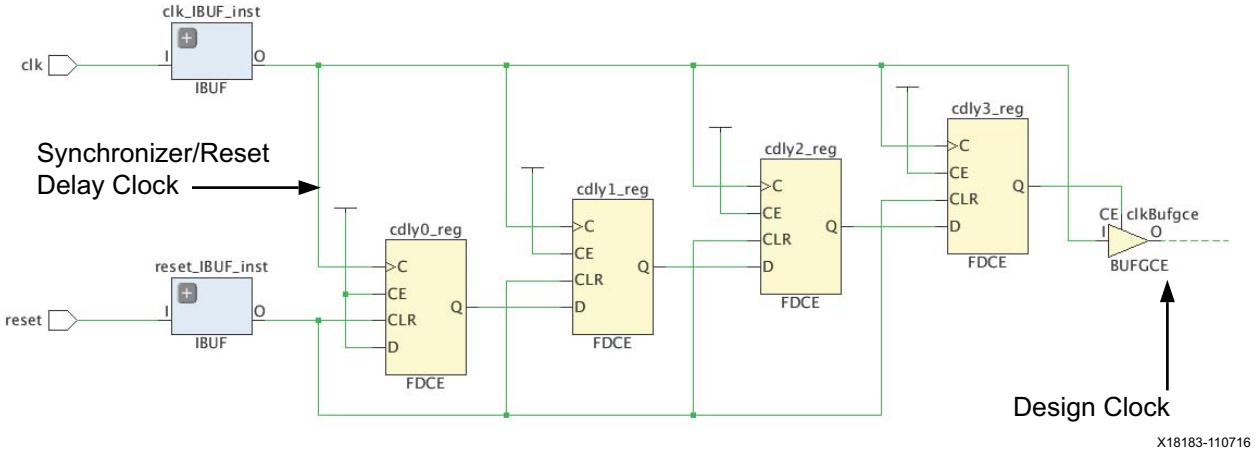



*Figure 3-61:* **Reset Synchronization and Delay for Safe Clock Startup Example**

- When using an MMCM, you can select the **Safe Clock Startup** option from the Clocking Wizard to ensure that design clocks are enabled only after they are stable and reliable. The following example shows the synchronization stages of an UltraScale device MMCM LOCKED signal connected to the CE pin of the BUFGCE, which drives the user logic. A second BUFGCE is connected in parallel to the high fanout BUFGCE (user clock) and is dedicated to the logic controlling the BUFGCE/CE pin. This topology helps timing closure on the BUFGCE/CE in UltraScale devices by minimizing the clock skew between the synchronizer and the BUFGCE pin.

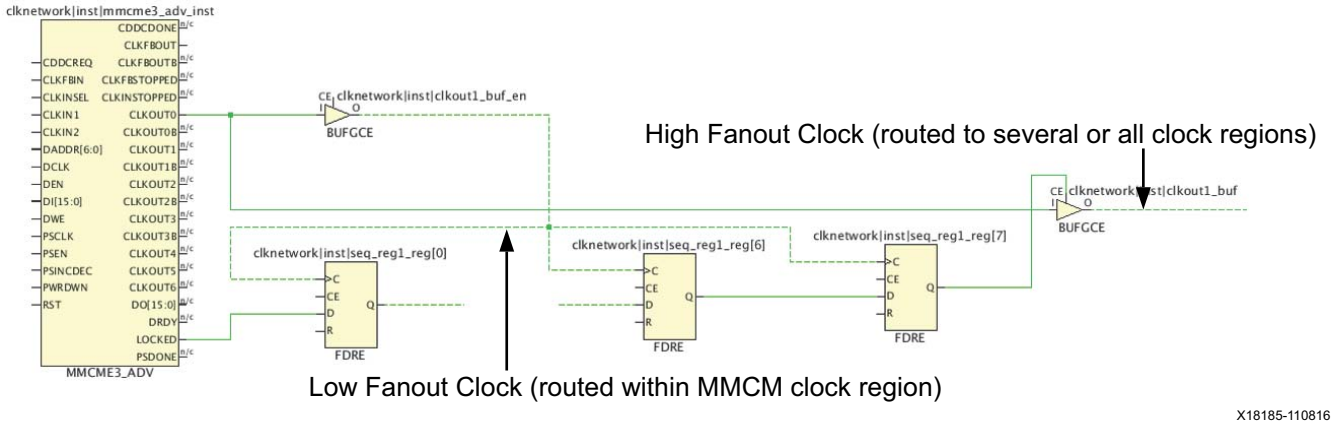



*Figure 3-62:* **UltraScale Device MMCM Safe Clock Startup Example**

**TIP:** *If the MMCM or PLL compensation mode is set to ZHOLD or BUF_IN, all clocks from CLKOUT0 are grouped with the feedback clock and use the same CLOCK_ROOT. If this introduces timing violations on BUFGCE/CE, create a CLOCK_DELAY_GROUP constraint between the high fanout clock and the feedback clock only. Optionally, you can also set a USER_CLOCK_ROOT constraint on the low fanout clock net to the same clock region as the MMCM. For 7 series devices, the second clock buffer is usually not needed for helping timing closure due to the different clocking architecture.*

- Use clock enables, local reset (synchronized), or both, on critical parts of the design, such as a state machine, to ensure that the start-up of those portions of the design are controlled and known.

## Avoiding Local Clocks

Local clocks are clock nets routed with regular fabric resources instead of dedicated global clocking resources. In most cases, the Vivado synthesis and Vivado logic optimization tools insert clock buffers where mandated by the architecture or for clock nets with more than 30 clock loads. Local clocks typically occur when:

- A global clock is divided by a counter implemented with fabric logic
- Clock gating conversion is not able to remove all LUTs from the clock path
- Too many clock buffers are used in 7 series devices

  ***Note:*** UltraScale devices have more clock buffers than 7 series devices, and high utilization of low fanout clock buffers is usually not a concern.

In general, avoid using local clocks. Local clocks introduce several challenges to the implementation tools:

- Unpredictable clock skew, leading to difficult timing closure
- Increase of low to medium fanout nets that are processed with special care by the router, leading to potential routability problems

**TIP:** *If local clocks introduce QoR problems, try floorplanning the clock driver and loads to a small area using a Pblock. Use* `report_clock_utilization` *to identify the location of the local clocks, review the clock placement, and decide on how to reduce their number or impact.*

## Creating an Output Clock

An effective way to forward a clock out of an FPGA device for clocking devices external to the FPGA device, is to use an ODDR component. By tying one of the inputs high and the other low, you can easily create a well controlled clock in terms of phase relationship and duty cycle (for example, by holding D1 to 0 and the D2 pin to 1, you can achieve a 180 degree phase shift). By utilizing the set/reset and clock enable, you also have control over stopping the clock and holding it at a certain polarity for sustained amounts of time.

If further phase control is necessary for an external clock, an MMCM or PLL can be used with external feedback compensation and/or coarse or fine grained, fixed or variable phase compensation. This allows great control over clock phase and propagation times to other devices simplifying external timing requirements from the device.

# Clock Domain Crossing

The clock domain crossing (CDC) circuits in the design directly impact design reliability. You can design your own circuits, but the Vivado Design Suite must recognize the circuit and you must apply the `ASYNC_REG` attributes correctly. Xilinx provides XPMs to ensure correct circuit design, including:

- Driving specific features in `place_design` that reduce mean time between failures (MTBF) on synchronization circuits.
- Ensuring recognition by `report_synchronizer_mtbf`.
- Avoiding `report_cdc` errors and warnings, which typically show up late in the design cycle when iterations are longer.

A CDC circuit is required when crossing between two asynchronous clocks or when attempting to relax timing between two synchronous clocks by adding false path constraints. When using XPMs, you can select a single-bit or a multi-bit bus to cross between the domains.

## Single-Bit CDC

The following figure shows the decisions required when using a single-bit crossing.

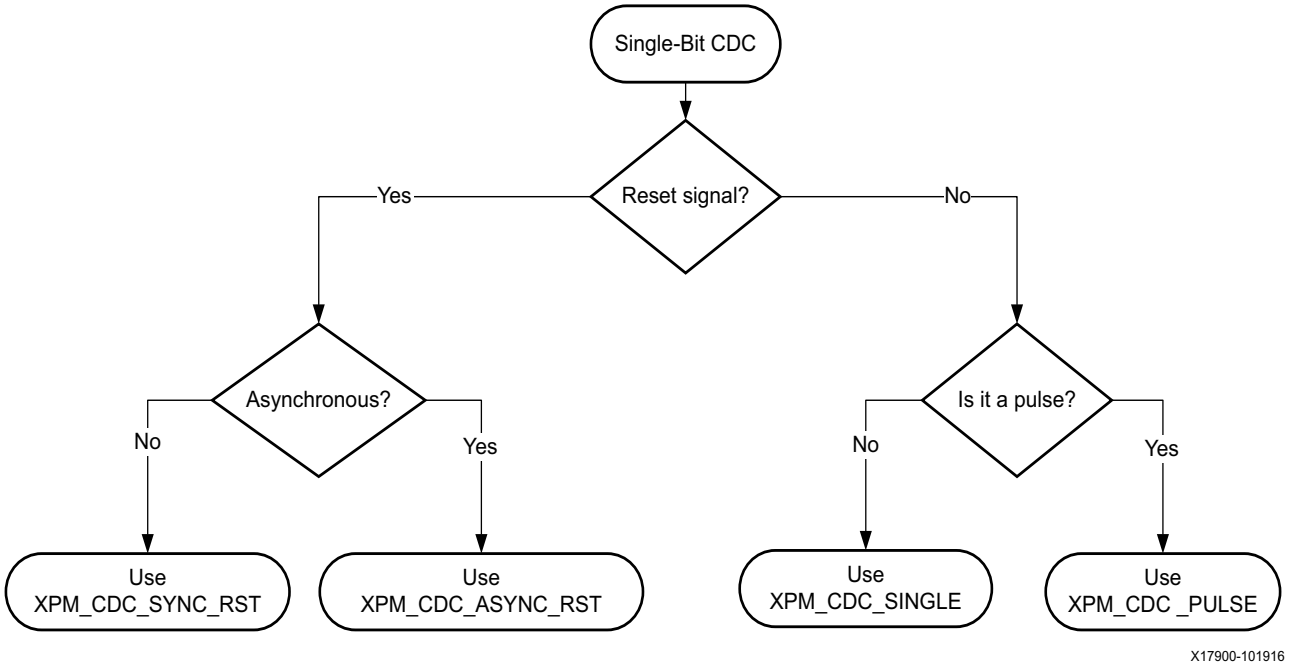



*Figure 3-63:* **Single-Bit CDC Decision Tree**

***Note:*** For more information on the different single-bit synchronizers, see the *Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide* (UG953) [Ref 28] and *UltraScale Architecture Libraries Guide* (UG974) [Ref 29].

## Multi-Bit CDC

The following figure shows the decisions required when using a multi-bit crossing.

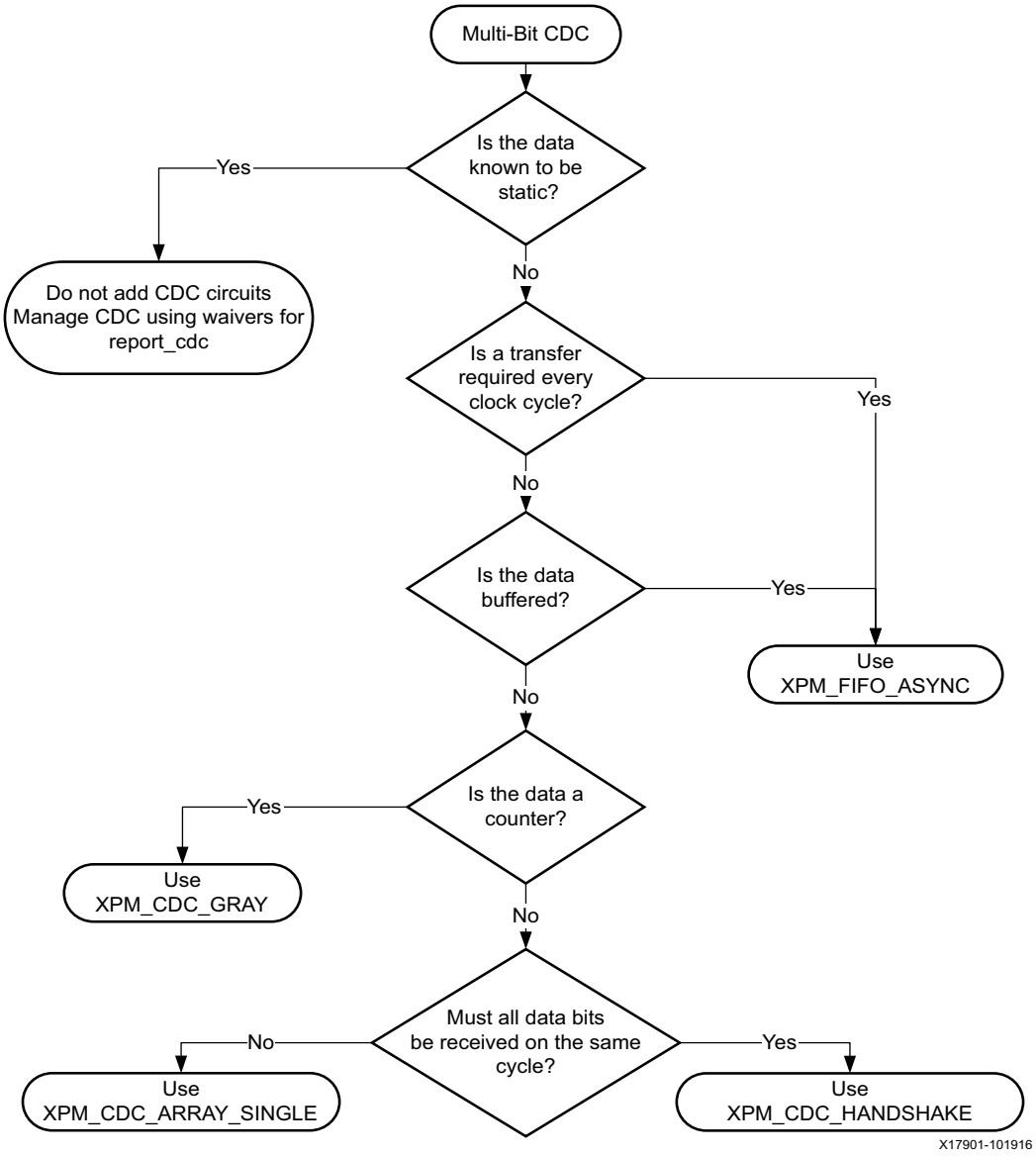



*Figure 3-64:* **Multi-Bit CDC Decision Tree**

***Note:*** For more information on the different multi-bit synchronizers, see the *Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide* (UG953) [Ref 28] and *UltraScale Architecture Libraries Guide* (UG974) [Ref 29].

## Optimizing for MTBF

The total MTBF of a design is a function of:

- Synchronizer MTBF
- Device failure in time (FIT) rate due to single-event upsets (SEUs)

***Note:*** The device FIT rate due to SEUs largely depends on process and device size.

The synchronizer MTBF is design dependent and varies with the following:

- Number of asynchronous CDC points
- Number of synchronizer stages at each crossing point
- Frequency of the destination FF
- Toggle rate of the source

### *Selecting the Correct Value for the DEST_SYNC_FF Parameter*

The `DEST_SYNC_FF` parameter sets the number of metastability protection registers when using an XPM CDC module. The value of this register influences MTBF, design size, and latency at the crossing point. Selecting the correct value of this register is an iterative process that requires the following:

1. Run the design through the Vivado Design Suite implementation flow.
2. Based on your targeted device, do one of the following:
    - For 7 series devices, select the default value for `DEST_SYNC_FF`. This is a conservative approach to meeting typical reliability requirements. For critical designs, conduct further analysis.
    - For UltraScale devices, run the `report_synchonizer_mtbf` command, which reports the MTBF for the entire design. By iterating through the flow as shown in the following figure, you can find a suitable trade-off between MTBF, latency, and resources.

***Note:*** You can also use this iterative process for a user CDC circuit in which the `ASYNC_REG` attribute is correctly applied to all the synchronization registers.

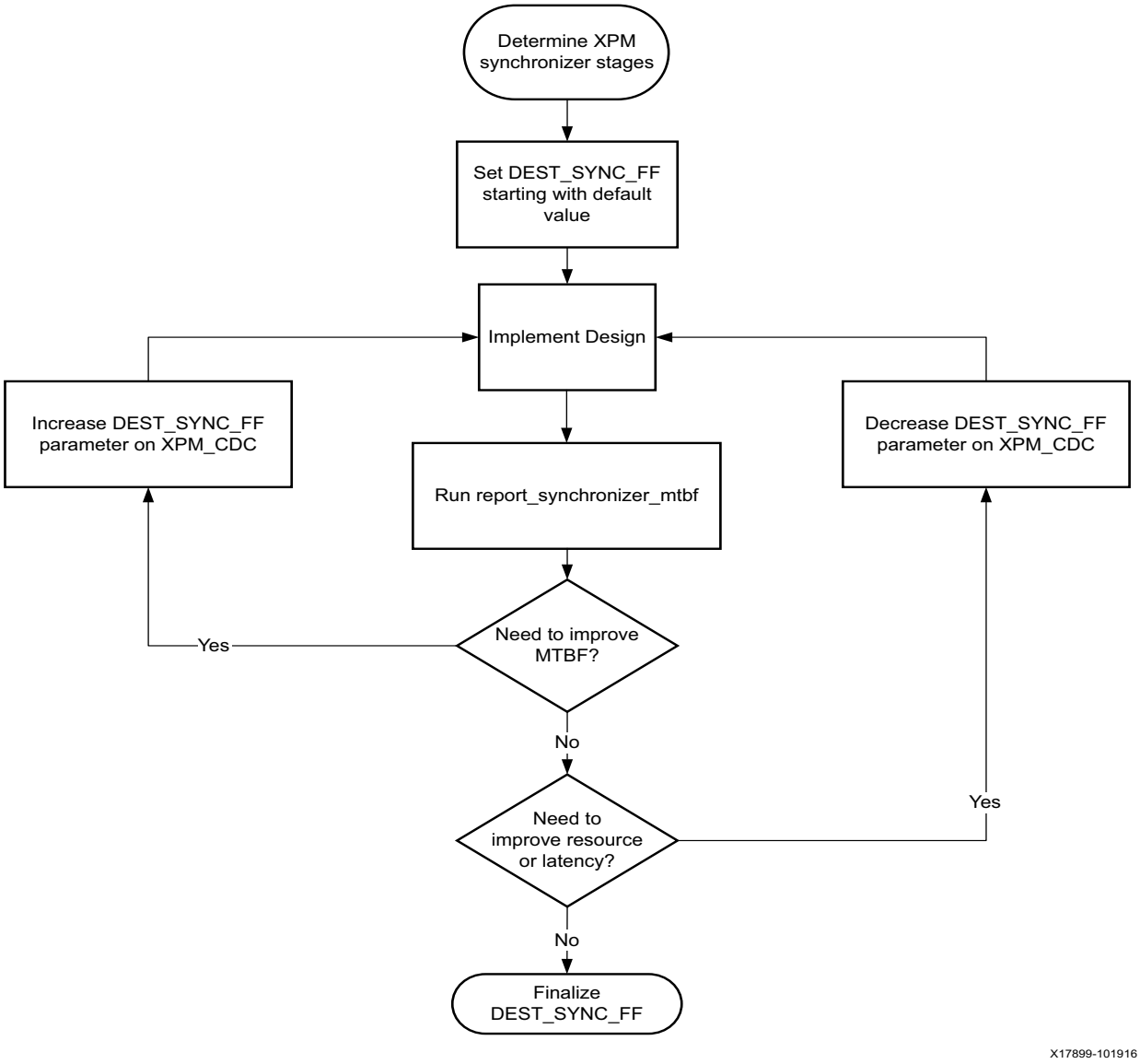



*Figure 3-65:* **Synchronizer MTBF Optimization Flow for UltraScale Device**

## Constraining the Design Correctly

XPM CDCs provide their own `set_max_delay -datapath_only` constraints. XPM CDCs are not compatible with the `set_clock_groups` constraint, which has a higher precedence and will overwrite the constraints in the XPM. For more information, see Defining Clock Groups and CDC Constraints.

# Working With Intellectual Property (IP)

Pre-validated Intellectual Property (IP) cores significantly reduce design and validation efforts, and ensure a large advantage in time-to-market. See the following resources for more information on working with IP:

- *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 10]
- *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 26]
- Vivado Design Suite QuickTake Video: Configuring and Managing Reusable IP in Vivado

## Planning IP Requirements

Planning IP requirements is one of the most important stages of any new project:

- Evaluate the IP options available from Xilinx or third-party partners against required functionality and other design goals. For example:
  - Is custom logic more desirable compared to an available IP core?
  - Does it make sense to package a custom design for reuse in multiple projects in an industry standard format?
- Consider the interfaces that are required such as, memory, network, and peripherals.

**IMPORTANT:** *To ensure that the tools process the IP-specific constraints properly, add the .xci or .xcix IP source files to the project. Do not use the IP-generated output DCP files as project sources when working with IP.*

## AMBA AXI

Xilinx has standardized IP interfaces on the open AMBA® 4 AXI4 interconnect protocol. This standardization eases integration of IP from Xilinx and third-party providers, and maximizes system performance. Xilinx worked with ARM to define the AXI4, AXI4-Lite, and AXI4-Stream specifications for efficient mapping into its FPGA device architectures.

AXI is targeted at high performance, high clock frequency system designs, and is suitable for high-speed interconnects. AXI4-Lite is a light-weight version of AXI4, and is used mostly for accessing control and status registers.

AXI-Stream is used for unidirectional streaming of data from Master to Slave. This is typically used for DSP, Video and Communications applications.

# Vivado Design Suite IP Catalog

The IP Catalog is a single location for Xilinx-supplied IP. In the IP Catalog, you can find IP cores for embedded systems, DSP, communication, interfaces, and more.

From the IP Catalog, you can explore the available IP cores, and view the Product Guide, Change Log, Product Web page, and Answer Records for any IP.

You can access and customize the cores in the IP Catalog through the GUI or Tcl shell. You can also use Tcl scripts to automate the customization of IP cores.

## *Custom IP*

Xilinx uses the industry standard IP-XACT format for delivery of IP, and provides tools (IP Packager) to package custom IP. Accordingly, you can also add your own customized IP to the catalog and create IP repositories that can be shared in a team or across a company. IP from third-party providers can also be added to this catalog, provided it is packaged in IP Packager, even if it is already in the IP-XACT format.

## *Selecting IP from the IP Catalog*

All Xilinx and third-party vendor IP is categorized based on applications such as communications and networking; video and image processing; and automotive and industrial. Use this categorization to browse the catalog to see which IP is available for your area of interest.

A majority of the IP in the IP Catalog is free. However, some high value IP has an associated cost and requires a license. The IP Catalog informs you about whether or not the IP requires purchase, as well as the status of the license. To select an IP from the catalog, consider the following key features, based on your design requirements, and what the specific IP offers:

- Silicon Resources required by this IP (found in the respective IP Product Guide)
- Is this IP supported in the device and speed grade being considered (the selection of the IP often drives the speed grade decision)? If supported, what is the max achievable throughput and Fmax?
- External interface standards, needed for your design to talk to its companion chip on board:
  - Industry-standard interfaces such as Ethernet, PCIe interfaces, etc.
  - Memory interfaces - number of memory interfaces, including their size and performance.
  - Xilinx proprietary interfaces such as Aurora

    ***Note:*** You can also choose to design your own custom interface.
- On-chip bus protocol supported by the IP (Application interface)

- On-chip bus protocol, needed for interaction with the rest of your design. Examples:
  - AXI4
  - AXI4-Lite
  - AXI4-Stream
- If multiple protocols are involved, bridging IP cores might have to be chosen using infrastructure IP from the IP Catalog. Examples:
  - AXI-AHB bridge
  - AXI-AXI interconnect
  - AXI-PCIe bridge
  - AXI-PLB bridge

## Customizing IP

IP can be customized through the GUI or through Tcl scripts.

- Using the Customization GUI
- Using a Tcl Script

### *Using the Customization GUI*

Using the graphical interface is the easiest way to find, research, and customize IP. Each IP is customized with its own set of tabs or pages. Related configuration options are grouped together. An example of a customization window is shown in the following figure. A unique customization of an IP can be created, which is represented in an XCI file. From this, the various output products of an IP can be created.

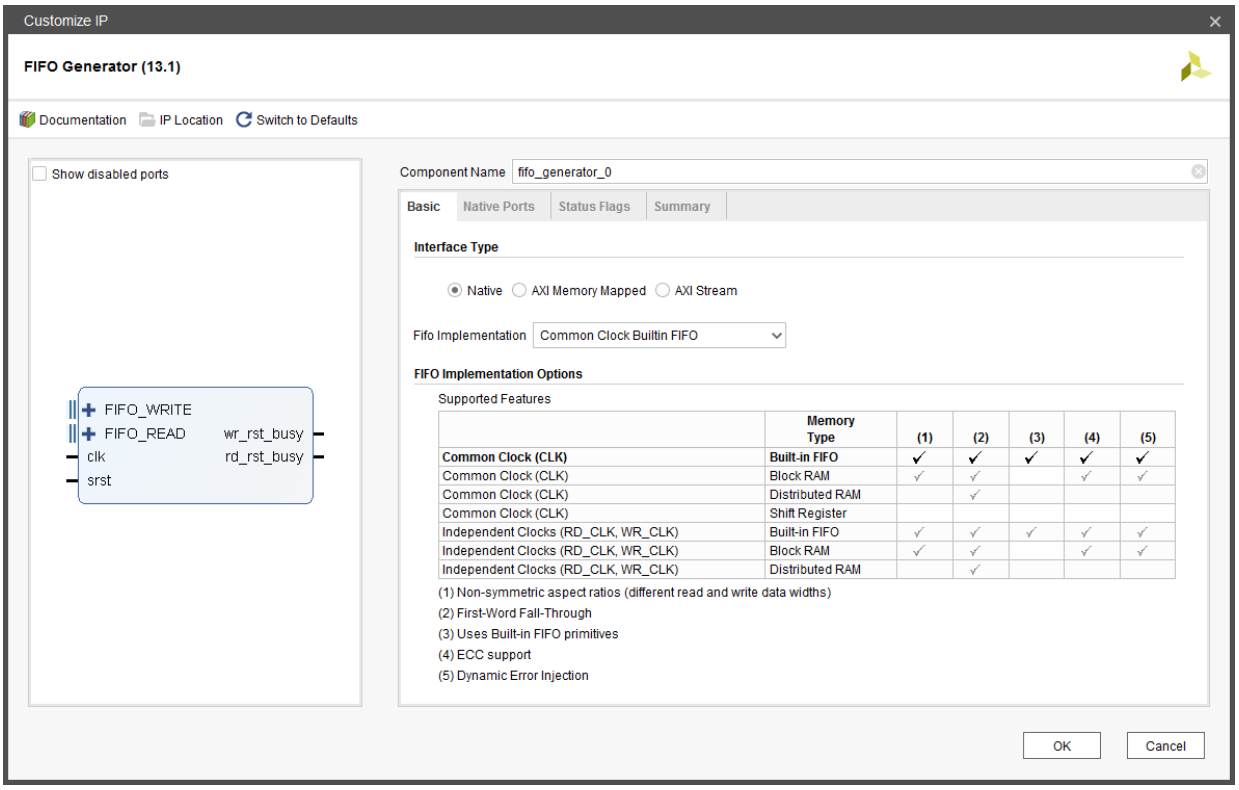



*Figure 3-66:* **Customization Window for an IP**

### *Using a Tcl Script*

Almost every GUI action results in the issuance of a Tcl command. The creation of an IP including the setting of all the customization options can be performed in a Tcl script without user interaction.

You would need to know the names of the configuration options, and the values to which they can be set. Typically, you first perform the customization through the GUI, and then create the script from that. Once you see the resulting Tcl script, you can easily modify the script for your needs, such as changing data sizes.

Tcl script based IP creation is useful for automation, for example working with version control system. For information about source management and revision control, see this link in the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 6].

## IP Versions and Revision Control

When IP is customized, the tool creates an XCI file containing all the selected parameterization values. Each Vivado IDE version supports only one version of an IP. Xilinx recommends that you use this latest IP version. If you use an older IP version, you must save all the output products for the older version. For information about source management

and revision control, see this link in the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 6].

**IMPORTANT:** *For memory IP in 7 series devices, a PRJ file is created in addition to the XCI file. When using revision control with 7 series memory IP, keep the PRJ file in the same directory as the XCI file.*

# Working with Constraints

## Organizing the Design Constraints

Design constraints define the requirements that must be met by the compilation flow in order for the design to be functional in hardware. For more complex designs, they also define guidance for the tools to help with convergence and closure. Not all constraints are used by all steps in the compilation flow. For example, physical constraints are used only during the implementation steps (that is, by the placer and the router).

Because synthesis and implementation algorithms are timing-driven, creating proper timing constraints is essential. Over-constraining or under-constraining your design makes timing closure difficult. You must use reasonable constraints that correspond to your application requirements. For more information on constraints, see the following resources:

- *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21]
- Applying Design Constraints video tutorials available from the Vivado Design Suite Video Tutorials page on the Xilinx website

The constraints are usually organized by category, by design module, or both, in one or many files. Regardless of how you organize them, you must understand their overall dependencies and review their final sequence once loaded in memory. For example, because timing clocks must be defined before they can be used by any other constraints, you must make sure that their definition is located at the beginning of your constraint file, in the first set of constraint files loaded in memory, or both.

### *Recommended Constraint Files*

There are many ways to organize your constraints depending on the size and complexity of your project. Following are a few suggestions.

**Simple Design**

For a simple design with a small team of designers:

- 1 file for all constraints
- 1 file for physical + 1 file for timing
- 1 file for physical + 1 file for timing (synthesis) + 1 file for timing (implementation)

**Complex Design**

For a complex design with IP cores or several designer teams:

- 1 file for top-level timing + 1 file for top-level physical + 1 file per IP/major block

### *Validating the Read Sequence*

Once you have settled on the organization of your project constraint files, you must validate the read sequence of the files depending on the content of the files. In Project Mode, you can modify the constraint file sequence in the Vivado IDE or by using the `reorder_files` Tcl command. In Non-Project Mode, the sequence is directly defined by the `read_xdc` (for XDC files) and `source` (for constraints generated by Tcl scripts) commands in your compilation flow Tcl script.

### *Recommended Constraints Sequence*

The constraints language (XDC) is based on Tcl syntax and interpretation rules. Like Tcl, XDC is a sequential language:

- Variables must be defined before they can be used. Similarly, timing clocks must be defined before they can be used in other constraints.
- For equivalent constraints that cover the same paths and have the same precedence, the last one applies.

When considering the priority rules above, the timing constraints should overall use the following sequence:

```
## Timing Assertions Section
# Primary clocks
# Virtual clocks
# Generated clocks
# Delay for external MMCM/PLL feedback loop
# Clock Uncertainty and Jitter
# Input and output delay constraints
# Clock Groups and Clock False Paths

## Timing Exceptions Section
# False Paths
# Max Delay / Min Delay
# Multicycle Paths
# Case Analysis
# Disable Timing
```

When multiple XDC files are used, you must pay particular attention to the clock definitions and validate that the dependencies are ordered correctly.

The physical constraints can be located anywhere in any constraint file.

### *Creating Synthesis Constraints*

Synthesis takes the RTL description of the design and transforms it into an optimized technology mapped netlist by using timing-driven algorithms. The quality of the results is affected by the quality of the RTL code and the constraints provided. At this point of the compilation flow, the net delay modeling is approximate and does not reflect placement constraints or complex effects such as congestion. The main objective is to obtain a netlist which meets timing, or fails by a small amount, with realistic and simple constraints.

The synthesis engine accepts all XDC commands, but only some have a real effect:

- Timing constraints related to setup/recovery analysis influence the QoR:
  - `create_clock` / `create_generated_clock`
  - `set_input_delay` / `set_output_delay`
  - `set_clock_groups` / `set_false_path` / `set_max_delay` / `set_multicycle_path`
- Timing constraints related to hold and removal analysis are ignored during synthesis:
  - `set_min_delay` / `set_false_path -hold` / `set_multicycle_path -hold`

- RTL attributes forces decisions made by the mapping and optimization algorithms. Following are a few examples:
  - DONT_TOUCH / KEEP / KEEP_HIERARCHY / MARK_DEBUG
  - MAX_FANOUT
  - RAM_STYLE / ROM_STYLE / USE_DSP48 / SHREG_EXTRACT
  - FULL_CASE / PARALLEL_CASE (Verilog RTL only)

  ***Note:*** The same attribute can also be set as a property from an XDC file. Using XDC-based constraints is convenient for influencing the synthesis results only in some cases without changing the RTL.
- Physical constraints are ignored (LOC, BEL, Pblocks)

Synthesis constraints must use names from the elaborated netlist, preferably ports and sequential cells. During elaboration, some RTL signals can disappear and it is not possible to attach XDC constraints to them. In addition, due to the various optimizations after elaboration, nets or logical cells are merged into the various technology primitives such as LUTs or DSP blocks. To know the elaborated names of your design objects, click **Open Elaborated Design** in the Flow Navigator and browse to the hierarchy of interest.

Some registers are absorbed into RAM blocks and some levels of the hierarchy can disappear to allow cross-boundary optimizations.

Any elaborated netlist object or level of hierarchy can be preserved by using a DONT_TOUCH, KEEP, KEEP_HIERARCHY, or MARK_DEBUG constraint, at the risk of degrading timing or area QoR.

Finally, some constraints can conflict and cannot be respected by synthesis. For example, if a MAX_FANOUT attribute is set on a net that crosses multiple levels of hierarchy, and some hierarchies are preserved with DONT_TOUCH, the fanout optimization will be limited or fully prevented.

**IMPORTANT:** *Unlike during implementation, RTL netlist objects that are used for defining timing constraints can be optimized away by synthesis to allow better QoR. This is usually not a problem as long as the constraints are updated and validated for implementation. But if needed, you can preserve any object by using the DONT_TOUCH constraint so that the constraints will apply during both synthesis and implementation.*

Once synthesis has completed, Xilinx recommends that you review the timing and utilization reports to validate that the netlist quality meets the application requirements and can be used for implementation.

### *Creating Implementation Constraints*

The implementation constraints must accurately reflect the requirements of the final application. Physical constraints such as I/O location and I/O standard are dictated by the board design, including the board trace delays, as well as the design internal requirements derived from the overall system requirements. Before you proceed to implementation, Xilinx highly recommends that you validate the correctness and accuracy of all your constraints. An improper constraint will likely contribute to degradation of the implementation QoR and can lower the confidence level in the timing signoff quality.

In many cases, the same constraints can be used during synthesis and implementation. However, because the design objects can disappear or have their name changed during synthesis, you must verify that all synthesis constraints still apply properly with the implementation netlist. If this is not the case, you must create an additional XDC file containing the constraints that are valid for implementation only.

### *Creating Block-Level Constraints*

When working on a multi-team project, it is convenient to create individual constraint files for each major block of the top-level design. Each of these blocks is usually developed and validated separately before the final integration into one or many top-level designs.

The block-level constraints must be developed independently from the top-level constraints, and must be as generic as possible so that they can be used in various contexts. In addition, these constraints must not affect any logic that is beyond the block boundaries.

When implementing a sub-block it is desirable to have the full clocking network included in timing analysis to ensure accurate skew and clock domain crossing analysis. This might require an HDL wrapper containing the clocking components and an additional constraint file to replicate top level clocking constraints. It is used only in the timing validation of the sub-module.

For more information on constraints scoping as well as rules, guidelines, and mechanisms for loading the block-level constraints into the top-level design, see this link in the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 18].

## Defining Timing Constraints in Four Steps

The process of defining good constraints is broken into the four major steps shown in the following figure. The steps follow the timing constraints precedence and dependency rules, as well as the logical way of providing information to the timing engine to perform the analysis.

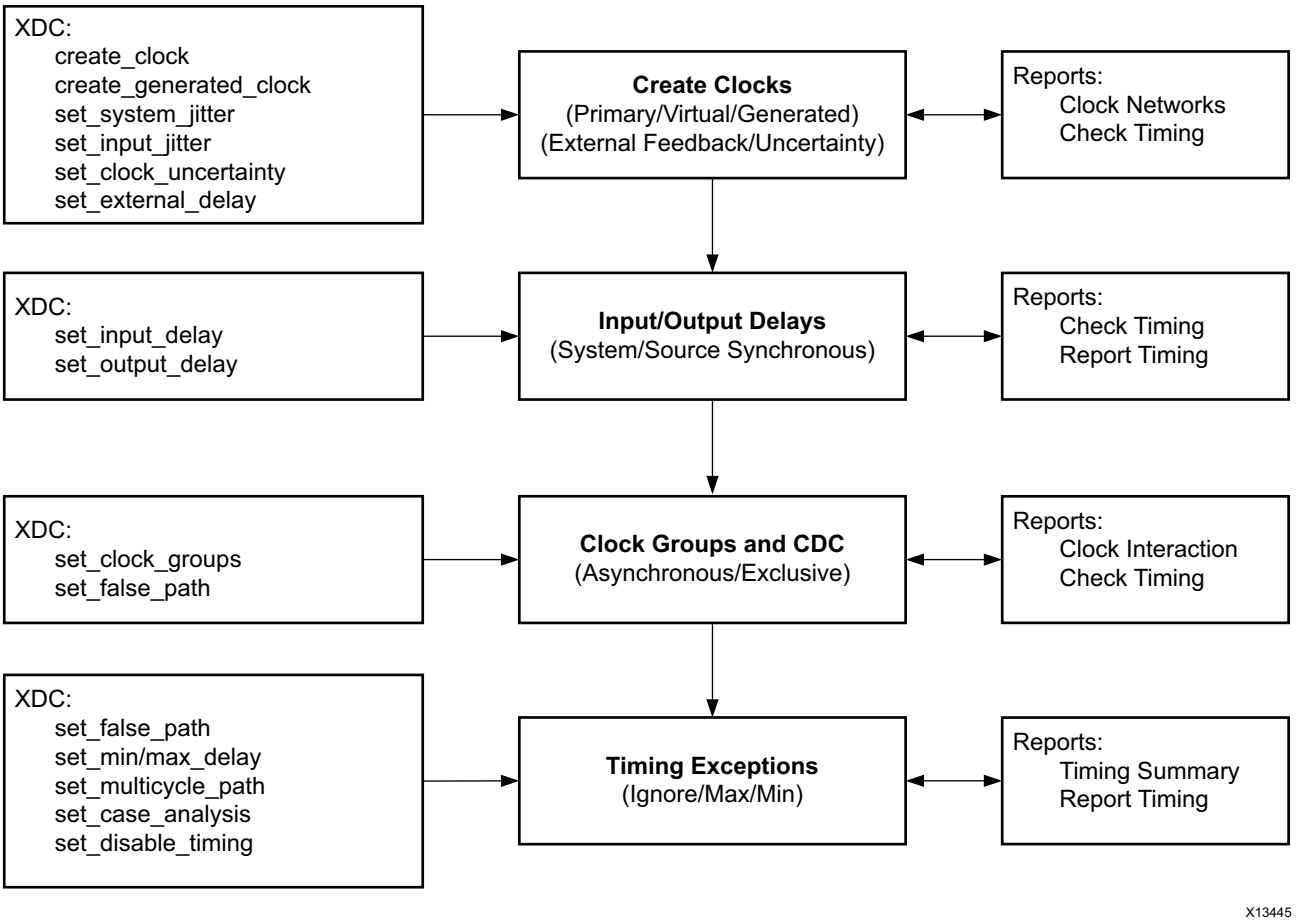



*Figure 3-67:* **Steps for Developing Timing Constraints**

- The first two steps refer to the timing assertions where the default timing path requirements are derived from the clock waveforms and I/O delay constraints.
- During the third step, relationships between the asynchronous/exclusive clock domains that share at least one logical path are reviewed. Based on the nature of the relationships, clock groups or false path constraints are entered to ignore the timing analysis on these paths.
- The last step corresponds to the timing exceptions, where the designer can decide to alter the default timing path requirements by ignoring, relaxing or tightening them with specific constraints.

Constraints creation is associated with constraints identification and constraints validation tasks that are only possible with the various reports generated by the timing engine. The timing engine only works with a fully mapped netlist, for example, after synthesis. While it is possible to enter constraints with an elaborated netlist, it is recommended to create the first set of constraints with the post-synthesis netlist so that analysis and reports on the constraints can be performed interactively.

When creating timing constraints for a new design or completing existing constraints, Xilinx recommends using the Timing Constraints Wizard to quickly identify missing constraints for the first three steps in Figure 3-67. The Timing Constraints Wizard follows the methodology described in this section to ensure the design constraints are safe and reliable for proper timing closure. You can find more information on the Timing Constraints Wizard in *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 18].

The following sections describe in detail the four steps described above:

- Defining Clock Constraints
- Constraining Input and Output Ports
- Defining Clock Groups and CDC Constraints
- Specifying Timing Exceptions

Refer to each section for a detailed methodology and use case when you are at the appropriate step in the constraint creation process.

## Defining Clock Constraints

Clocks must be defined first so that they can be used by other constraints. The first step of the timing constraint creation flow is to identify where the clocks must be defined and whether they must be defined as *primary clock* or a *generated clock*.

**IMPORTANT:** *When defining a clock with a specific name (`-name` option), you must verify that the clock name is not already used by another clock constraint or an existing auto-generated clock. The Vivado Design Suite timing engine issues a message when a clock name is used in several clock constraints to warn you that the first clock definition is overridden. When the same clock name is used twice, the first clock definition is lost as well as all constraints referring to that name and entered between the two clock definitions. Xilinx recommends that you avoid overriding clock definitions unless no other constraints are impacted and all timing paths remain constrained.*

### *Identifying Clock Sources*

The unconstrained clock sources can be identified in the design by the following two reports:

- Clock Networks Report
- Check Timing Report

#### Clock Networks Report

Both constrained and unconstrained clock source points are listed in two separate categories. For each unconstrained source point, you must identify whether a primary clock or a generated clock must be defined.

```
% report_clock_networks

Unconstrained Clocks
Clock sysClk (endpoints: 15633 clock, 0 nonclock)
Port sysClk

Clock TXOUTCLK (endpoints: 148 clock, 0 nonclock)
GTXE2_CHANNEL/TXOUTCLK
(mgtEngine/ROCKETIO_WRAPPER_TILE_i/gt0_ROCKETIO_WRAPPER_TILE_i/gtxe2_i)

Clock Q (endpoints: 8 clock, 0 nonclock)
FDRE/Q (usbClkDiv2_reg)
```

#### Check Timing Report

The `no_clock` check reports the groups of active leaf clock pins with no clock definition. Each group is associated with a clock source point where a clock must be defined in order to clear the issue.

```
% check_timing -override_defaults no_clock

1. checking no_clock
--------------------
 There are 15633 register/latch pins with no clock driven by root clock pin: sysClk
(HIGH)

There are 148 register/latch pins with no clock driven by root clock pin:
mgtEngine/ROCKETIO_WRAPPER_TILE_i/gt0_ROCKETIO_WRAPPER_TILE_i/gtxe2_i/TXOUTCLK
(HIGH)

There are 8 register/latch pins with no clock driven by root clock pin:
usbClkDiv2_reg/C (HIGH)
```

With `check_timing`, the same clock source pin or port can appear in several groups depending on the topology of the entire clock tree. In such case, creating a clock on the recommended source pin or port will resolve the missing clock definition for all the associated groups.

For more information, see Checking That Your Design is Properly Constrained in Chapter 5.

### *Creating Primary Clocks*

A primary clock is a clock that defines a timing reference for your design and that is used by the timing engine to derive the timing path requirements and the phase relationship with other clocks. Their insertion delay is calculated from the clock source point (driver pin/port where the clock is defined) to the clock pins of the sequential cells to which it fans out.

For this reason, it is important to define the primary clocks on objects that correspond to the boundary of the design, so that their delay, and indirectly their skew, can be accurately computed.

Typical primary clock roots are:

- Input Ports
- Gigabit Transceiver Output Pins in 7 Series Devices
- Certain Hardware Primitive Output Pins

### Input Ports

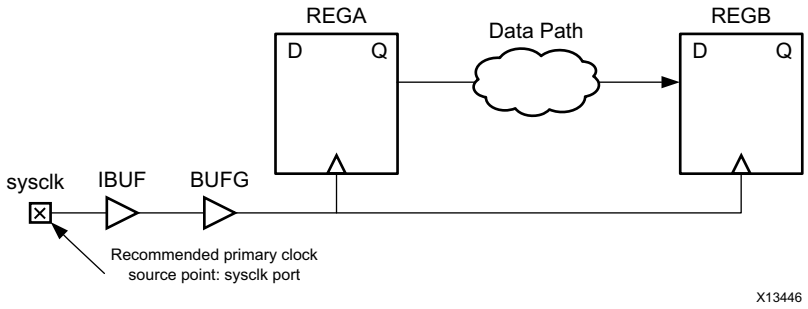



*Figure 3-68:* **create_clock for Input Ports**

Constraint example:

```
create_clock -name SysClk -period 10 -waveform {0 5} [get_ports sysclk]
```

In this example, the waveform is defined to have a 50% duty cycle. The `-waveform` argument is shown above to illustrate its usage and is only necessary to define a clock with a duty cycle other than 50%. For a differential clock input buffer, the primary clock only needs to be defined on the P-side of the pair.

### Gigabit Transceiver Output Pins in 7 Series Devices

Gigabit transceiver output pin, for example, a recovered clock:

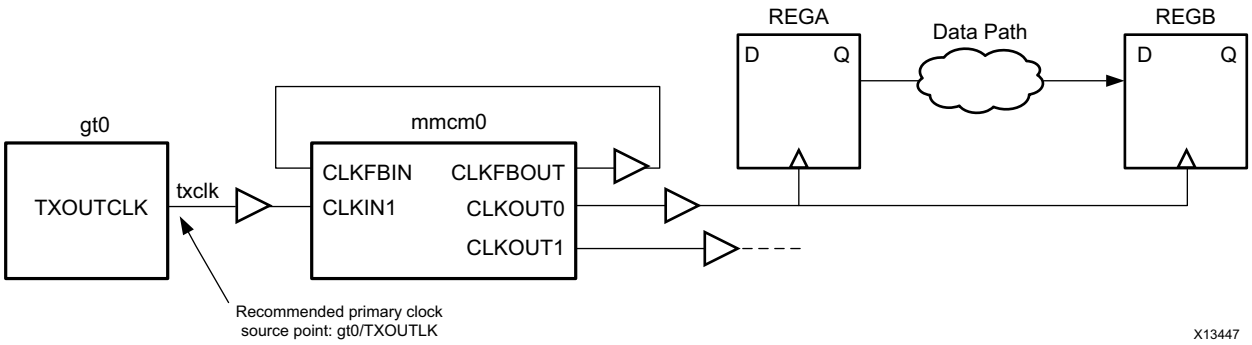



*Figure 3-69:* **create_clock on a Primitive Pin**

Constraint example:

```
create_clock -name txclk -period 6.667 [get_pins gt0/TXOUTCLK]
```

**RECOMMENDED:** *For designs that target 7 series devices, Xilinx recommends also defining the GT incoming clocks, because the Vivado tools calculate the expected clocks on the GT output pins and compare these clocks with the user created clocks. If the clocks differ or if the incoming clocks to the GT are missing, the tools issue a methodology check warning.*

***Note:*** For designs that target UltraScale devices, Xilinx does not recommend defining a primary clock on the output of GTs, because GT clocks are automatically derived when the related board input clocks are defined.

### Certain Hardware Primitive Output Pins

The output pin of certain hardware primitives, for example, BSCANE2, which does not have a timing arc from an input pin of the same primitive.

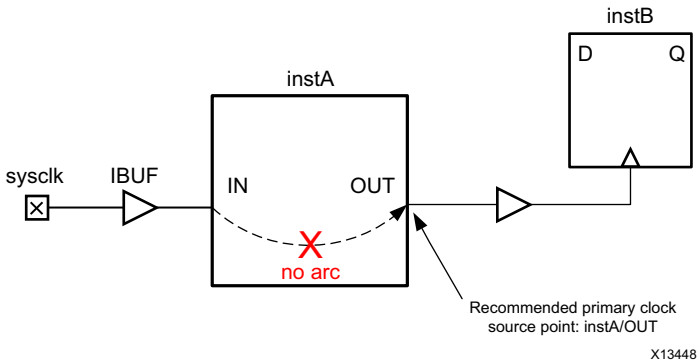



*Figure 3-70:* **Clock Path Broken Due to a Missing Timing Arc**

**IMPORTANT:** *No primary clock should be defined in the transitive fanout of another primary clock because this situation does not correspond to any hardware reality. It will also prevent proper timing analysis by preventing the complete clock insertion delay calculation. Any time this situation occurs, the constraints must be revisited and corrected.*

The following figure shows an example in which the clock `clk1` is defined in the transitive fanout of the clock `clk0`. `clk1` overrides `clk0` starting at the output of BUFG1, where it is defined. Therefore, the timing analysis between REGA and REGB will not be accurate because of the invalid skew computation between `clk0` and `clk1`.

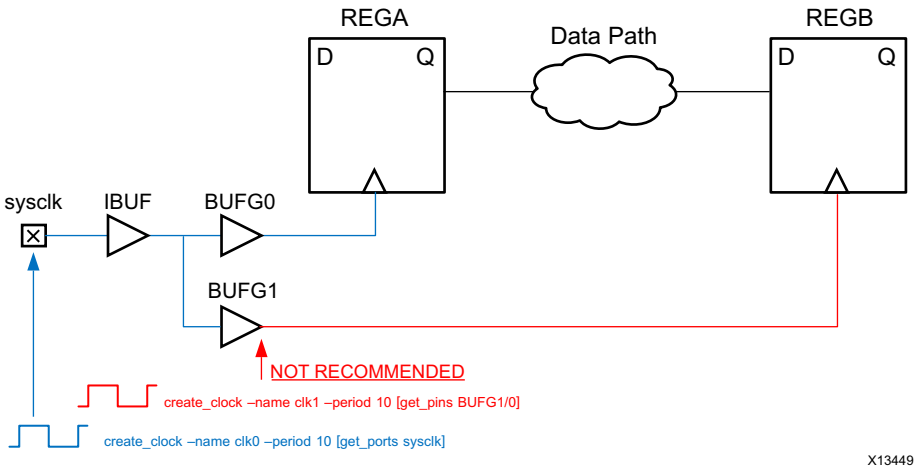



*Figure 3-71:* **create_clock in the Fanout of Another Clock is Not Recommended**

## *Creating Generated Clocks*

A generated clock is a clock derived from another existing clock called the master clock. It usually describes a waveform transformation performed on the master clock by a logic block. Because the generated clock definition depends on the master clock characteristics, the master clock must be defined first. For explicitly defining a generated clock, the `create_generated_clock` command must be used.

### Auto-Derived Clocks

Most generated clocks are automatically derived by the Vivado Design Suite timing engine which recognizes the Clock Modifying Blocks (CMB) and the transformation they perform on the master clocks.

In the Xilinx 7 series device family, the CMBs are:

- MMCM*/ PLL*
- BUFR
- PHASER*

In the Xilinx UltraScale device family, the CMBs are:

- MMCM* / PLL*
- BUFG_GT / BUFGCE_DIV
- GT*_COMMON / GT*_CHANNEL / IBUFDS_GTE3
- BITSLICE_CONTROL / RX*_BITSLICE
- ISERDESE3

For any other combinatorial cell located on the clock tree, the timing clocks propagate through them and do not need to be redefined at their output, unless the waveform is transformed by the cell. In general, you must rely on the auto-derivation mechanism as much as possible as it provides the safest way to define the generated clocks that correspond to the actual hardware behavior.

If the auto-derived clock name chosen by the Vivado Design Suite timing engine does not seem appropriate, you can force your own name by using the `create_generated_clock` command without specifying the waveform transformation. This constraint should be located right after the constraint that defines the master clock in the constraint file. For example, if the default name of a clock generated by a MMCM instance is net0, you can add the following constraint to force your own name (`fftClk` in the given example):

```
create_generated_clock -name fftClk [get_pins mmcm_i/CLKOUT0]
```

To avoid any ambiguity, the constraint must be attached to the source pin of the clock. For more information, see *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 18].

### User-Defined Generated Clocks

Once all the primary clocks have been defined, you can use the Clock Networks or Check Timing (`no_clock`) reports to identify the clock tree portions that do not have a timing clock and define the generated clocks accordingly.

It is sometimes difficult to understand the transformation performed by a cone of logic on the master clock. In this case, you must adopt the most conservative constraint. For example, the source pin is a sequential cell output. The master clock is at least divided by two, so the proper constraint should be, for example:

```
create_generated_clock -name clkDiv2 -divide_by 2 \
-source [get_pins fd/C] [get_pins fd/Q]
```

Finally, if the design contains latches, the latch gate pins also need to be reached by a timing clock and will be reported by Check Timing (`no_clock`) if the constraint is missing. You can follow the examples above to define these clocks.

### Path Between Master and Generated Clocks

Unlike primary clocks, generated clocks must be defined in the transitive fanout of their master clock, so that the timing engine can accurately compute their insertion delay. Failure to follow this rule will result in improper timing analysis and most likely in invalid slack computation. For example, in the following figure `gen_clk_reg/Q` is being used as a clock for the next flop (`q_reg`), and it is also in the fanout cone of the primary clock `c1`. Hence `gen_clk_reg/Q` should have a `create_generated_clock` on it, rather than a `create_clock`.

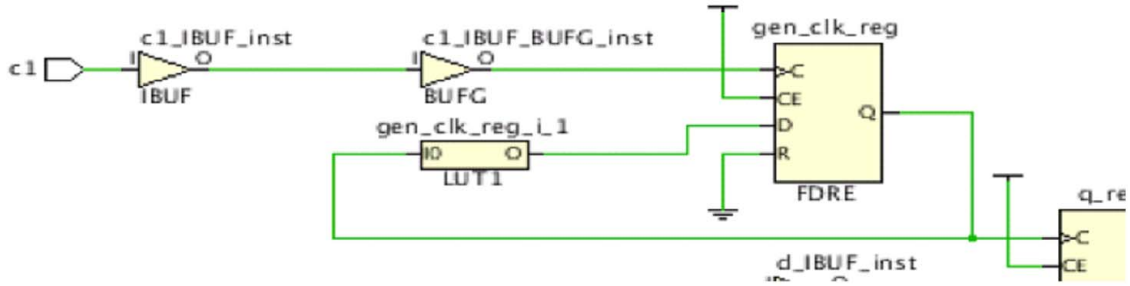



*Figure 3-72:* **Generated Clock in the Fanout Of Master Clock**

```
create_generated_clock -name GC1 -source [get_pins gen_clk_reg/C] -divide_by 2
[get_pins gen_clk_reg/Q]
```

### *Verifying Clocks Definition and Coverage*

Once all design clocks are defined and applied in memory, you can verify the waveform of each clock, the relationship between master and generated clocks by using the `report_clocks` command:

```
Clock       Period    Waveform             Attributes  Sources
sysClk      10.00000  {0.00000 5.00000}    P           {sysClk}
clkfbout    10.00000  {0.00000 5.00000}    P,G         {clkgen/mmcm_adv_inst/CLKFBOUT}
cpuClk      20.00000  {0.00000 10.00000}   P,G         {clkgen/mmcm_adv_inst/CLKOUT0}
…
=================================================
Generated Clocks
=================================================

Generated Clock    : cpuClk
Master Source      : clkgen/mmcm_adv_inst/CLKIN1
Master Clock       : sysClk
Edges              : {1 2 3}
Edge Shifts        : {0.000 5.000 10.000}
Generated Sources : {clkgen/mmcm_adv_inst/CLKOUT0}
```

You can also verify that all internal timing paths are covered by at least one clock. The check_timing report provides two checks for that purpose:

- **no_clock**

  Reports any active clock pin that is not reached by a defined clock.

- **unconstrained_internal_endpoint**

  Reports all the data input pins of sequential cells that have a timing check relative to a clock but the clock has not been defined.

If both checks return zero, the timing analysis coverage will be high.

Alternatively, you can run the XDC and Timing Methodology checks to verify that all clocks are defined on recommended netlist objects without introducing any constraint conflict or inaccurate timing analysis scenario.

If you are using a Vivado Design Suite version prior to 2016.3, use the following command to run these checks:

```
report_drc -checks [get_drc_checks {XDC* TIMING-*}]
```

If you are using Vivado Design Suite version 2016.3 or later, use the following command to run these checks:

```
report_methodology -checks [get_methodology_checks {TIMING-* XDC*}]
```

For more information, see Running Report Methodology in Chapter 4.

### *Adjusting Clock Characteristics*

After defining the clocks and their waveform, the next step is to enter any information related to noise or uncertainty modeling. The XDC language differentiates uncertainty related to jitter and phase error from the one related to skew and delay modeling.

- Jitter
- Additional Uncertainty
- Clock Latency at the Source
- MMCM or PLL External Feedback Loop Delay

### Jitter

For jitter, it is best to use the default values used by the Vivado Design Suite. You can modify the default computation as follows:

- If a primary clock enters the device with a random jitter greater than zero, use the `set_input_jitter` command to specify the jitter value.
- To adjust the global jitter if the device power supply is noisy, use `set_system_jitter`. Xilinx does *not* recommend increasing the default system jitter value.

For generated clocks, the jitter is derived from the master clock and the characteristics of the clock modifying block. The user does not need to adjust these numbers.

### Additional Uncertainty

When you need to add extra margin on the timing paths of a clock or between two clocks, you must use the `set_clock_uncertainty` command. This is also the best and safest way to over-constrain a portion of a design without modifying the actual clock edges and the overall clocks relationships. The clock uncertainty defined by the user is additive to the jitter computed by the Vivado tools, and can be specified separately for setup and hold analyses.

For example, the margin on all intra-clock paths of the design clock clk0 needs to be tightened by 500ps to make the design more robust to noise for both setup and hold:

```
set_clock_uncertainty -from clk0 -to clk0 0.500
```

If you specify additional uncertainty between two clocks, the constraint must be applied in both directions (assuming data flows in both directions). The example below shows how to increase the uncertainty by 250ps between `clk0` and `clk1` for setup only:

```
set_clock_uncertainty -from clk0 -to clk1 0.250 -setup
set_clock_uncertainty -from clk1 -to clk0 0.250 -setup
```

### Clock Latency at the Source

It is possible to model the latency of a clock at its source by using the `set_clock_latency` command with the `-source` option. This is useful in two cases:

- To specify the clock delay propagation outside the device independently from the input and output delay constraints.
- To model the internal propagation latency of a clock used by a block during out-of-context compilation. In such a compilation flow, the complete clock tree is not described, so the variation between min and max operating conditions outside the block cannot be automatically computed and must be manually modeled.

This constraint should only be used by advanced users as it is usually difficult to provide valid latency values.

#### MMCM or PLL External Feedback Loop Delay

When the MMCM or PLL feedback loop is connected for compensating a board delay instead of an internal clock insertion delay, you must specify the delay outside the FPGA device for both best and worst delay cases by using the `set_external_delay` command. Failure to specify this delay will make I/O timing analysis associated with the MMCM or PLL irrelevant and can potentially lead to an impossible timing closure situation. Also, when using external compensation, you must adjust the input and output delay constraint values accordingly instead of just considering the clock trace delay on the board like in normal cases.

## Constraining Input and Output Ports

In addition to specifying the location and I/O standard for each port of the design, input and output delay constraints must be specified to describe the timing of external paths to/from the interface of the FPGA device. These delays are defined relative to a clock that is usually also generated on the board and enters the FPGA device. In some cases, the delays must be defined related to a virtual clock when the I/O path is related to a clock that has a waveform different from the board clock.

### *System Level Perspective*

The I/O paths are modeled like any other `reg-to-reg` paths by the Vivado Design Suite timing engine, except that part of the path is located outside the FPGA device and needs to be described by the user. When analyzing internal paths, minimum and maximum delays are considered for both setup and hold analysis. This is also true for I/O paths. For this reason, it is important to describe both min and max delay conditions. The I/O timing paths are analyzed as single-cycle paths by default, which means:

- For max delay analysis (setup), the data is captured one clock cycle after the launch edge for single data rate interface, and half clock cycle after the launch edge for a double data rate interface.
- For min delay analysis (hold), the data is launched and captured by the same clock edge.

If the relationship between the clock and I/O data must be timed differently, like for example in a source synchronous interface, different I/O delays and additional timing exceptions must be specified. This corresponds to an advanced I/O timing constraints scenario.

### *Defining Input Delays*

The input delay is defined relative to a clock at the interface of the device. Unless `set_clock_latency` has been specified on the source pin of the reference clock, the input delay corresponds to the absolute time from the launch edge, through the clock trace, the external device and the data trace. If clock latency has already been specified separately, you can ignore the clock trace delay.

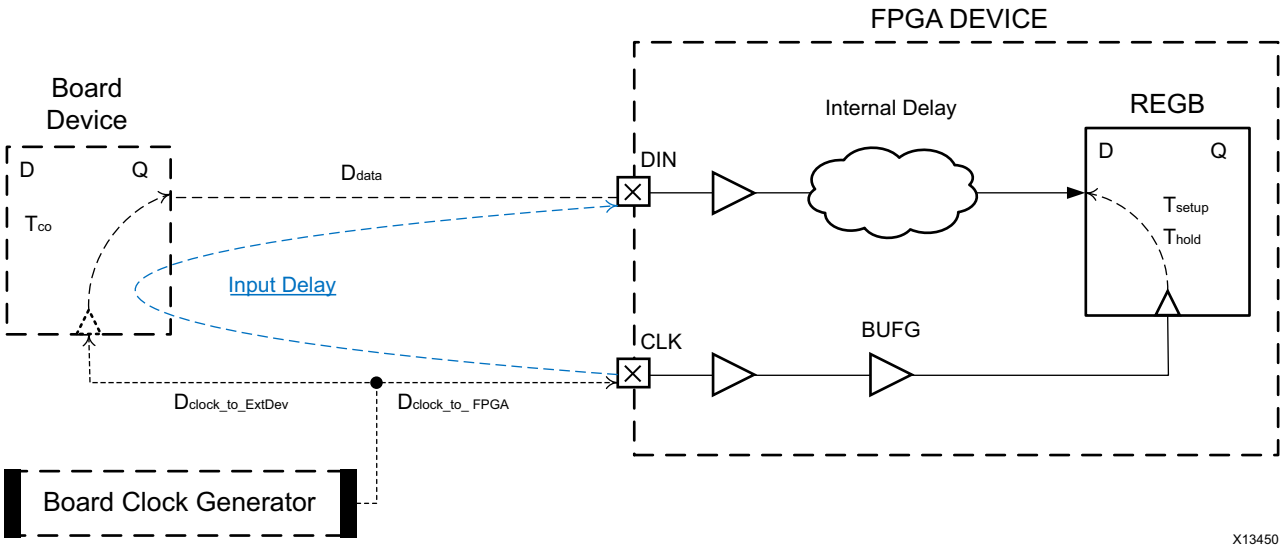



*Figure 3-73:* **Input Delay Computation**

The input delay values for the both types of analysis are:

```
Input Delay(max) = Tco(max) + Ddata(max) + Dclock_to_ExtDev(max) - Dclock_to_FPGA(min)
Input Delay(min) = Tco(min) + Ddata(min) + Dclock_to_ExtDev(min) - Dclock_to_FPGA(max)
```

The following figure shows a simple example of input delay constraints for both setup (max) and hold (min) analysis, assuming the sysClk clock has already been defined on the CLK port:

```
set_input_delay -max -clock sysClk 5.4 [get_ports DIN]
set_input_delay -min -clock sysClk 2.1 [get_ports DIN]
```

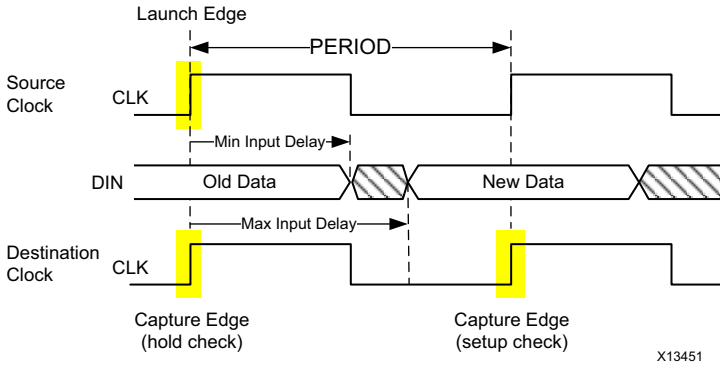



*Figure 3-74:* **Interpreting Min and Max Input Delays**

A negative input delay means that the data arrives at the interface of the device before the launch clock edge.

## *Defining Output Delays*

Output delays are similar to input delays, except that they refer to the output path minimum and maximum time outside the FPGA device in order to be functional under all conditions.

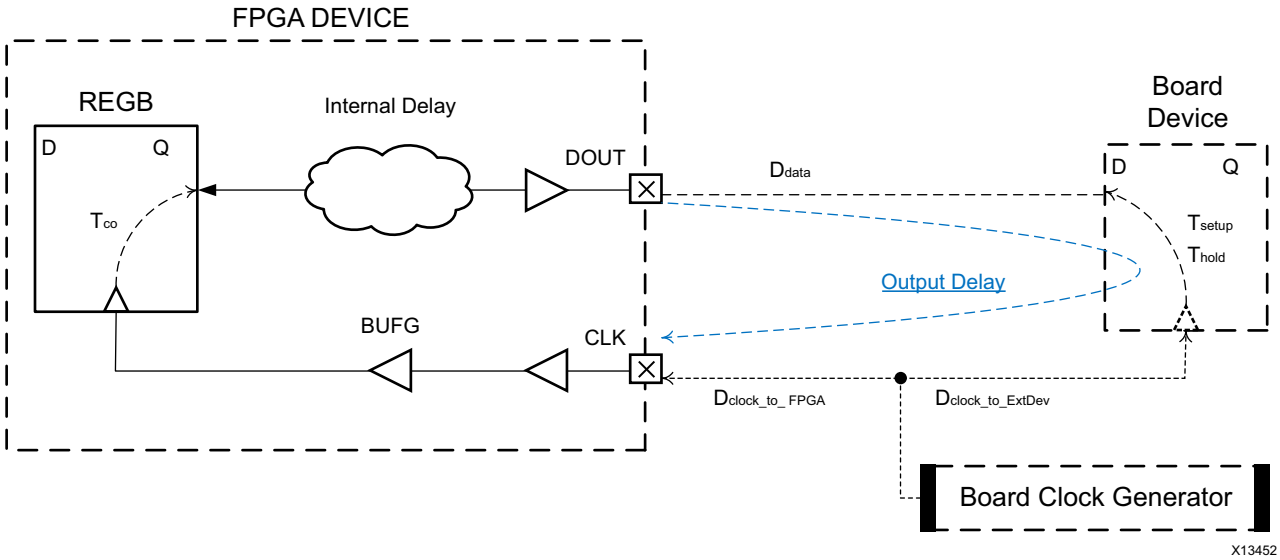



*Figure 3-75:* **Output Delay Computation**

The output delay values for the both types of analysis are:

```
Output Delay(max) = Tsetup + Ddata(max) + Dclock_to_FPGA(max) - Dclock_to_ExtDev(min)
Output Delay(min) = Ddata(min) - Thold + Dclock_to_FPGA(min) - Dclock_to_ExtDev(max)
```

The following figure shows a simple example of output delay constraints for both setup (max) and hold (min) analyses, assuming the sysClk clock has already been defined on the CLK port:

```
set_output_delay -max -clock sysClk 2.4 [get_ports DOUT]
set_output_delay -min -clock sysClk -1.1 [get_ports DOUT]
```

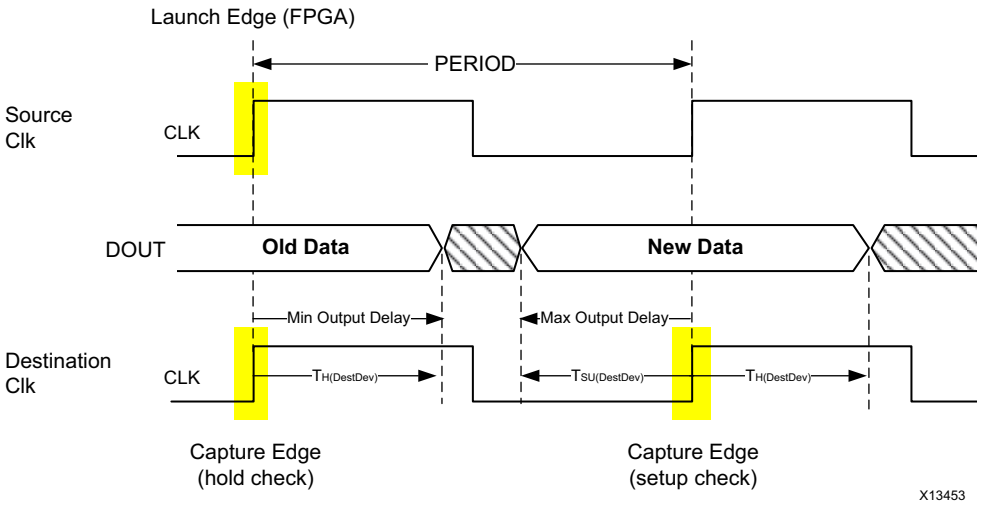



*Figure 3-76:* **Interpreting Min and Max Output Delays**

The output delay corresponds to the delay on the board before the capture edge. For a regular system synchronous interface where the clock and data board traces are balanced, the setup time of the destination device defines the output delay value for max analysis. And the destination device hold time defines the output delay for min analysis. The specified min output delay indicates the minimum delay that the signal will incur after coming out of the design, before it will be used for hold analysis at the destination device interface. Thus, the delay inside the block can be that much smaller. A positive value for min output delay means that the signal can have negative delay inside the design. This is why min output delay is often negative. For example:

```
set_output_delay -min -0.5 -clock CLK [get_ports DOUT]
```

means that the delay inside the design until DOUT has to be at least +0.5 ns to meet the hold time requirement.

### *Choosing the Reference Clock*

Depending on the clock tree topology that controls the sequential cells related to input or output ports, you have to choose the most appropriate clock to define the input or output

delay constraints. If the clock of the I/O path register is a generated clock, the delay constraint usually needs to be defined relative to the primary clock, which is defined upstream of the generated clocks. There are some exceptions to this rule that are explained in this section.

### Identifying the Clocks Related to Each Port

Before defining the I/O delay constraint, you must identify which clocks are related to each port. There are several ways to identify those clocks:

- Browse the Board Schematics
- Browse the Design Schematics
- Report Timing from or to the Port
- Using Automatically Identified Sampling Clocks

### Browse the Board Schematics

For a group of ports connected to a particular device on the board, you can use the same board clock that goes to both the device and the FPGA as the input or output delay reference clock. You need to verify in the device data sheet if the board clock is internally transformed for timing the I/O ports to make sure the FPGA design generates the same clock to control the timing of the related group of ports.

### Browse the Design Schematics

For each port, you can expand the path schematics to the first level of sequential cells, and then trace the clock pins of those cells back to the clock source(s). This approach can be impractical for ports that are connected to high fanout nets.

### Report Timing from or to the Port

Whether a port is already constrained or not, you can use the `report_timing` command to identify its related clocks in the design. Once all the timing clocks have been defined, you can report the worst path from or to the I/O port, create the I/O delay constraint relative to the clock reported, and re-run the same timing report from/to the other clocks of the design. If it appears that the port is related to more than one clock, create the corresponding constraint and repeat the process.

For example, the `din` input port is related to the clocks `clk1` and `clk2` inside the design:

```
report_timing -from [get_ports din] -sort_by group
```

The report shows that the `din` port is related to `clk1`. The input delay constraint is (for both min and max delay in this example):

```
set_input_delay -clock clk1 5 [get_ports din]
```

Rerun timing analysis with the same command as previously, and observe that `din` is also related to clk2 due to the `-sort_by group` option, which reports N paths per endpoint clock. You can add the corresponding delay constraint and re-run the report to validate that the `din` port is not related to another clock.

You can also run the same analysis using the Timing Summary report with the `-report_unconstrained` option. With only clock constraints in your design, the Unconstrained Paths section appears as follows:

```
------------------------------------------------
| Unconstrained Path Table
------------------------------------------------

Path Group      From Clock      To Clock
----------      ----------      --------
(none)
(none)          clk1
(none)          clk2
(none)                          clk1
(none)                          clk2
```

The fields without a clock name (or <NONE> in the Vivado IDE) refer to a group of paths where the startpoints (From Clock) or the endpoints (To Clock) are not associated with a clock. The unconstrained I/O ports fall in this category. You can retrieve their name by browsing the rest of the report. For example in the Vivado IDE, by selecting the Setup paths for the `clk1` to NONE category, you can see the ports driven by `clk1` in the To column:

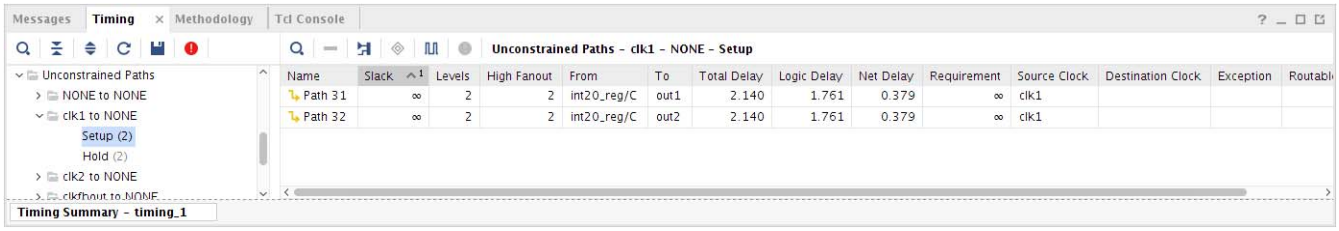



*Figure 3-77:* **Getting a List of Unconstrained Output Ports**

After adding the new constraints and applying them in memory, you must re-run the report to determine which ports are still unconstrained. For most designs, you must increase the number of reported paths to make sure all the I/O paths are listed in the report.

### Using Automatically Identified Sampling Clocks

You can use the `set_input_delay` and `set_output_delay` constraints without specifying the related clock. The Vivado Design Suite timing engine will analyze the design and associate each port with all the sampling clocks automatically. Then by reporting timing on the I/O paths, you can see how the tool constrained each I/O port. This is convenient for quickly constraining a design, but this type of generic constraints can become a problem if they are too generic and do not model the hardware reality accurately.

### Using a Primary Clock

A primary clock (that is, an incoming board clock) should be used when it directly controls the I/O path sequential cells, without traversing any clock modifying block. I/O delay lines are not considered as clock modifying blocks because they only affect the clock insertion delay and not the waveform. This case is illustrated by the two examples previously provided in Defining Input Delays and Defining Output Delays. Most of the time, the external device also has its interface characteristics defined with respect to the same board clock.

When the primary clock is compensated by a PLL or MMCM inside the FPGA with the zero hold violation (ZHOLD) mode, the I/O paths sequential cells are connected to an internal copy (for example, a generated clock) of the primary clock. Because the waveforms of both clocks are identical, Xilinx recommends using the primary clock as the reference clock for the input/output delay constraints.

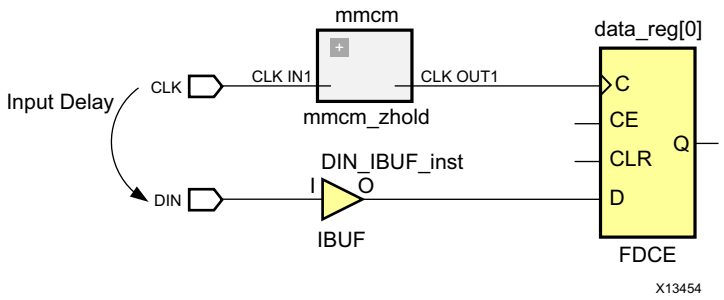



*Figure 3-78:* **Input Delay in the Presence of a ZHOLD MMCM in Clock Path**

The constraints are identical to the example provided in Defining Input Delays because the ZHOLD MMCM acts like a clock buffer with a negative insertion delay which corresponds to the amount of compensation.

### Using a Virtual Clock

When the board clock traverses a clock modifying block which transforms the waveform in addition to compensating the overall insertion delay, it is recommended to use a virtual clock as a reference clock for the input and output delay instead of the board clock. There are three main cases for using a virtual clock:

- The internal clock and the board clock have different period: The virtual clock must be defined with the same period and waveform as the internal clock. This results in a regular single-cycle path requirement on the I/O paths.
- For input paths, the internal clock has a positive shifted waveform compared to the board clock: the virtual clock is defined like the board clock, and a multicycle path constraint of 2 cycles for setup is defined from the virtual clock to the internal clock. These constraints force the setup timing analysis to be performed with a requirement of 1 clock cycle + amount of phase shift.

- For output paths, the internal clock has a negative shifted waveform compared to the board clock: the virtual clock is defined like the board clock and a multicycle path constraint of 2 cycles for setup is defined from the internal clock to the virtual clock. These constraints force the setup timing analysis to be performed with a requirement of 1 clock cycle + amount of phase shift.

To summarize, the use of a virtual clock adjusts the default timing analysis to avoid treating I/O paths as clock domain crossing paths with a very tight and unrealistic requirement.

**IMPORTANT:** *You only need to use the multicycle path for I/O paths with phase-shifted clocks when the phase-shift results in modification of the clock waveform. When the phase shift is added to the insertion delay of the clock modifying block and the clock waveform is preserved, you do not need to use a multicycle path. For more information, see this link in the Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906) [Ref 21].*

For example, consider the sysClk board clock that runs at 100 MHz and gets multiplied by an MMCM to generate clk266 that runs at 266 MHz. An output that is generated by clk266 should use clk266 as the reference clock. If you try to use sysClk as the reference clock (for the `set_output_delay` specification), it will appear as asynchronous clocks, and the path can no longer be timed as a single cycle.

### Using a Generated Clock

For an output source synchronous interface, the design generates a copy of the internal clock and forwards it to the board along with the data. This clock is usually used as the reference clock for the output data delay constraints whenever the intent is to control and report on the phase relationship (skew) between the forwarded clock and the data. The forwarded clock can also be used in input and output delay constraints for a system synchronous interface.

### Rising and Falling Reference Clock Edges

The clock edges used in the I/O constraint must reflect the data sheet of the external device connected to the FPGA device. By default, the `set_input_delay` and `set_output_delay` commands define a delay constraint relative to the rising reference clock edge. You must use the `clock_fall` option to specify a delay relative the falling clock edge. You can also specify separate constraints for delays related to both rising and falling clock edges by using the `add_delay` option with the second constraint on a port.

In most cases, the I/O reference clock edges correspond to the clock edges used to latch or launch the I/O data inside the FPGA. By analyzing the I/O timing paths, you can review which clock edges are used and verify that they correspond to the actual hardware behavior. If by mistake a rising clock edge is used as a reference clock for an I/O path that is only related to the falling clock edge internally, the path requirement is ½-period, which makes timing closure more difficult.

### Verifying Delay Constraints

Once the I/O timing constraints have been entered, it is important to review how timing is analyzed on the I/O paths and the amount of slack violation for both setup and hold checks. By using the timing reports from/to all ports for both setup and hold analysis (that is, `delay type = min_max`), you can verify that:

- The correct clocks and clock edges are used as reference for the delay constraints.
- The expected clocks are launching and capturing the I/O data inside the FPGA device.
- The violations can reasonably be fixed by placement or by setting the proper delay line tap configuration. If this is not the case, you must review the I/O delay values entered in the constraints and evaluate whether they are realistic, and whether you must modify the design to meet timing.

#### I/O Path Report Command Lines Example

```
report_timing -from [all_inputs] -nworst 1000 -sort_by group \
-delay_type min_max

report_timing -to [all_outputs] -nworst 1000 -sort_by group \
-delay_type min_max
```

Improper I/O delay constraints can lead to impossible timing closure. The implementation tools are timing driven and work on optimizing the placement and routing to meet timing. If the I/O path requirements cannot be met and I/O paths have the worst violations in the design, the overall design QoR will be impacted.

### Input to Output Feed-through Path

There are several equivalent ways to constrain a combinatorial path from an input port to an output port.

#### Example One

Use a virtual clock with a period greater or equal to the target maximum delay for the feed-through path, and apply max input and output delay constraints as follows:

```
create_clock -name vclk -period 10
set_input_delay -clock vclk <input_delay_val> [get_ports din] -max
set_output_delay -clock vclk <output_delay_val> [get_ports dout] -max
```

*where*

```
input_delay_val(max) +  feedthrough path delay (max) + output_delay_val(max)
<= vclk period.
```

In this example, only the maximum delay is constrained.

**Example Two**

Use a combination of min and max delay constraints between the feedthrough ports. Example:

```
set_max_delay -from [get_ports din] -to [get_ports dout] 10
set_min_delay -from [get_ports din] -to [get_ports dout] 2
```

This is a simple way to constrain both minimum and maximum delays on the path. Any existing input and output delay constraints on the same ports are also used during the timing analysis. For this reason, this style is not very popular.

The max delay is usually optimized and reported against the Slow timing corner, while the min delay is in the Fast timing corner. It is best to run a few iterations on the feedthrough path delay constraints to validate that they are reasonable and can be met by the implementation tools, especially if the ports are placed far from one another.

### *Using XDC Templates - Source Synchronous Interfaces*

While most users can properly write timing constraints for system synchronous interfaces, Xilinx recommends using I/O constraint templates for the source synchronous interfaces. The source synchronous constraints can be written in several ways. The templates provided by the Vivado Design Suite are based on the default timing analysis path requirement. The syntax is simpler, but the delay values must be adjusted to account for the fact that the setup analysis is performed with different launch and capture edges (1-cycle or 1/2-cycle) instead of same edge (0-cycle). The timing reports can be more difficult to read as the clock edges do not directly correspond to the active ones in hardware. You can navigate to these templates in Vivado GUI through **Tools > Language Templates > XDC > TimingConstraints > Input Delay Constraints > Source Synchronous**.

## Defining Clock Groups and CDC Constraints

The Vivado IDE times the paths between all the clocks in your design by default. You can use the following constraints to modify this default behavior:

- `set_clock_groups`: Disables timing analysis between groups of clocks that you identify but not between the clocks within a same group.
- `set_false_path`: Disables timing analysis between the clocks only in the direction specified by the `-from` and `-to` options.

In some cases, you might want to use the following constraints on one or more paths of the clock domain crossing (CDC) to limit latency or bus skew:

- `set_max_delay -datapath_only`: Sets the maximum delay constraints on asynchronous CDC paths to limit the latency.

  ***Note:*** If clock groups or false path constraints already exist between the clocks or on the same CDC paths, the maximum delay constraints will be ignored. Therefore, it is important to

thoroughly review every path between all clock pairs before choosing one CDC timing constraint over another to avoid constraints collision.

**RECOMMENDED:** *Xilinx also recommends running* `report_methodology` *to identify when a* `set_max_delay -datapath_only` *constraint is overridden by a* `set_clock_groups` *or* `set_false_path` *constraint. For details, see Running Report Methodology in Chapter 4.*

- `set_bus_skew`: Constrains a set of signals between asynchronous CDC paths by bus skew instead of latency.

**TIP:** *You can also set a bus skew constraint from the Vivado IDE. In the Timing Constraints window, expand* ***Assertions****, and double-click* ***Set Bus Skew****.*

### *Reviewing Clock Interactions*

Clocks that have a logical path between them are timed. The possible clock relationships are:

- Synchronous
- Asynchronous
- Exclusive

#### Synchronous

Clock relationships are synchronous when two clocks have a fixed phase relationship. This is the case when two clocks share the following:

- Common circuitry (common node)
- Primary clock (same initial phase)

#### Asynchronous

Clock relationships are asynchronous when they do not have a fixed phase relationship. This is the case when one of the following is true:

- They do not share any common circuitry in the design and do not have a common primary clock.
- They do not have a common period within 1000 cycles (unexpandable) and the timing engine cannot properly time them together.

If two clocks are synchronous but their common period is very small, the setup paths requirement is too tight for timing to be met. Xilinx recommends that you treat the two clocks as asynchronous and implement safe asynchronous CDC circuitry.

#### Exclusive

Clock relationships are exclusive when they propagate on a same clock tree and reach the same sequential cell clock pins but cannot physically be active at the same time.

### *Categorizing Clock Pairs*

The clock pairs can be categorized by using the following reports:

- Clock Interaction Report
- Check Timing Report

#### Clock Interaction Report

The Clock Interaction report provides a high-level summary of how two clocks are timed together:

- Do the two clocks have a common primary clock? When clocks are properly defined, all clocks that originate from the same source in the design share the same primary clock.
- Do the two clocks have a common period? This shows in the setup or hold path requirement column ("unexpandable"), when the timing engine cannot determine the most pessimistic setup or hold relationship.
- Are the paths between the two clocks partially or completely covered by clock groups or timing exception constraints?
- Is the setup path requirement between the two clocks very tight? This can happen, when two clocks are synchronous, but their period is not specified as an exact multiple (for example, due to rounding off). Over multiple clock cycles, the edges could drift apart, causing the worst case timing requirement to be very tight.

#### Check Timing Report

The Check Timing report (`multiple_clock`) identifies the clock pins that are reached by more than one clock and a `set_clock_groups` or `set_false_path` constraint has not already been defined between these clocks.

### *Constraining Exclusive Clock Groups*

You can use the regular timing or clock network reports to review the clock paths and identify the situations where two clocks propagate on a same clock tree and are used at the same time in a timing path where the startpoint and endpoint clock pins are connected to the same clock tree. This analysis can be a time consuming task. Instead, you can review the `multiple_clock` section of the Check Timing report. This section returns a list of clock pins and their associated timing clocks.

Based on the clock tree topology, you must apply different constraints:

- Overlapping Clocks Defined on the Same Clock Source
- Overlapping Clocks Driven by a Clock Multiplexer

### Overlapping Clocks Defined on the Same Clock Source

This occurs when two clocks are defined on the same netlist object with the `create_clock -add` command and represent the multiple modes of an application. In this case, it is safe to apply a clock groups constraint between the clocks. For example:

```
create_clock -name clk_mode0 -period 10 [get_ports clkin]
create_clock -name clk_mode1 -period 13.334 -add [get_ports clkin]
set_clock_groups -physically_exclusive -group clk_mode0 -group clk_mode1
```

If the `clk_mode0` and `clk_mode1` clocks generate other clocks, the same constraint needs to be applied to their generated clocks as well, which can be done as follows:

```
set_clock_groups -physically_exclusive \
-group [get_clocks -include_generated_clock clk_mode0] \
-group [get_clocks -include_generated_clock clk_mode1]
```

### Overlapping Clocks Driven by a Clock Multiplexer

When two or more clocks drive into a multiplexer (or more generally a combinatorial cell), they all propagate through and become overlapped on the fanout of the cell. Realistically, only one clock can propagate at a time, but timing analysis allows reporting several timing modes at the same time.

For this reason, you must review the CDC paths and add new constraints to ignore some of the clock relationships. The correct constraints are dictated by how and where the clocks interact in the design.

The following figure shows an example of two clocks driving into a multiplexer and the possible interactions between them before and after the multiplexer.

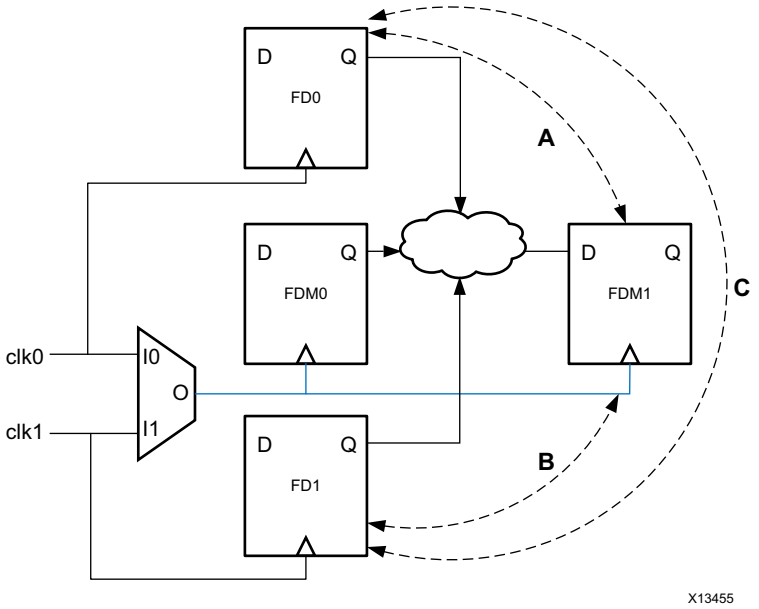



*Figure 3-79:* **Muxed Clocks**

- **Case in which the paths A, B, and C do not exist**

  `clk0` and `clk1` only interact in the fanout of the multiplexer (FDM0 and FDM1). It is safe to apply the clock groups constraint to `clk0` and `clk1` directly.

```
set_clock_groups -logically_exclusive -group clk0 -group clk1
```

- **Case in which only the paths A or B or C exist**

  `clk0` and/or `clk1` directly interact with the multiplexed clock. In order to keep timing paths A, B and C, the constraint cannot be applied to `clk0` and `clk1` directly. Instead, it must be applied to the portion of the clocks in the fanout of the multiplexer, which requires additional clock definitions.

```
create_generated_clock -name clk0mux -divide_by 1 \
-source [get_pins mux/I0] [get_pins mux/O]

create_generated_clock -name clk1mux -divide_by 1 \
  -add -master_clock clk1 \
-source [get_pins mux/I1] [get_pins mux/O]

set_clock_groups -physically_exclusive -group clk0mux -group clk1mux
```

### *Constraining Asynchronous Clock Groups and Clock Domain Crossings*

The asynchronous relationship can be quickly identified in the Clock Interaction report: clock pairs with no common primary clock or no common period (unexpanded). Even if clock periods are the same, the clocks will still be asynchronous, if they are being generated from different sources. The asynchronous Clock Domain Crossing (CDC) paths must be reviewed carefully to ensure that they use proper synchronization circuitry that does not rely on timing correctness and that minimizes the chance for metastability to occur. Asynchronous CDC paths usually have high skew and/or unrealistic path requirements. They should not be timed with the default timing analysis, which cannot prove they will be functional in hardware.

#### Report CDC

The Report CDC (`report_cdc`) command performs a structural analysis of the clock domain crossings in your design. You can use this information to identify potentially unsafe CDCs that might cause metastability or data coherency issues. Report CDC is similar to the Clock Interaction Report, but Report CDC focuses on structures and related timing constraints. Report CDC does not provide timing information because timing slack does not make sense on paths that cross asynchronous clock domains.

Report CDC identifies the most common CDC topologies as follows:

- Single bit synchronizers
- Multi-bit synchronizers for buses
- Asynchronous reset synchronizers
- MUX and CE controlled circuitry
- Combinatorial logic before synchronizer
- Multi-clock fanin to synchronizer
- Fanout to destination clock domain

For more information on the `report_cdc` command, see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21] and see report_cdc in the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 14].

Specific constraints should be applied to prevent default timing analysis on asynchronous clock domain crossings:

- Global Constraints Between Clocks in Both Directions
- Constraints on Individual CDC Paths

### Global Constraints Between Clocks in Both Directions

When there is no need to limit the maximum latency, the clock groups can be used. Following is an example to ignore paths between `clkA` and `clkB`:

```
set_clock_groups -asynchronous -group clkA -group clkB
```

When two master clocks and their respective generated clocks form two asynchronous domains between which all the paths are properly synchronized, the clock groups constraint can be applied to several clocks at once:

```
set_clock_groups -asynchronous \
-group {clkA clkA_gen0 clkA_gen1 …} \
-group {clkB clkB_gen0 clkB_gen1 …}
```

Or simply:

```
set_clock_groups -asynchronous \
-group [get_clocks -include_generated_clock clkA] \
-group [get_clocks -include_generated_clock clkB]
```

### Constraints on Individual CDC Paths

If a CDC bus uses gray-coding (e.g., FIFO) or if latency needs to be limited between the two asynchronous clocks on one or more signals, you must use the `set_max_delay` constraint with the option `-datapath_only` to ignore clock skew and jitter on these paths, plus override the default path requirement by the latency requirement. It is usually sufficient to use the source clock period for the max delay value, just to ensure that no more than one data is present on the CDC path at any given time.

When the ratio between clock periods is high, choosing the minimum of the source and destination clock periods is also a good option to reduce the transfer latency. A clean asynchronous CDC path should not have any logic between the source and destination sequential cells, so the Max Delay Datapath Only constraint is normally easy to meet for the implementation tools.

Some asynchronous CDC paths require a skew control between the bits of the bus instead of a constraint on the bus latency. Using a bus skew constraint prevents the receiving clock domain from latching multiple states of the bus on the same clock edge. You can set the bus skew constraint on the bus with `set_bus_skew` command. For example, you can apply `set_bus_skew` to a CDC bus that uses gray-coding instead of using the Max Delay Datapath Only constraint. For more information, see this link in the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 18].

For the paths that do not need latency control, you can define a point-to-point false path constraint.

### Clock Exceptions Precedence Over set_max_delay

When writing the CDC constraints, verify that the precedence is respected. If you use `set_max_delay -datapath_only` on at least one path between two clocks, the `set_clock_groups` constraint cannot be used between the same clocks, and the `set_false_path` constraint can only be used on the other paths between the two clocks.

In the following figure, the clock `clk0` has a period of 5ns and is asynchronous to `clk1`. There are two paths from `clk0` domain to `clk1` domain. The first path is a 1-bit data synchronization. The second path is a multi-bit gray-coded bus transfer.

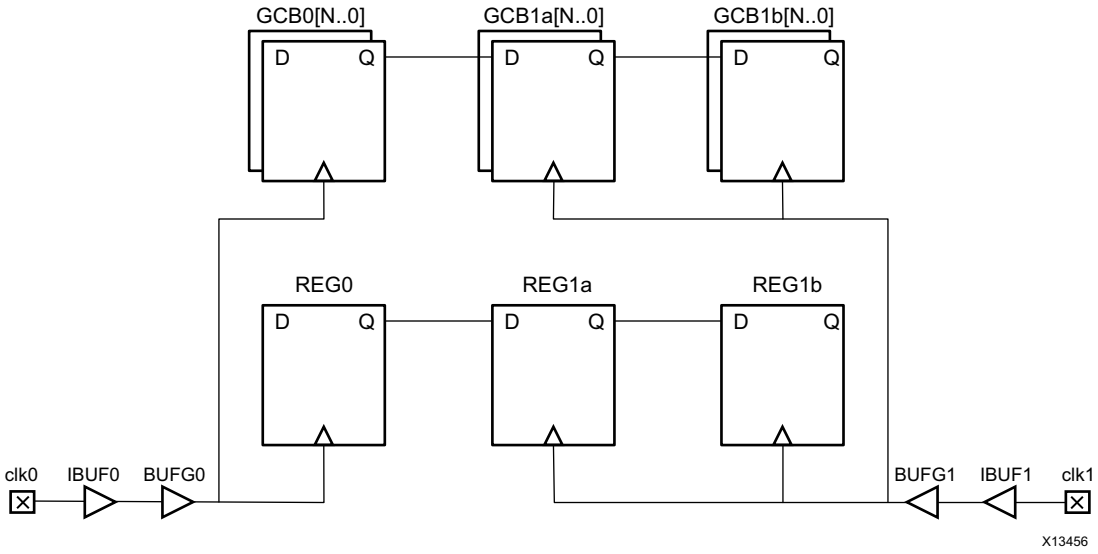



*Figure 3-80:* **Multiple Interactions Between Two Asynchronous Clocks**

The designer decides that the gray-coded bus transfer requires a Max Delay Datapath Only to limit the delay variation among the bits, so it becomes impossible to use a Clock Groups or False Path constraint between the clocks directly. Instead, two constraints must be defined:

```
set_max_delay -from [get_cells GCB0[*]] -to [get_cells [GCB1a[*]] \
-datapath_only 5
set_false_path -from [get_cells REG0] -to [get_cells REG1a]
```

There is no need to set a false path from `clk1` to `clk0` because there is no path in this example.

## Specifying Timing Exceptions

Timing exceptions are used to modify how timing analysis is done on specific paths. By default, the timing engine assumes that all paths should be timed with a single cycle requirement for setup analysis in order to cover the most pessimistic clocking scenario. For certain paths, this is not true. Following are a few examples:

- Asynchronous Clock Domain Crossing paths cannot be safely timed due to the lack of fixed phase relationship between the clocks. They should be ignored (Clock Groups, False Path), or simply have datapath delay constraint (Max Delay Datapath Only)
- The sequential cells launch and capture edges are not active at every clock cycle, so the path requirement can be relaxed accordingly (Multicycle Path)
- The path delay requirement needs to be tightened in order to increase the design margin in hardware (Max Delay)
- A path through a combinatorial cell is static and does not need to be timed (False Path, Case Analysis)
- The analysis should be done with only a particular clock driven by a multiplexer (Case Analysis).

In any case, timing exceptions must be used carefully and must not be added to hide real timing problems.

### *Timing Exceptions Guidelines*

Use a limited number of timing exceptions and keep them simple whenever possible. Otherwise, you will face the following challenges:

- The runtime of the compilation flow will significantly increase when many exceptions are used, especially when they are attached to a large number of netlist objects.
- Constraints debugging becomes extremely complicated when several exceptions cover the same paths.
- Presence of constraints on a signal can hamper the optimization of that signal. Therefore, including unnecessary exceptions or unnecessary points in exception commands can hamper optimization.

Following is an example of timing exceptions that can negatively impact the runtime:

```
set_false_path -from [get_ports din] -to [all_registers]
```

- If the `din` port does not have an input delay, it is not constrained. So there is no need to add a false path.
- If the `din` port feeds only to sequential elements, there is no need to specify the false path to the sequential cells explicitly. This constraint can be written more efficiently:

```
set_false_path -from [get_ports din]
```

- If the false path is needed, but only a few paths exist from the `din` port to any sequential cell in the design, then it can be more specific (`all_registers` can potentially return thousands of cells, depending upon the number of registers used in the design):

```
set_false_path -from [get_ports din] -to [get_cells blockA/config_reg[*]]
```

#### Timing Exceptions Precedence and Priority Rules

Timing exceptions are subject to strict precedence and priority rules. The most important rules are:

- The more specific the constraint, the higher the priority. For example:

```
set_max_delay -from [get_clocks clkA] -to [get_pins inst0/D] 12
set_max_delay -from [get_clocks clkA] -to [get_clocks clkB] 10
```

The first `set_max_delay` constraint has a higher priority because the `-to` option uses a pin, which is more specific than a clock.

- The exceptions priority is as follows:

1. `set_false_path`
2. `set_max_delay` or `set_min_delay`
3. `set_multicycle_path`

The `set_clock_groups` command is not considered a timing exception even though it is equivalent to two `set_false_path` commands between two clocks. It has higher precedence than the timing exceptions.

The `set_case_analysis` and `set_disable_timing` commands disable timing analysis on specific portions of the design. They have higher precedence than the timing exceptions.

For details on XDC precedence and priorities, see this link in the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 18].

### *Adding False Path Constraints*

False path exceptions can be added to timing paths to ignore slack computation on these paths. It is usually difficult to prove that a path does not need timing to be functional, even with simulation tools. Xilinx does not usually recommend using a false path unless the risk associated with it has been properly assessed and appear to be acceptable.

### Use Cases

The typical cases for using the false path constraint are:

- Ignoring timing on a path that is never active. For example, a path that crosses two multiplexers that can never let the data propagate in a same clock cycle because of the select pins connectivity.

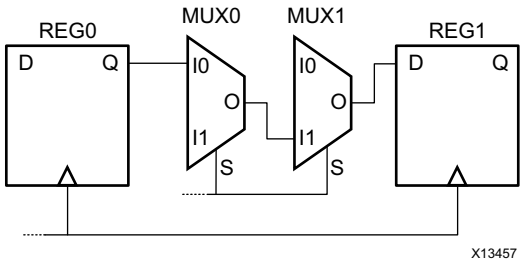



*Figure 3-81:* **Path Cannot be Sensitized**

```
set_false_path -through [get_pins MUX0/I0] -through [get_pins MUX1/I1]
```

- Ignoring timing on an asynchronous CDC path. This case is already discussed in Defining Clock Groups and CDC Constraints.
- Ignoring static paths in the design. Some registers take a value once during the initialization phase of the application and never toggle again. When they appear to be on the critical path of the design, they can be ignored for timing in order to relax the constraints on the implementation tools and help with timing closure. It is sufficient to define a false path constraint from the static register only, without explicitly specifying the paths endpoints. Example: the paths from a 32-bit configuration register `config_reg[31..0]` to the rest of the design can be ignored by adding the following false path constraint:

```
set_false_path -from [get_cells config_reg[*]]
```

#### Impact on Synthesis

The false path constraint is supported by synthesis and will only impact max delay (setup/recovery) path optimization. It is usually not needed to use false path exceptions during synthesis except for ignoring CDC paths.

#### Impact on Implementation

All the implementation steps are sensitive to the false path timing exception.

### Adding Min and Max Delay Constraints

The min and max delay exceptions are used to override the default path requirement respectively for hold/removal and setup/recovery checks by replacing the launch and capture edge times with the delay value from the constraint.

### Use Cases

Common reasons for using the min or max delay constraints are for:

- Over-constraining a few paths of the design by tightening the setup/recovery path requirement.

  This is useful for forcing the logic optimization or placement tools to work harder on some critical path cells, which can provide more flexibility to the router to meet timing later on (after removing the max delay constraint).

- Replacing a multicycle constraint.

  This is a valid, but not the recommended way, to relax the setup requirement on a path that has active launch and capture edges every N clock cycles. Although it is the only option to over-constrain a multicycle path by a fraction of a clock period to help with timing closure during the routing step. For example, a path with a multicycle constraint of 3 appears to be the worst violating path after route and fails timing by a few hundred ps.

  The original multicycle path constraint can be replaced by the following constraint during optimization and placement:

  ```
  set_max_delay -from [get_pins <startpointCell>/C] \
  -to [get_pins <endpointCell>/D] 14.5
  ```

  *where*

  14.5 corresponds to 3 clock periods (of 5 ns each), minus 500 ps that correspond to amount of extra margin desired.

- Constraining the maximum datapath delay on asynchronous CDC paths.

  This technique has already been described in Defining Clock Groups and CDC Constraints.

It is not common or recommended to force extra delay insertion on a path by using the `set_min_delay` constraint. The default min delay requirement for hold or removal is usually sufficient to ensure proper hardware functionality when the slack is positive.

### Impact on Synthesis

The `set_max_delay` constraint is supported by synthesis, including the `-datapath_only` option. The `set_min_delay` constraint is ignored.

### Impact on Implementation

The `set_max_delay` constraint replaces the setup path requirement and influences the entire implementation flow. The `set_min_delay` constraint replaces the hold path

requirement and only affects the router behavior whenever it introduces the need to fix hold.

#### Avoiding Path Segmentation

Path segmentation is introduced when specifying invalid startpoint or endpoint for the `-from` or `-to` options of the `set_max_delay` and `set_min_delay` commands only. When a `set_max_delay` introduces path segmentation on a path, the default hold analysis no longer takes place. You must constrain the same path with `set_min_delay` if you desire to constrain the hold analysis as well. The same rule applies with the `set_min_delay` command relative to the setup analysis.

Path segmentation must only be used by experts as it alters the fundamentals of timing analysis:

- Path segmentation breaks clock skew computation on the segmented path.
- Path segmentation can break more paths than the one constrained by the segmenting `set_max_delay` or `set_min_delay` command.

Path segmentation is reported by the tools in the log file when the constraints are applied. You must avoid it by using valid startpoints and endpoints:

- **Startpoints**

  clock, clock pin, sequential cell (implies valid startpoint pins of the cell), input or inout port

- **Endpoints**

  clock, input data pin of sequential cell, sequential cell (implies valid endpoint pins of the cell), output or inout port

  For details on path segmentation, see this link in the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 18].

## Adding Multicycle Path Constraints

Multicycle path exceptions must reflect the design functionality and must be applied on paths that do not have an active clock edge at every cycle, on either the source clock, the destination clock or both clocks. The path multiplier is expressed in terms of clock cycles, either based on the source clock when the `-start` option is used, or the destination clock when the `-end` option is used. This is particularly convenient for modifying the setup and hold relationships between the startpoint and endpoint independently of the clock period value.

The hold relationships are always tied to the setup ones. Consequently, in most cases, the hold relationship also needs to be separately adjusted after the setup one has been modified. This is why a second constraint with the `-hold` option is needed. The main

exception to this rule is for synchronous CDC paths between phase-shifted clocks: only setup needs to be modified.

### *Relaxing the Setup Requirement While Keeping Hold Unchanged*

This occurs when the source and destination sequential cells are controlled by a clock enable signal that activates the clock every N cycles. The following example has a clock enable active every 3 cycles, with the same clock for both startpoint and endpoint:

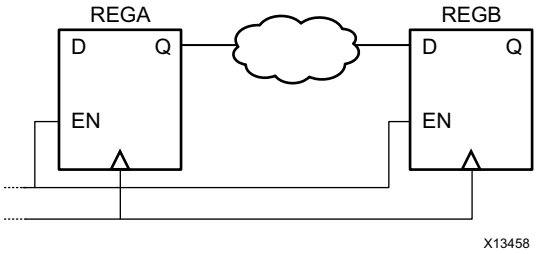



*Figure 3-82:* **Enabled Flops with Same Clock Signal**

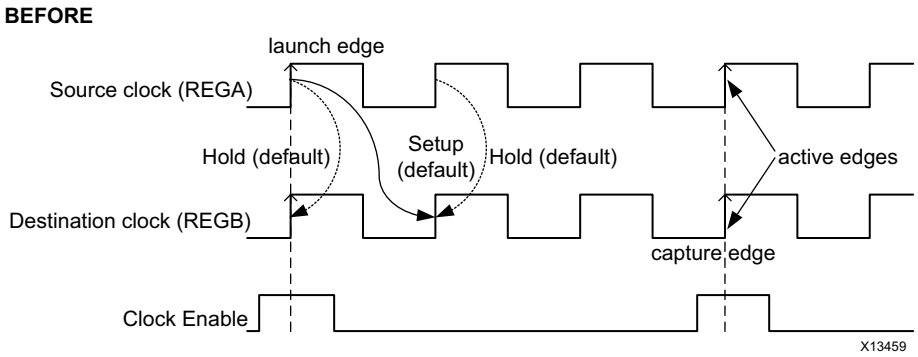



*Figure 3-83:* **Timing Diagram for Setup/Hold Check**

Constraints:

```
set_multicycle_path -from [get_pins REGA/C] -to [get_pins REGB/D] -setup 3
set_multicycle_path -from [get_pins REGA/C] -to [get_pins REGB/D] -hold 2
```

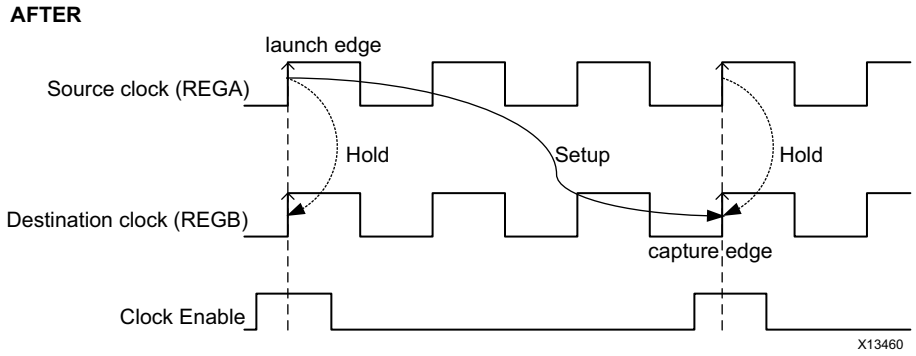



*Figure 3-84:* **Setup/Hold Checks Modified After Multi-cycle Specification**

***Note:*** With the first command, as the setup capture edge moved to the third edge (that is, by 2 cycles from its default position), the hold edge also moved by 2 cycles. The second command is for bringing the hold edge back to its original location by moving it again by 2 cycles (in the reverse direction).

For more information on other common multicycle path scenarios, such as phase shift and multicycle paths between synchronous clocks, see the see this link in the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 18].

**IMPORTANT:** *When the clock phase shift does not modify the clock waveform but is instead included in the insertion delay of the clock modifying block, you do not need to add a setup-only multicycle path to properly time the path from or to the clock. For more information, see this link in the Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906) [Ref 21].*

## *Impact on Synthesis and Implementation*

The `set_multicycle_path` constraint is supported by synthesis and can greatly improve the timing QoR (for setup only) by relaxing long paths that are functionally not active at every clock cycle.

As for synthesis, multicycle path exceptions help the implementation timing-driven algorithms to focus on the real critical paths. The hold requirements are important only during routing. If a setup relationship was adjusted with a `set_multicycle_path` constraint but not its corresponding hold relationships, the worst hold requirement may become too hard to meet if it is over 2 or 3 ns. This situation can have a negative impact on setup slack because of the additional delay inserted by the router while fixing hold violations.

### Common Mistakes

Following are two typical mistakes that you must absolutely avoid:

- Relaxing setup without adjusting hold back to same launch and capture edges in the case of a multicycle path not functionally active at every clock cycle. The hold requirement can become very high (at least one clock period in most cases) and impossible to meet.
- Setting a multicycle path exception between incorrect points in the design.

  This occurs when you assume that there is only one path from a startpoint cell to an endpoint cell. In some cases this is not true. The endpoint cell can have multiple data input pins, including clock enable and reset pins, which are active on at least two consecutive clock edges.

  For this reason, Xilinx recommends that you specify the endpoint pin instead of just the cell (or clock). Example: the endpoint cell REGB has three input pins: C, EN and D. Only the REGB/D pin should be constrained by the multicycle path exception, not the EN pin because it can change at every clock cycle. If the constraint is attached to a cell instead of a pin, all the valid endpoint pins are considered for the constraints, including the EN (clock enable) pin.

  To be safe, Xilinx recommends that you always use the following syntax:

```
set_multicycle_path -from [get_pins REGA/C] \
-to [get_pins REGB/D] -setup 3
set_multicycle_path -from [get_pins REGA/C] \
-to [get_pins REGB/D] -hold 2
```

## Other Advanced Timing Constraints

A few other timing constraints can be set to ignore and modify the default timing analysis:

- Case Analysis
- Disable Timing
- Data Check
- Max Time Borrow

### Case Analysis

The case analysis command is commonly used to describe a functional mode in the design by setting constants in the logic like what configuration registers do. It can be applied to input ports, nets, hierarchical pins, or leaf cell input pins. The constant value propagates through the logic and turns off the analysis on any path that can never be active. The effect is similar to how the false path exception works.

The most common example is to set a multiplexer select pin to 0 or 1 in order to only allow one of the two multiplexer inputs to propagate through. The following example turns off the analysis on the paths through the `mux/S` and `mux/I1` pins:

```
set_case_analysis 0 [get_pins mux/S]
```

### *Disable Timing*

The disable timing command turns off a timing arc in the timing database, which completely prevents any analysis through that arc. The disabled timing arcs can be reported by the `report_disable_timing` command.

**CAUTION!** *Use the disable timing command carefully. It can break more paths than desired!*

### *Data Check*

The `set_data_check` command sets the equivalent of a setup or hold timing check between two pins in a design. It is commonly used to constrain and report asynchronous interfaces. This command should be used by expert users.

### *Max Time Borrow*

The `set_max_time_borrow` command sets the maximum amount of time a latch can borrow from the next stage (logic after the latch), and give it the previous stage (logic before the latch). Latches are not recommended in general as they are difficult to test and validate in hardware. This command should be used by expert users.

## Defining Physical Constraints

Physical constraints are used to control floorplan, specific placement, I/O assignments, routers and similar functions. Make sure that each pin has an I/O location and standard specified. Physical Constraints are covered in the following user guides:

- *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 18] for locking placement and routing, including relative placement of macros
- *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21] for floorplanning
- *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 24] for configuration

# Implementation

## Overview of Synthesis and Implementation

After selecting your device, choosing and configuring the IP, and writing the RTL and the constraints, the next step is implementation. Implementation compiles the design through synthesis and place and route, and then generates the bitstream that is used to program the device. The implementation process might have some iterative loops, as discussed in Chapter 1, Introduction. This chapter describes the various implementation steps, highlights points for special attention, and gives tips and tricks to identify and eliminate specific bottlenecks.

## Running Synthesis

Synthesis takes in RTL and timing constraints and generates an optimized netlist that is functionally equivalent to the RTL. In general, the synthesis tool can take any legal RTL and create the logic for it. Synthesis requires realistic timing constraints, as described in Working with Constraints in Chapter 3 and Baselining the Design in Chapter 5.

For additional information about synthesis, refer to the following resources:

- *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 16]
- Vivado Design Suite QuickTake Video: Design Flows Overview

### Synthesis Attributes

Synthesis attributes allow you to control the logic inference in a specific way. Although synthesis algorithms are set to give the best results for the largest number of designs, there are often designs with differing requirements. In this case, you can use attributes to alter the design to improve QoR. For information on the attributes supported by synthesis, see the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 16].

***Note:*** Before retargeting your design to a new device, Xilinx recommends reviewing any synthesis attributes from previous design runs that target older devices.

When using the KEEP, DONT_TOUCH, and MAX_FANOUT attributes, be aware of the special considerations described in the following sections.

### *KEEP and DONT_TOUCH*

KEEP and DONT_TOUCH are valuable attributes for debugging a design. They direct the tool to not optimize the objects on which are they are placed.

- KEEP is used by the synthesis tool and is not passed along as a property in the netlist. KEEP can be used to retain a specific signal, for example, to turn off specific optimizations on the signal during synthesis.
- DONT_TOUCH is used by the synthesis tool and then passed along to the place and route tools so the object is never optimized.

Take care when using these attributes:

- A KEEP attribute on a register that receives RAM output prevents that register from being merged into the RAM, thereby preventing a block RAM from being inferred.
- Do not use these attributes on a level of hierarchy that is driving a 3-state output or bidirectional signal in the level above. If the driving signal and the 3-state condition are in this level of hierarchy, the IOBUF is not inferred, because the tool must change the hierarchy to create the IOBUF.
- Attributes that disable optimization often result in larger, higher power-consuming circuits. Xilinx recommends using these controls sparingly and removing them when no longer needed.

Also, be aware that there is a difference between putting DONT_TOUCH on a signal or on a level of hierarchy:

- If the attribute is placed on a signal, that signal is kept.
- If the attribute is placed on a level of hierarchy, the tool does not touch the boundaries of that hierarchy, and no constant propagation occurs through the hierarchy. However, optimizations inside that level of hierarchy are retained.

### *MAX_FANOUT*

MAX_FANOUT forces the synthesis to replicate logic in order to meet a fanout limit. The tool is able to replicate logic, but not inputs or black boxes. Accordingly, if a MAX_FANOUT attribute is placed on a signal that is driven by a direct input to the design, the tool is unable to handle the constraint.

Take care to analyze the signals on which a MAX_FANOUT is placed. If a MAX_FANOUT is placed on a signal that is driven by a register with a DONT_TOUCH or drives signals that are in a different level of hierarchy when the DONT_TOUCH attribute is on that hierarchy, the MAX_FANOUT attribute will not be honored.

Synthesis appends the replicated cells with `_rep` for the first replication and subsequent replications are `_rep__0`, `_rep__1` and so on. These cells can be seen in the post synthesized netlist by selecting **Edit** > **Find** on cells.

**IMPORTANT:** *Use MAX_FANOUT sparingly during synthesis. The `phys_opt_design` command in the Vivado® tools has a better idea of the placement of the design, and can do a better job of replication than synthesis. If a specific fanout is desired, it is often worth the time and effort to manually code the extra registers.*

## Block-Level Synthesis Strategy

With Vivado synthesis, you can use various strategies and global settings to customize how the design is synthesized. In most cases, these options are global and affect the entire design. You can use the block-level synthesis strategy to synthesize different levels of hierarchy with different global options in a top-down flow. This flow is faster and easier to perform than a bottom-up compile. You can set constraints for the full design rather than setting constraints for a lower level and then resetting for the top level.

### *Using the Block-Level Synthesis Strategy*

Set the block-level synthesis strategy in the XDC file using the following syntax:

```
set_property BLOCK_SYNTH.<option_name> <value> [get_cells <instance_name>]
```

Where:

- `<option_name>` is the option to be set.
- `<value>` is the value to be assigned to the option.
- `<instance_name>` is the hierarchical instance on which to set the option.

***Note:*** These properties are always set on hierarchical instances. This allows modules or entities that are instantiated more than once to be synthesized with different options.

For example, you can set the following strategies in an XDC file:

```
set_property BLOCK_SYNTH.RETIMING 1                 [get_cells U1]
set_property BLOCK_SYNTH.STRATEGY {AREA_OPTIMIZED}  [get_cells U2]
set_property BLOCK_SYNTH.STRATEGY {AREA_OPTIMIZED}  [get_cells U3]
set_property BLOCK_SYNTH.STRATEGY {DEFAULT}         [get_cells U3/inst1]
```

Vivado synthesis is performed as shown in the following figure.

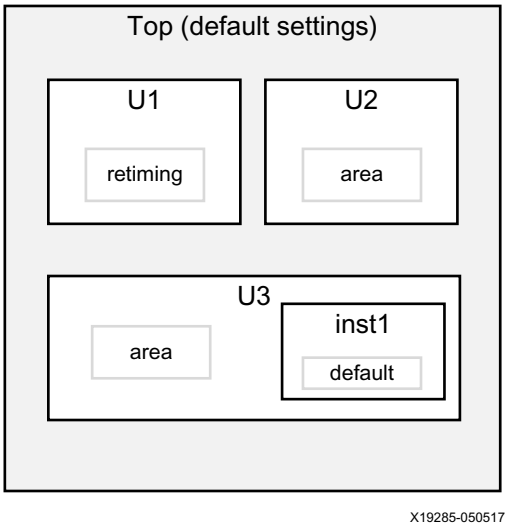



*Figure 4-1:* **Block-Level Synthesis Strategy Example**

You can set multiple BLOCK_SYNTH properties on the same instance to experiment with different options. For example:

```
set_property BLOCK_SYNTH.STRATEGY {ALTERNATE_ROUTABILITY} [get_cells inst]
set_property BLOCK_SYNTH.FSM_EXTRACTION {OFF} [get_cells inst]
```

When working with IP, you can use the block-level synthesis strategy as follows:

- If the IP is compiled globally, you can use this strategy on the top level of the IP.
- If the IP is out-of-context, you cannot use the strategy, because the IP appears as a black box. Instead, use global settings when compiling the IP.

***Note:*** For more information on this feature and the supported strategies and options, see the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 16].

# Moving Past Synthesis

Be sure that the netlist you obtained during synthesis is of good quality so that it does not create problems downstream. The following sections cover important items to check before proceeding with the rest of the implementation flow.

## Reviewing and Cleaning DRCs

The `report_drc` command runs design rule checks (DRCs) to look for common design issues and errors. There are multiple rule decks. The default rule deck checks the following:

- Post-synthesis netlist
- I/O, BUFG, and other placement specific requirements
- Attributes and wiring on MGTs, IODELAYs, MMCMs, PLLs and other primitives

**RECOMMENDED:** *Review and correct DRC violations as early as possible in the design process to avoid timing or logic related issues later in the implementation flow.*

## Running Report Methodology

Due to the importance of the UltraFast design methodology, the Vivado tools provide a Methodology Report that specifically checks for compliance with methodology. The tools run different checks depending on the stage of the design process:

- RTL design: RTL lint-style checks
- Synthesized and Implemented designs: netlist, constraints and timing checks.

To run these checks at the Tcl prompt, open the design to be validated and enter following Tcl command:

```
report_methodology
```

To run these checks from the Vivado IDE, open the design to be validated, and select **Tools > Report > Report Methodology**. Any violations are listed in the Methodology window, as shown in the following figure. If a specific methodology violation does not need to be cleaned for your design, make sure that you understand the violation and its implication clearly and why the violation does not negatively impact your design.

***Note:*** For Xilinx supplied IP cores, the violations have been already reviewed and checked.

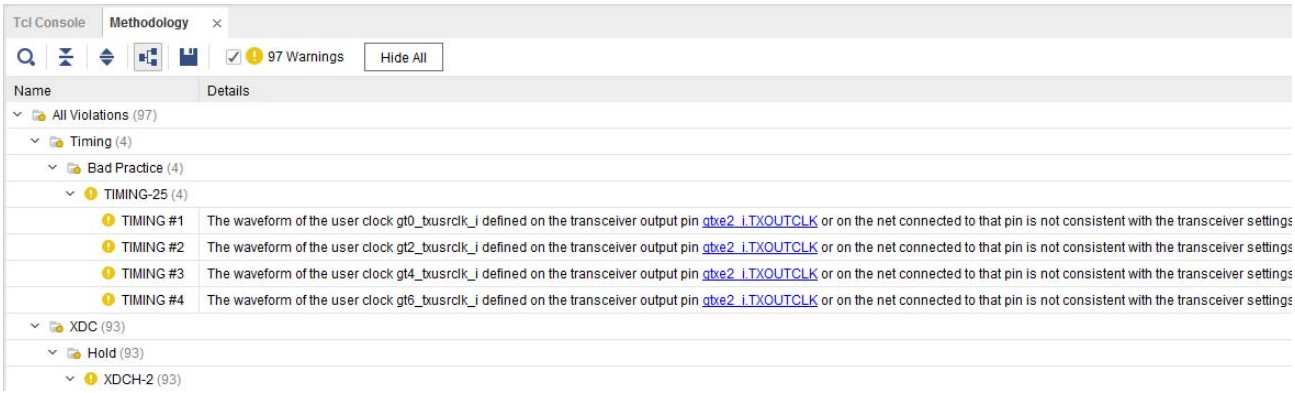



*Figure 4-2:* **Methodology Window**

**RECOMMENDED:** *To identify common design issues, run Report Methodology the first time you synthesize the design. Run this report again when there is a change in constraints, clocking topologies, or large logic changes.*

**TIP:** *You can also set these checks to run by default after routing. In the Properties view of the Implementation Run Properties window, select* ***Run report ultrafast methodology after routing****.*

For more information on running Report Methodology, see the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) [Ref 9] and *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21].

## Reviewing the Synthesis Log

You must review the synthesis log files and confirm that all messages given by the tool match your expectations in terms of the design intent. Pay special attention to Critical Warnings and Warnings. In most cases, Critical Warnings need to be cleaned up for a reliable synthesis result.

**CAUTION!** *If a message appears more than 100 times, the tool writes only the first 100 occurrences to the synthesis log file. You can change the limit of 100 through the Tcl command* `set_param messaging.defaultLimit`*.*

## Reviewing Timing Constraints

You must provide clean timing constraints, along with timing exceptions, where applicable. Bad constraints result in long runtime, performance issues, and hardware failures.

**RECOMMENDED:** *Review all Critical Warnings and Warnings related to timing constraints which indicate that constraints have not been loaded or properly applied.*

For more information, see Organizing the Design Constraints in Chapter 3.

## Meeting Post-Synthesis Timing

The following sections discuss how to meet post-synthesis timing.

### *Following Guidelines to Address Remaining Violations*

**IMPORTANT:** *Analyze timing post-synthesis to identify the major design issues that must be resolved before you move forward in the flow.*

HDL changes tend to have the biggest impact on QoR. You are therefore better off solving problems before implementation to achieve faster timing convergence. When analyzing timing paths, pay special attention to the following:

- Most frequent offenders (that is, the cells or nets that show up the most in the top worst failing timing paths)
- Paths sourced by unregistered block RAMs
- Paths sourced by SRL
- Paths containing unregistered, cascaded DSP blocks
- Paths with large number of logic levels
- Paths with large fanout

For more information see Timing Closure in Chapter 5.

### *Dealing with High Levels of Logic*

Identifying long logic paths is useful to diagnose difficult QOR challenges. Estimated net delays post-synthesis are close to the best possible placement. To evaluate if a path with high logic-level delay is meeting timing, you can generate timing reports with no net delay. Timing closure cannot be achieved on paths that are still violating timing with no net delays. For more information, see Timing Closure in Chapter 5.

### *Reviewing Utilization*

It is important to review utilization for LUT, FF, block RAM, and DSP components independently. A design with low LUT/FF utilization might still experience placement difficulties if block RAM utilization is high. The `report_utilization` command generates a comprehensive utilization report with separate sections for all design objects.

***Note:*** After synthesis, utilization numbers might change due to optimization later in the design flow.

### *Reviewing Clock Trees*

This section discusses reviewing clock trees, including clock buffer utilization and clock tree topology.

#### Clock Buffer Utilization

The `report_clock_utilization` command provides details on clock primitive utilization. Observe the architecture clocking rules to avoid downstream placement issues. For example, in 7 series devices, a BUFH can only fanout to loads in its clock region. Invalid placement constraints or very high fanout for regional clock buffers might cause issues in the placer. For designs with very high clock buffer utilization, it might be necessary to lock the clock generators and some regional clock buffers to aid placement.

For some interfaces needing very tight timing relationship, it is sometimes better to lock specific resources for these signals which need very tight timing relationship, for example, source synchronous interfaces. In general, as a starting point for your design, lock only the I/Os unless there are specific reasons not to follow this approach as cited above.

For more information on recommended placement constraints, see Timing Closure in Chapter 5.

#### Clock Tree Topology

When working with clock trees, follow these recommendations:

- Run the `report_clock_networks` command to show the clock network in detail tree view.
- Utilize clock trees in a way to minimize skew.
- For the outputs of PLLs and MMCMs, use the same clock buffer type to minimize skew.
- Look for unintended cascaded BUFG elements that can introduce additional delay, skew, or both.

## Implementing the Design

Vivado Design Suite implementation includes all steps necessary to place and route the netlist onto the FPGA device resources, while meeting the design's logical, physical, and timing constraints. For additional information about implementation, refer to the following resources:

- *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19]
- Vivado Design Suite QuickTake Video: Design Flows Overview

## Using Project Mode vs. Non-Project Mode

You can run implementation in Project Mode or Non-Project Mode. Project Mode provides the project infrastructure such as runs management, file sets management, reports generation, and cross probing. Non-Project Mode provides easy integration and is driven by a Tcl script which must explicitly call all the desired reports along the flow. For additional information about these modes, see this link in the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 6].

### *Strategies*

Strategies are used by the Vivado Design Suite to control both the tool options and the reports that are generated by synthesis and implementation runs in Project Mode. You can use the strategies to adjust the implementation goals and to control the reports that are generated. For more information on strategies, see the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

**RECOMMENDED:** *Try the default strategy Vivado Design Suite implementation defaults first. It provides a good trade-off between runtime and design performance.*

***Note:*** Strategies are tool and version specific. In some cases, strategies might require a longer runtime.

### *Directives*

Directives provide different modes of behavior for the following implementation commands:

- `opt_design`
- `place_design`
- `phys_opt_design`
- `route_design`

Use the default directive initially. Use other directives when the design nears completion to explore the solution space for a design. You can specify only one directive at a time. For more information on directives, see the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

### *Iterative Flows*

In Non-Project Mode, you can iterate between various optimization commands with different options. For example, you can run `phys_opt_design -directive AggressiveFanoutOpt` followed by `phys_opt_design -directive AlternateFlowWithRetiming` to run different physical synthesis optimizations on a placed design that does not meet timing.

Running `phys_opt_design` iteratively can provide timing improvement. The `phys_opt_design` command attempts to optimize the top timing problem paths. By running `phys_opt_design` iteratively, lower-level timing problems can benefit from the optimization. Running `phys_opt_design` at the post-route stage reroutes any nets that might have been unrouted. Therefore, after running `phys_opt_design` at post-route, you do not need to explicitly run `route_design`.

## Analyzing a Design at Different Stages Using Checkpoints

The Vivado Design Suite uses a physical design database to store placement and routing information. Design checkpoint files (`.dcp`) allow you to save (`write_checkpoint` command) and restore (`read_checkpoint` command) this physical database at key points in the design flow. Checkpoints are a snapshot of the design at a specific point in the flow. In Project Mode, the Vivado tools automatically generate design checkpoint files and store them in the implementation runs directory. These can be opened in a separate instance of Vivado.

This design checkpoint file includes the following:

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

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

A few common examples for the use of checkpoints are:

- Saving results so you can go back and do further analysis on that part of the flow.
- Trying `place_design` using multiple directives and saving the checkpoints for each. This would allow you to select the place_design checkpoint with the best timing results for the subsequent implementation steps.

For more information on checkpoints, see the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

## Using Interactive Report Files

After opening a checkpoint, you can read in and immediately analyze generated reports in the Vivado IDE. To generate the reports, use the following reporting commands and append the `-rpx <filename.rpx>` option:

```
report_timing_summary
report_timing
report_power
report_methodology
report_drc
```

After the checkpoint is open, you can open the interactive report file using **File > Open Interactive Report**.

***Note:*** In Project Mode, the interactive reports are generated and opened automatically.

**RECOMMENDED:** *When a report is generated, there is a size limit on the RPX file. Therefore, Xilinx recommends using the catch command to prevent errors that might stop the flow. For example:*

```
catch {report_timing_summary -rpx timing_summary.rpx -file timing_summary.rpt}
```

## Using Incremental Compile Flows

In the Vivado Design Suite, you can use incremental compile to reuse existing placement and routing data, which reduces implementation runtime and produces more predictable results. When working with designs that have 95% or higher reuse, incremental place and route typically achieves at least a twofold improvement over normal place and route runtimes while maintaining the WNS of the reference run. For more information, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

**RECOMMENDED:** *Incremental compile is most useful during critical stages of the design cycle when changes to the flow scripts are difficult to make. Ensure that your flow scripts include incremental compile early in the design cycle so you can enable incremental compile during critical periods.*

### *Incremental Compile Flow Modes*

Incremental compile supports both high and low reuse modes, which enable different directives and alter the behavior of the flow. For details on these reuse modes, including the supported directives and target WNS, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

### *Runtime Considerations*

In high reuse mode where 95% or more of the design is reused, runtimes can be reduced by half. As reuse declines, the benefits also decline.

In low reuse mode, runtime is not predictable. When the place and route runs get closer to meeting timing, the Vivado tools might increase runtime to meet timing. In other cases, the Vivado tools might decrease runtime if existing placement and routing data is reused efficiently.

The following factors can also affect runtime:

- If critical path placement and routing cannot be reused, more effort is required to preserve timing.
- Small design changes can introduce new timing problems that did not exist in the reference design.

### *Parallel Flows*

Run the standard flow runs in parallel with incremental compile runs for maximum flexibility. This allows you to take advantage of the benefits of incremental compile while preserving timing closure. It also allows you to use the incremental compile algorithms, which are different from the standard flow and can yield different results.

**TIP:** *You can update the reference checkpoint regularly to help maintain the effectiveness of the flow. Where resources are limited, launch incremental compile runs before launching standard flow runs to take advantage of better runtimes.*

## Opening the Synthesized Design

The first steps after synthesis are to read the netlist from the synthesized design into memory and apply design constraints. You can open the synthesized design in various ways, depending on the flow used. For more information, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

## Logic Optimization (opt_design)

Vivado Design Suite logic optimization optimizes the current in-memory netlist. Since this is the first view of the assembled design (RTL and IP blocks), the design can usually be further optimized. By default the `opt_design` command performs logic trimming, removing of cells with no loads, propagating constant inputs, and block RAM power optimization. It also optionally performs other optimizations such as remap, which combines LUTs in series into fewer LUTs to reduce path depth.

### *Optimization Analysis*

The `opt_design` command generates messages detailing the results for each optimization phase. After optimization you can run `report_utilization` to analyze utilization improvements. To better analyze optimization results, use the `-verbose` option to see additional details of the logic affected by `opt_design` optimization.

For more information on logic optimization, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

## Power Optimization in Implementation

For optimizing your design for power, see Power Optimization in Chapter 5.

## Placement (place_design)

The Vivado Design Suite placer engine positions cells from the netlist onto specific sites in the target Xilinx device.

### *Placement Analysis*

Use the timing summary report after placement to check the critical paths.

- Paths with very large negative setup time slack may require that you check the constraints for completeness and correctness, or logic restructuring to achieve timing closure.
- Paths with very large negative hold time slack are most likely due to incorrect constraints or bad clocking topologies and should be fixed before moving on to route design.
- Paths with small negative hold time slack are likely to be fixed by the router. You can also run `report_clock_utilization` after `place_design` to view a report that breaks down clock resource and load counts by clock region.

For more information, see Timing Closure in Chapter 5. For more information on placement, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

## Physical Optimization (phys_opt_design)

Physical optimization is an optional step of the flow. It performs timing-driven optimization on the negative-slack paths of a design. Optimizations involve replication, retiming, hold fixing, and placement improvement. Because physical optimization automatically performs all necessary netlist and placement changes, `place_design` is not required after `phys_opt_design`.

### *Need for Physical Synthesis*

To determine if a design would benefit from physical synthesis, evaluate timing after placement. Analyze failing paths for fanout. High fanout critical paths can benefit from fanout optimization. Additionally, high-fanout data, address and control nets of large RAM blocks involving multiple block RAMs that fail timing after `route_design` might benefit from Forced Net Replication. For more information on physical synthesis, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

## Routing (route_design)

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

### *Route Analysis*

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

Common examples of poor timing constraints include cross-clock paths and incorrect multi-cycle paths causing route delay insertion for hold fixing. Congested areas can be addressed by targeted fanout optimization in RTL synthesis or through physical optimization. You can preserve all or some of the design hierarchy to prevent cross-boundary optimization and reduce the netlist density. Or you can use floorplan constraints to ease congestion.

For more information, see Timing Closure in Chapter 5. For more information on routing, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

# Design Closure

## Overview of Design Closure

Design closure consists of meeting both timing and power requirements and successfully writing a configuration bitstream to validate the functionality in hardware. Design closure usually takes several iterations between results analysis, design modification, and constraints modification.

## Timing Closure

Timing closure consists of the design meeting all timing requirements. It is easier to reach timing closure if you have the right HDL and constraints for synthesis. In addition, it is important to iterate through the synthesis stages with improved HDL, constraints, and synthesis options, as shown in the following figure.

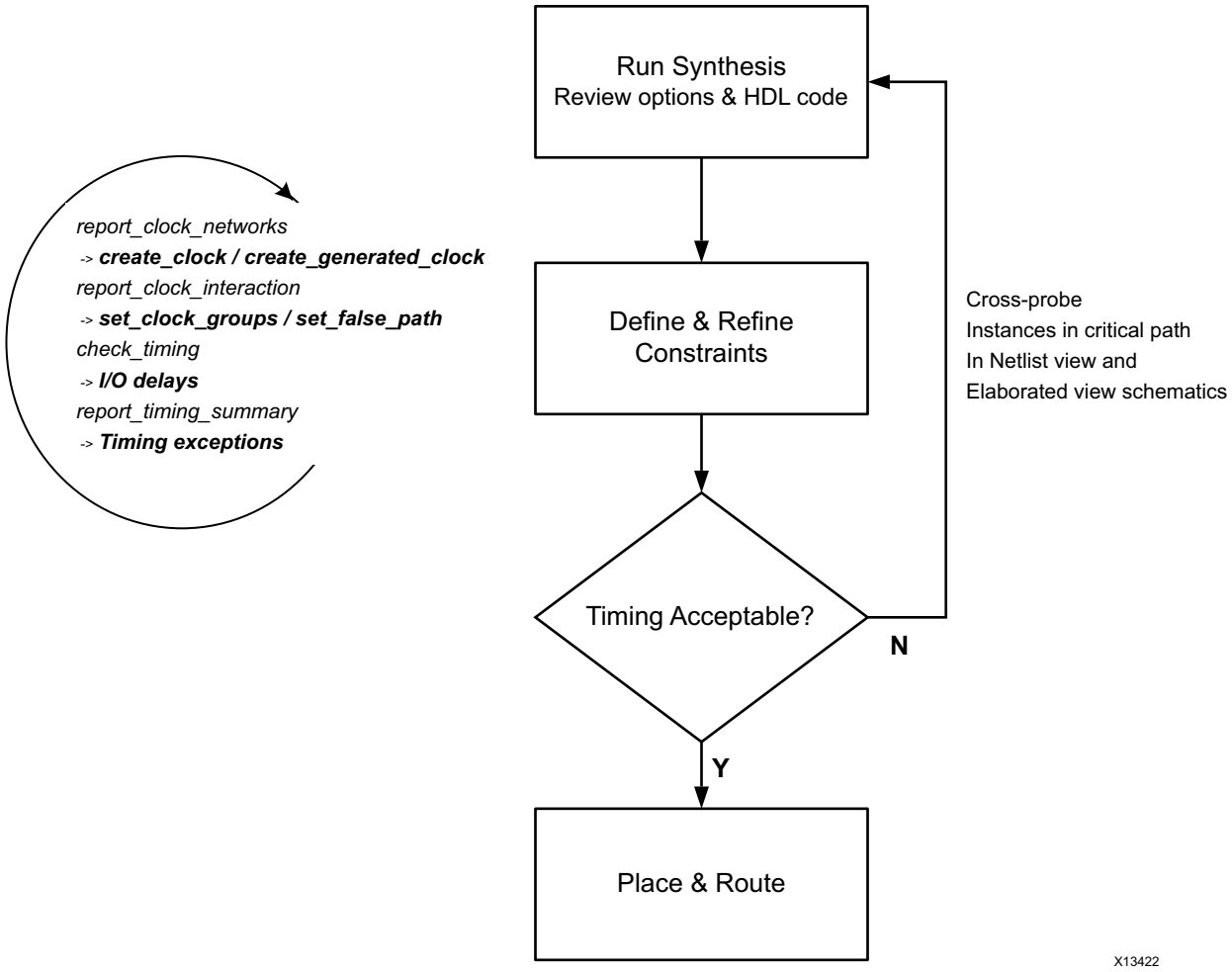



*Figure 5-1:* **Design Methodology for Rapid Convergence**

To successfully close timing, follow these general guidelines:

- When initially not meeting timing, evaluate timing throughout the flow.
- Focus on WNS (of each clock) as the main way to improve TNS.
- Review large WHS violations (<-1 ns) to identify missing or inappropriate constraints.
- Revisit the trade-offs between design choices, constraints, and target architecture.
- Know how to use the tool options and XDC.
- Be aware that the tools do not try to further improve timing (additional margin) after timing is met.

The following sections provide recommendations for reviewing the completeness and correctness of the timing constraints using methodology DRCs and baselining, identifying the timing violation root causes, and addressing the violations using common techniques.

# Understanding Timing Closure Criteria

Timing closure starts with writing valid constraints that represent how the design will operate in hardware. Review the Timing Summary report as described in the following sections.

## Checking for Valid Constraints

Review the Check Timing section of the Timing Summary report to quickly assess the timing constraints coverage, including the following:

- All active clock pins are reached by a clock definition.
- All active path endpoints have requirement with respect to a defined clock (setup/hold/recovery/removal).
- All active input ports have an input delay constraint.
- All active output ports have an output delay constraint.
- Timing exceptions are correctly specified.

**CAUTION!** *Excessive use of wildcards in constraints can cause the actual constraints to be different from what you intended. Use the* `report_exceptions` *command to identify timing exception conflicts and to review the netlist objects, timing clocks, and timing paths covered by each exception.*

In addition to `check_timing`, the Methodology report (TIMING and XDC checks) flags timing constraints that can lead to inaccurate timing analysis and possible hardware malfunction. You must carefully review and address all reported issues.

***Note:*** When baselining the design, you must use all Xilinx® IP constraints. Do not specify user I/O constraints, and ignore the violations generated by `check_timing` and `report_methodology` due to missing user I/O constraints.

## Checking for Positive Timing Slacks

The following timing metrics indicate timing violations. Numbers must be positive to meet timing. For more information, see Understanding Timing Reports.

- Setup/Recovery (max delay analysis): WNS > 0ns and TNS = 0ns
- Hold/Removal (min delay analysis): WHS > 0ns and THS = 0ns
- Pulse Width: WPWS > 0ns and TPWS = 0ns

### *Understanding Timing Reports*

The Timing Summary report provides high-level information on the timing characteristics of the design compared to the constraints provided. Review the timing summary numbers during signoff:

- TNS (Total Negative Slack): The sum of the setup/recovery violations for each endpoint in the entire design or for a particular clock domain. The worst setup/recovery slack is the WNS (Worst Negative Slack).
- THS (Total Hold Slack): The sum of the hold/removal violations for each endpoint in the entire design or for a particular clock domain. The worst hold/removal slack is the WHS (Worst Hold Slack).
- TPWS (Total Pulse Width Slack): The sum of the violations for each clock pin in the entire design or a particular clock domain for the following checks:
  - minimum low pulse width
  - minimum high pulse width
  - minimum period
  - maximum period
  - maximum skew (between two clock pins of a same leaf cell)
- WPWS (Worst Pulse Width Slack): The worst slack for all pulse width, period, or skew checks on any given clock pin.

The Total Slack (TNS, THS or TPWS) only reflects the violations in the design. When all timing checks are met, the Total Slack is null.

The timing path report provides detailed information on how the slack is computed on any logical path for any timing check. In a fully constrained design, each path has one or several requirements that must all be met in order for the associated logic to function reliably.

The main checks covered by WNS, TNS, WHS, and THS are derived from the sequential cell functional requirements:

- Setup time: The time before which the new stable data must be available before the next active clock edge in order to be safely captured.
- Hold requirement: The amount of time the data must remain stable after an active clock edge to avoid capturing an undesired value.
- Recovery time: The minimum time required between the time the asynchronous reset signal has toggled to its inactive state and the next active clock edge.
- Removal time: The minimum time after an active clock edge before the asynchronous reset signal can be safely toggled to its inactive state.

A simple example is a path between two flip-flops that are connected to the same clock net.

After a timing clock is defined on the clock net, the timing analysis performs both setup and hold checks at the data pin of the destination flip-flop under the most pessimistic, but reasonable, operating conditions. The data transfer from the source flip-flop to the destination flip-flop occurs safely when both setup and hold slacks are positive.

For more information on timing analysis, see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21].

## Checking That Your Design is Properly Constrained

Before looking at the timing results to see if there are any violations, be sure that every synchronous endpoint in your design is properly constrained.

Run `check_timing` to identify unconstrained paths. You can run this command as a standalone command, but it is also part of `report_timing_summary`. In addition, `report_timing_summary` includes an Unconstrained Paths section where N logical paths without timing requirements are listed by the already defined source or destination timing clock. N is controlled by the `-max_path` option.

After the design is fully constrained, run the `report_methodology` command and review the TIMING and XDC checks to identify non-optimal constraints, which will likely make timing analysis not fully accurate and lead to timing margin variations in hardware.

***Note:*** For Vivado Design Suite 2015.4 and previous releases, use the `report_drc -ruledeck methodology_checks` Tcl command instead.

### *Fixing Issues Flagged by check_timing*

The `check_timing` Tcl command reports that something is missing or wrong in the timing definition. When reviewing and fixing the issues flagged by `check_timing`, focus on the most important checks first. Following are the checks listed from most important to least important.

#### No Clock and Unconstrained Internal Endpoints

This allows you to determine whether the internal paths in the design are completely constrained. You must ensure that the unconstrained internal endpoints are at zero as part of the Static Timing Analysis signoff quality review.

Zero unconstrained internal endpoints indicate that all internal paths are constrained for timing analysis. However, the correct value of the constraints is not yet guaranteed.

#### Generated Clocks

Generated clocks are a normal part of a design. However, if a generated clock is derived from a master clock that is not part of the same clock tree, this can cause a serious problem. The timing engine cannot properly calculate the generated clock tree delay. This results in erroneous slack computation. In the worst case situation, the design meets timing according to the reports but does not work in hardware.

#### Loops and Latch Loops

A good design does not have any combinational loops, because timing loops are broken by the timing engine. The broken paths are not reported during timing analysis or evaluated during implementation. This can lead to incorrect behavior in hardware, even if the overall timing requirements are met.

#### No Input/Output Delays and Partial Input/Output Delays

All I/O ports must be properly constrained.

**RECOMMENDED:** *Start by validating baselining constraints and then complete the constraints with the I/O timing.*

#### Multiple Clocks

Multiple clocks are usually acceptable. Xilinx recommends that you ensure that these clocks are expected to propagate on the same clock tree. You must also verify that the paths requirement between these clocks does not introduce tighter requirements than needed for the design to be functional in hardware.

If this is the case, you must use `set_clock_groups` or `set_false_path` between these clocks on these paths. Any time that you use timing exceptions, you must ensure that they affect only the intended paths.

**IMPORTANT:** *Because the XDC follows Tcl syntax and semantics rules, the order of constraints matters.*

### *Fixing Issues Flagged by report_methodology*

The `report_methodology` command reports additional constraints and timing analysis issues, which you must carefully review before and after running the place and route tools. This section describes the three main XDC and TIMING categories of checks, along with their relative impact on timing closure and hardware stability. You must focus on resolving the checks that impact timing closure first.

### Methodology DRCs with Impact on Timing Closure

The DRCs shown in the following table flag design and timing constraint combinations that increase the stress on implementation tools, leading to impossible or inconsistent timing closure. These DRCs usually point to missing clock domain crossing (CDC) constraints, inappropriate clock trees, or inconsistent timing exception coverage due to logic replication. They must be addressed with highest priority.

*Table 5-1:* **Timing Closure Methodology DRCs**

| Check | Description |
|---|---|
| TIMING-6 | No common primary clock between related clocks |
| TIMING-7 | No common node between related clocks |
| TIMING-8 | No common period between related clocks |
| TIMING-14 | LUT on the clock tree |
| TIMING-15 | Large hold violation on inter-clock path |
| TIMING-16 | Large setup violation |
| TIMING-30 | Sub-optimal master source pin selection for generated clock |
| XDCB-3 | Same clock mentioned in multiple groups in the same `set_clock_groups` command |
| XDCH-1 | Hold option missing in multicycle path constraint |
| XDCV-1 | Incomplete constraint coverage due to missing original object used in replication |
| XDCV-2 | Incomplete constraint coverage due to missing replicated objects |

### Methodology DRCs with Impact on Signoff Quality

The DRCs shown in the following table do not usually flag issues that impact the ease of closing timing. Instead, these DRCs flag problems with timing analysis accuracy due to non-recommended constraints. Even when setup and hold slacks are positive, the hardware might not function properly under all operating conditions. Most checks refer to clocks not defined on the boundary of the design, clocks with unexpected waveform, missing timing requirements, or inappropriate CDC circuitry. For this last category, use the `report_cdc` command to perform a more comprehensive analysis.

*Table 5-2:* **Signoff Quality Methodology DRCs**

| Check | Description |
|---|---|
| TIMING-1, TIMING-2, TIMING-3, TIMING-4, TIMING-27 | Non-recommended clock source point definition |
| TIMING-5, TIMING-25, TIMING-19 | Unexpected clock waveform |
| TIMING-9, TIMING-10 | Unknown or incomplete CDC circuitry |
| TIMING-11 | Inappropriate `set_max_delay -datapath_only` command |
| TIMING-12 | Clock Reconvergence Pessimism Removal disabled |
| TIMING-13, TIMING-23 | Incomplete timing analysis due to broken paths |
| TIMING-17, TIMING-18, TIMING-20, TIMING-27 | Missing clock or input/output delay constraints |
| TIMING-21, TIMING-22 | Issues with MMCM compensation |
| TIMING-24 | Overridden `set_max_delay -datapath_only` command |
| TIMING-29 | Inconsistent pair of multicycle paths |
| TIMING-35 | No common node in paths with the same clock |

### Other Timing Methodology DRCs

Other TIMING and XDC checks identify constraints that can incur higher runtime, override existing constraints, or are highly sensitive to netlist names change. The corresponding information is useful for debugging constraints conflicts. You must pay particular attention to the TIMING-28 check (Auto-derived clock referenced by a timing constraint), because the auto-derived clock names can change when modifying the design source code and resynthesizing. In this case, previously defined constraints will not work anymore or will apply to the wrong timing paths.

## Baselining the Design

Baselining is a process in which you create the simplest timing constraints and initially ignore I/O timing. After all clocks are completely constrained, all paths with start and endpoints within the design are automatically constrained. This provides an easy mechanism to identify internal device timing challenges, even while the design is evolving. Because the design might also have clock domain crossings, baseline constraints must also include the relationship among the specified clocks, including generated clocks.

When baselining the design, you must meet timing after each implementation step by analyzing and resolving timing challenges throughout the flow. First, you create simple and valid constraints to give a realistic picture of timing in the Vivado implementation tools. Then, while iterating through different implementation steps, you solve timing violations before moving onto the next step. The following figure shows the baselining process.

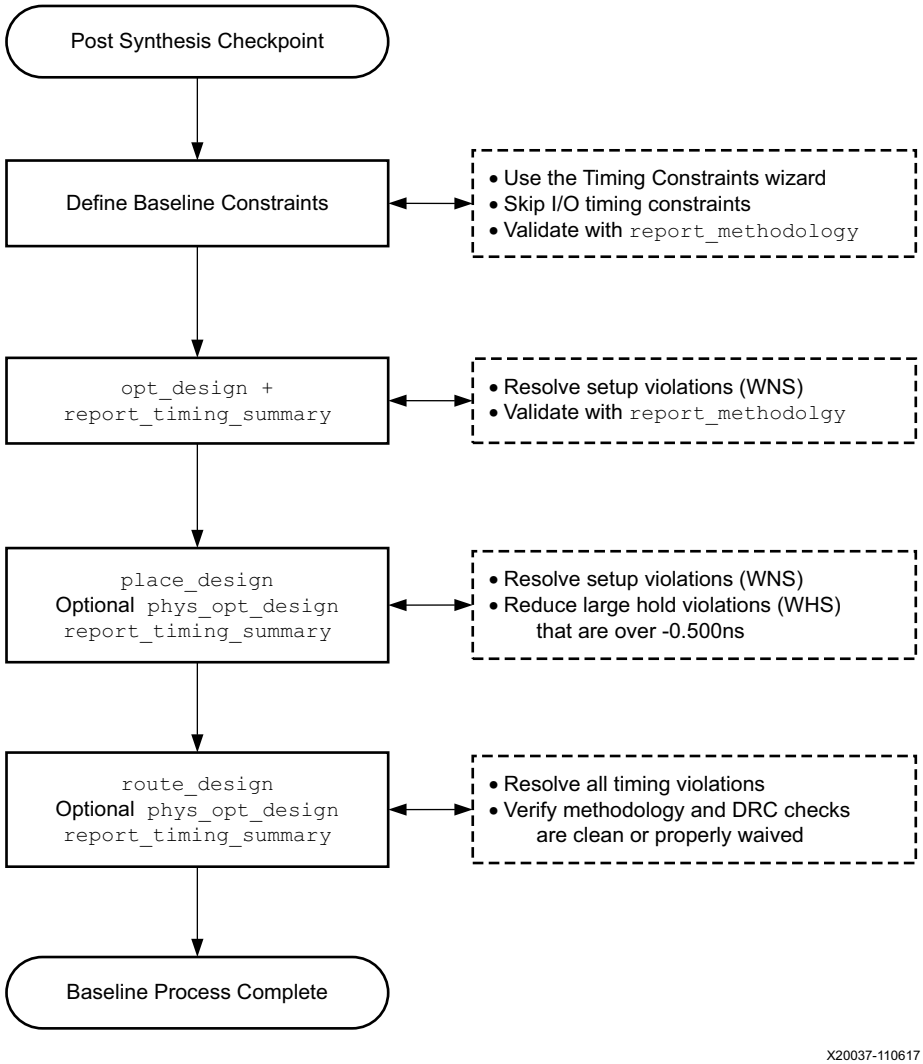



*Figure 5-2:* **Baselining the Design**

After baselining is complete, you can:

- Eliminate smaller timing violations
- Achieve full constraint coverage
- Individually baseline new modules before adding the modules to the top-level design

**RECOMMENDED:** *Xilinx recommends that you create the baseline constraints very early in the design process, and plan any major change to the design HDL against these baseline constraints.*

### *Defining Baseline Constraints*

To create the simplest set of constraints, use a valid post-synthesis Vivado checkpoint without user timing constraints. With the checkpoint open, use the Timing Constraints wizard to define the constraints. The wizard guides you through the process of creating constraints in a structured manner.

Not all constraints need to be defined at this stage. The Vivado tools ignore I/O timing by default if there are no constraints. Therefore, you do not need to define I/O timing constraints at this point. Instead, define the I/O timing constraints later in the flow after the baselining process is complete.

**TIP:** *When using the Timing Constraints wizard, deselect the suggested I/O timing constraints.*

To get an accurate picture of internal timing in the device, define the following constraints:

- All clock constraints
- Clock domain crossings (CDC)

  CDC paths between synchronous clocks are safely timed by default, but you must use safe CDC circuitry and specify timing exceptions between asynchronous clocks.

- Paths with timing that is impossible to meet

  Modify the RTL to reduce the logic and meet the path requirements.

**IMPORTANT:** *All Xilinx IP or partner IP are delivered with specific XDC constraints that comply with the Xilinx constraints methodology. The IP constraints are automatically included during synthesis and implementation. You must keep the IP constraints intact when creating the baselining constraints.*

If you do not use the Timing Constraints wizard to define the constraints, the following sections cover the steps you must take to define the baseline constraints manually.

### Identifying Which Clocks Must be Created

Begin by loading the post synthesized netlist or checkpoint into the Vivado IDE. In the Tcl console, use the `reset_timing` command to ensure that all timing constraints are removed.

Use the `report_clock_networks` Tcl command to create a list of all the primary clocks that must be defined in the design. The resulting list of clock networks shows which clock constraints should be created. Use the **Timing Constraints Editor** to specify the appropriate parameters for each clock.

### Verifying That No Clocks Are Missing

After the clock network report shows that all clock networks are constrained, you can begin verifying of the accuracy of the generated clocks. Because the Vivado tools automatically propagate clock constraints through clock-modifying blocks, it is important to review the constraints that were generated. Use `report_clocks` to show which clocks were created with a `create_clock` constraint and which clocks were generated.

***Note:*** MMCMs, PLLs, and clock buffers are clock-modifying blocks. For UltraScale™ devices, GTs are also clock-modifying blocks.

The `report_clocks` results show that all clocks are propagated. The difference between the primary clocks (created with `create_clock`) and the generated clocks is displayed in the attributes field:

- Clocks that are propagated (P) only are primary clocks.
- Clocks that are derived from other clocks are shown as both propagated (P) and generated (G).
- Clocks that are generated by a clock-modifying block are shown as auto-derived (A).
- Other attributes indicate that an auto-derived clock was renamed (R), a generated clock has an inverted waveform (I) relative to the incoming master clock, or a primary clock is virtual (V).

You can also create generated clocks using the `create_generated_clock` constraint. For more information, see the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 18].

Attributes
  P: Propagated
  G: Generated
  A: Auto-derived
  R: Renamed
  V: Virtual
  I: Inverted

| Clock | Period(ns) | Waveform(ns) | Attributes | Sources |
|---|---|---|---|---|
| sysClk | 10.000 | {0.000 5.000} | P | {sysClk} |
| clkfbout | 10.000 | {0.000 5.000} | P,G,A | {clkgen/mmcm_adv_inst/CLKFBOUT} |
| cpuClk | 20.000 | {0.000 10.000} | P,G,A,R | {clkgen/mmcm_adv_inst/CLKOUT0} |
| wbClk_4 | 20.000 | {0.000 10.000} | P,G,A | {clkgen/mmcm_adv_inst/CLKOUT1} |
| usbClk_3 | 10.000 | {0.000 5.000} | P,G,A | {clkgen/mmcm_adv_inst/CLKOUT2} |
| phyClk0_2 | 10.000 | {0.000 5.000} | P,G,A | {clkgen/mmcm_adv_inst/CLKOUT3} |
| phyClk1_1 | 10.000 | {0.000 5.000} | P,G,A | {clkgen/mmcm_adv_inst/CLKOUT4} |
| fftClk_0 | 10.000 | {0.000 5.000} | P,G,A | {clkgen/mmcm_adv_inst/CLKOUT5} |

*Figure 5-3:* **Clock Report Shows the Clocks Generated from Primary Clocks**

**TIP:** *To verify that there are no unconstrained endpoints in the design, see the Check Timing report (no_clock category). The report is available from within the Report Timing Summary or by using the* `check_timing` *Tcl command.*

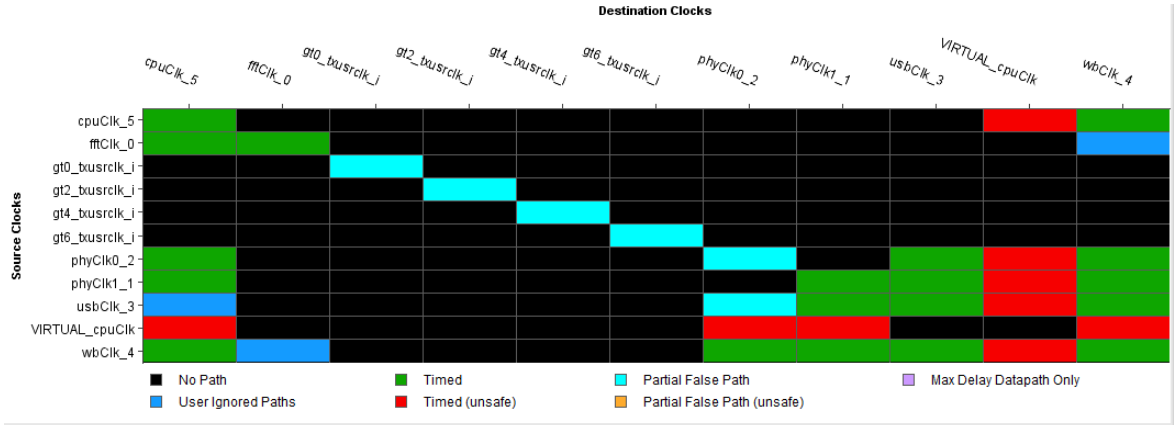
## *Constraining Clock Domain Crossings*

Upon verification of the clocking constraints, you must identify asynchronous and over-constrained clock domain crossing paths.

***Note:*** This section does not explain how to properly cross clock region boundaries. Instead, it explains how to identify which crossings exist and how to constrain them.

### Reviewing Clock Relationships

You can view the relationship between clocks using the `report_clock_interaction` Tcl command. The report shows a matrix of source clocks and destination clocks. The color in each cell indicates the type of interaction between clocks, including any existing constraints between them. The following figure shows a sample clock interaction report.




| Id | Source Clock | Destination Clock | Edges (WNS) | WNS (ns) | TNS (ns) | Failing Endpoints (TNS) | Total Endpoints (TNS) | Path Req (WNS) | Edges (WHS) | WHS (ns) | THS (ns) | Clock Pair Classification | Inter-Clock Constraints |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 26 | phyClk1_1 | cpuClk_5 | rise - rise | 1.901 | 0.000 | 0 | 61 | 7.000 | rise - rise | 1.040 | 0.000 | Clean | Timed |
| 31 | fftClk_0 | cpuClk_5 | rise - rise | 1.063 | 0.000 | 0 | 61 | 7.000 | rise - rise | 1.488 | 0.000 | Clean | Timed |
| 34 | VIRTUAL_c... | cpuClk_5 | rise - rise | -1.372 | -196.991 | 532 | 834 | | rise - rise | 1.089 | 0.000 | Virtual Clock | Timed (unsafe) |
| 13 | wbClk_4 | fftClk_0 | | | | 0 | 1635 | | | | | Ignored | Asynchronous Groups |
| 33 | fftClk_0 | fftClk_0 | rise - rise | 0.673 | 0.000 | 0 | 8320 | 7.000 | rise - rise | 0.046 | 0.000 | Clean | Timed |
| 1 | gt0_txusrclk_i | gt0_txusrclk_i | rise - rise | 9.923 | 0.000 | 0 | 222 | 12.800 | rise - rise | 0.135 | 0.000 | Clean | Partial False Path |
| 2 | gt2_txusrclk_i | gt2_txusrclk_i | rise - rise | 9.734 | 0.000 | 0 | 222 | 12.800 | rise - rise | 0.095 | 0.000 | Clean | Partial False Path |
| 3 | gt4_txusrclk_i | gt4_txusrclk_i | rise - rise | 8.856 | 0.000 | 0 | 222 | 12.800 | rise - rise | 0.123 | 0.000 | Clean | Partial False Path |

*Figure 5-4:* **Sample Clock Interaction Report**

The following table explains the meaning of each color in this report.

*Table 5-3:* **report_clock_interaction Colors**

| Color | Label | Meaning | What Next |
|---|---|---|---|
| Black | No path | No interaction among these clock domains. | Primarily for information unless you expected these clock domains to be interacting. |
| Green | Timed | There is interaction among these clock domains, and the paths are getting timed. | Primarily for information unless you do not expect any interaction among the clock domains. |
| Cyan | Partial False Path | Some of the paths for the interacting domains are not being timed due to user exceptions. | Ensure that the timing exceptions are really desired. |
| Red | Timed (unsafe) | There is interaction among these clock domains, and the paths are being timed. However, the clocks appear to be independent (and hence, asynchronous). | Check whether these clocks are supposed to be declared as asynchronous, or whether they are supposed to be sharing a common primary source. |
| Orange | Partial False Path (unsafe) | There is interaction among these clock domains. The clocks appear to be independent (and hence, asynchronous). However, only some of the paths are not timed due to exceptions. | Check why some paths are not covered by timing exceptions. |
| Blue | User Ignored Paths | There is interaction among these clock domains, and the paths are not being timed due to clock groups or false path timing exceptions. | Confirm that these clocks are supposed to be asynchronous. Also, check that the corresponding HDL code is written correctly to ensure proper synchronization and reliable data transfer across clock domains. |
| Light blue | Max Delay Datapath Only | There is interaction among these clock domains, and the paths are getting timed through: `set_max_delay -datapath only`. | Confirm that the clocks are asynchronous and that the specified delay is correct. |

Before the creation of any false paths or clock group constraints, the only colors that appear in the matrix are black, red, and green. Because all clocks are timed by default, the process of decoupling asynchronous clocks takes on a high degree of significance. Failure to decouple asynchronous clocks often results in a highly over-constrained design.

### Identifying Clock Pairs Without Common Primary Clocks

The clock interaction report indicates whether or not each pair of interacting clocks has a common primary clock source. Clock pairs that do not share a common primary clock are frequently asynchronous to each other. Therefore, it is helpful to identify these pairs by sorting the columns in the report using the Common Primary Clock field. The report does not determine whether clock-domain crossing paths are or are not designed properly.

Use the `report_cdc` Tcl command for a comprehensive analysis of clock domain crossing circuitry between asynchronous clocks. For more information on the `report_cdc` command, see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21] and see report_cdc in the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 14].

### Identifying Tight Timing Requirements

For each clock pair, the clock interaction report also shows setup requirement of the worst path. Sort the columns by **Path Req (WNS)** to view a list of the tightest requirements in the design. Figure 5-4 shows the timing report sorted by WNS column. Review these requirements to ensure that no invalid tight requirements exist.

The Vivado tools identify the path requirements by expanding each clock out to 1000 cycles, then determining where the closest, non-coincident edge alignment occurs. When 1000 cycles are not sufficient to determine the tightest requirement, the report shows Not Expanded, in which case you must treat the two clocks as asynchronous.

For example, consider a timing path that crosses from a 250 MHz clock to a 200 MHz clock:

- The positive edges of the 200 MHz clock are {0, 5, 10, 15, 20 ...}.
- The positive edges of the 250 MHz clock are {0, 4, 8, 12, 16, 20 ...}.

The tightest requirement for this pair of clocks occurs when the following is true:

- The 250 MHz clock has a rising edge at 4 ns
- The next rising edge of the 200 MHz clock is at 5 ns.

This results in all paths timed from the 250 MHz clock domain into the 200 MHz clock domain being timed at 1 ns.

***Note:*** The simultaneous edge at 20 ns is *not* the tightest requirement in this example, because the capture edge cannot be the same as the launch edge.

Because this is a fairly tight timing requirement, you must take additional steps. Depending on the design, one of the following constraints might be the correct way to handle these crossings:

- `set_clock_groups` / `set_false_path` / `set_max_delay -datapath_only`

  Use one of these constraints when treating the clock pair as asynchronous. Use the `report_cdc` Tcl command to validate that the clock domain crossing circuitry is safe.

- `set_multicycle_path`

  Use this constraint when relaxing the timing requirement, assuming proper clock circuitry controls the launch and capture clock edges accordingly.

If nothing is done, the design might exhibit timing violations that cross these two domains. In addition, all of the best optimization, placement and routing might be dedicated to these paths instead of given to the real critical paths in the design. It is important to identify these types of paths before any timing-driven implementation step.

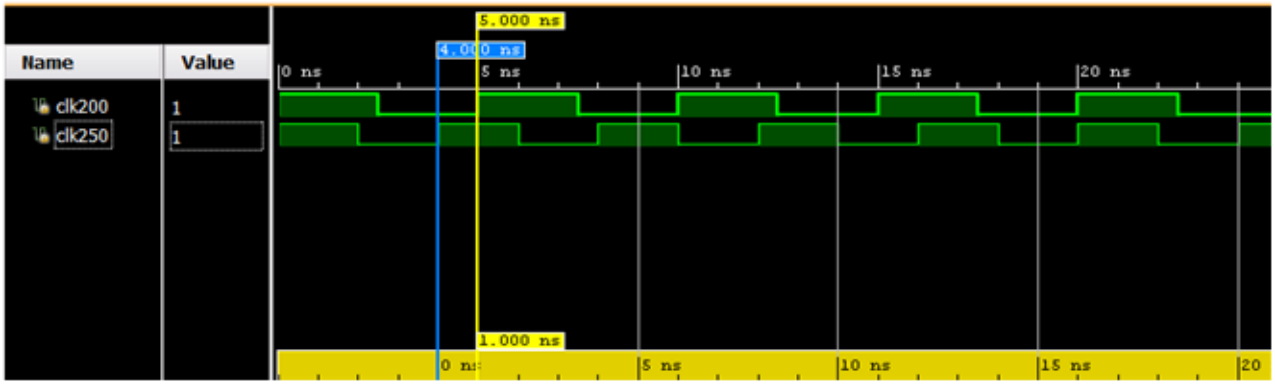



*Figure 5-5:* **Clock Domain Crossing from 250MHz to 200 MHz**

### Constraining Both Primary and Generated Clocks at the Same Time

Before any timing exceptions are created, it is helpful to go back to `report_clock_networks` to identify which primary clocks exist in the design. If all primary clocks are asynchronous to each other, you can use a single constraint to decouple the primary clocks from each other and to decouple their generated clocks from each other. Using the primary clocks in `report_clock_networks` as a guide, you can decouple each clock group and associated clocks as shown in the following figure.

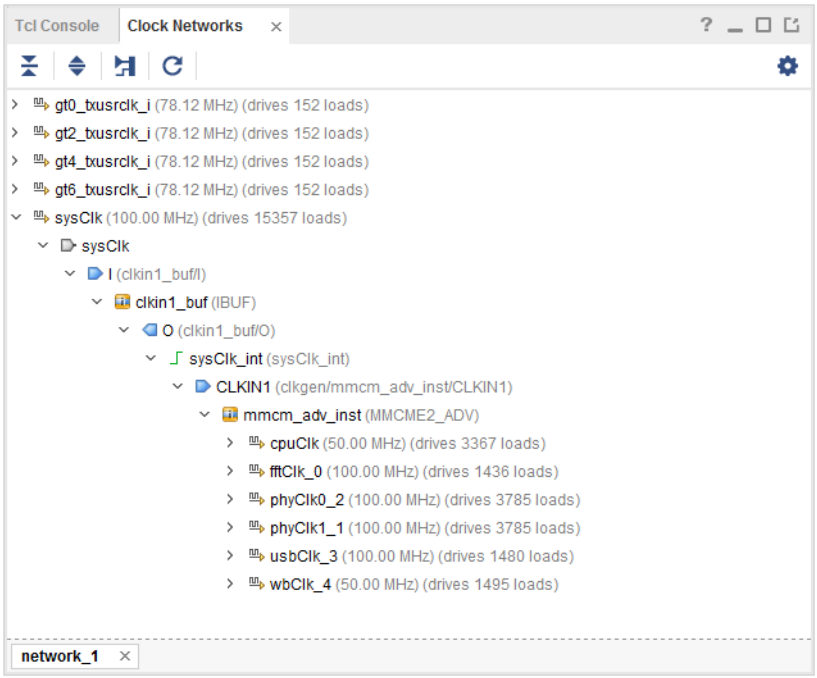



*Figure 5-6:* **Report Clock Networks**

```
### Decouple asynchronous clocks
set_clock_groups -asynchronous \
-group [get_clocks sysClk -include_generated_clocks] \
-group [get_clocks gt0_txusrclk_i -include_generated_clocks] \
-group [get_clocks gt2_txusrclk_i -include_generated_clocks] \
-group [get_clocks gt4_txusrclk_i -include_generated_clocks] \
-group [get_clocks gt6_txusrclk_i -include_generated_clocks]
```

## *Limiting I/O Constraints and Timing Exceptions*

Most timing violations are on internal paths. I/O constraints are not needed during the first baselining iterations, especially for I/O timing paths in which the launching or capturing register is located inside the I/O bank. You can add the I/O timing constraints after the design and other constraints are stable and the timing is nearly closed.

**TIP:** *Starting with the Vivado Design Suite 2015.3 release, you can use the* `config_timing_analysis -ignore_io_paths yes` *Tcl command to ignore timing on all I/O paths during implementation and in reports that use timing information. You must manually enter this command before or immediately after opening a design in memory.*

Based on recommendations of the RTL designer, timing exceptions must be limited and must not be used to hide real timing problems. Prior to this point, the false path or clock groups between clocks must be reviewed and finalized.

IP constraints must be entirely kept. When IP timing constraints are missing, known false paths might be reported as timing violations.

### *Evaluating Design WNS Before and After Each Step*

You must evaluate the design WNS after each implementation step. If you are using the Tcl command line flow, you can easily incorporate `report_timing_summary` after each implementation step in your build script. If you are using the Vivado IDE, you can use simple `tcl.post` scripts to run `report_timing_summary` after each step. In both cases, when a significant degradation in WNS is noted, you must analyze the checkpoint immediately preceding that step.

In addition to evaluating the timing for the entire design before and after each implementation step, you can take a more targeted approach for individual paths to evaluate the impact of each step in the flow on the timing. For example, the estimated net delay for a timing path after the optimization step might differ significantly from the estimated net delay for the same path after placement. Comparing the timing of critical paths after each step is an effective method for highlighting where the timing of a critical path diverges from closure.

#### Post-Synthesis and Post-Logic Optimization

Estimated net delays are close to the best possible placement for all paths. To fix violating paths try the following:

- Change the RTL.
- Use different synthesis options.
- Add timing exceptions such as multicycle paths, if appropriate and safe for the functionality in hardware.

#### Pre- and Post-Placement

After placement, the estimated net delays are close to the best possible route, except for long and medium-to-high fanout nets, which use more pessimistic delays. In addition, congestion or hold fixing impact are not accounted for in the net delays at this point, which can make the timing results optimistic.

Clock skew is accurately estimated and can be used to review imbalanced clock trees impact on slack.

You can estimate hold fixing by running min delay analysis. Large hold violations where the WHS is -0.500ns or greater between slices, block RAMs, or DSPs will need to be fixed. Small violations are acceptable and will likely be fixed by the router.

***Note:*** Paths to/from dedicated blocks like the PCIe® block can have hold time estimates greater than -0.500 ns that get automatically fixed by the router. For these cases, check `report_timing_summary` after routing to verify that all corresponding hold violations are fixed.

#### Pre- and Post-Physical Optimization

Evaluate the need for running physical optimization to fix timing problems related to:

- Nets with high fanout (`report_high_fanout_nets` shows highest fanout non-clock nets)
- Nets with drivers and loads located far apart
- DSP and block RAM with sub-optimal pipeline register usage

#### Pre and Post-Route

Slack is reported with actual routed net delays except for the nets that are not completely routed. Slack reflects the impact of hold fixing on setup and the impact of congestion.

No hold violation should remain after route, regardless of the worst setup slack (WNS) value. If the design fails hold, further analysis is needed. This is typically due to very high congestion, in which case the router gives up on optimizing timing. This can also occur for large hold violations (over 4 ns) which the router does not fix by default. Large hold violations are usually due to improper clock constraints, high clock skew or, improper I/O constraints which can already be identified after placement or even after synthesis.

If hold is met (WHS>0) but setup fails (WNS<0), follow the analysis steps described in Analyzing and Resolving Timing Violations.

### *Baselining and Timing Constraints Validation Procedure*

The following procedure helps track your progress towards timing closure and identify potential bottlenecks:

1. Open the synthesized design.
2. Run `report_timing_summary -delay_type min_max`, and record the information shown in the following table.

| | WNS | TNS | Num Failing Endpoints | WHS | THS | Num Failing Endpoints |
|---|---|---|---|---|---|---|
| Synth | | | | | | |

3. Open the post-synthesis `report_timing_summary` text report and record the `no_clock` section of `check_timing`.

   Number of missing clock requirements in the design: _____

4. Run `report_clock_networks` to identify primary clock source pins/ports in the design. (Ignore QPLLOUTCLK, QPLLOUTREFCLK because they are pulse-width only checks.)

   Number of unconstrained clocks in the design: _____

5. Run `report_clock_interaction -delay_type min_max` and sort the results by WNS path requirement.

   Smallest WNS path requirement in the design: _____

6. Sort the results of `report_clock_interaction` by WHS to see if there are large hold violations (>500 ps) after synthesis.

   Largest negative WHS in the design: _____

7. Sort results of `report_clock_interaction` by Inter-Clock Constraints and list *all* the clock pairs that show up as unsafe:

8. Upon opening the synthesized design, how many CRITITCAL_WARNINGS exist?

   Number of synthesized design CRITICAL WARNINGS: _____

9. What types of CRITICAL WARNINGS exist?

   Record examples of each type.

10. Run `report_high_fanout_nets -timing -load_types -max_nets 25`.

    Number of high fanout nets NOT driven by FF: _____

    Number of loads on highest fanout net NOT driven by FF: _____

    Do any high fanout nets have negative slack?- If yes, WNS = _____

11. Implement the design. After each step, run `report_timing_summary` and record the information shown in the following table.

| | WNS | TNS | Num Failing Endpoints | WHS | THS | Num Failing Endpoints |
|---|---|---|---|---|---|---|
| Opt | | | | | | |
| Place | | | | | | |
| Physopt | | | | | | |
| Route | | | | | | |

Run `report_exceptions -ignored` to identify if there are constraints that overlap in the design. Record the results.

# Analyzing and Resolving Timing Violations

The timing-driven algorithms focus on the worst violations for each clock domain. Understanding and fixing problems related to the worst violation for each clock usually resolves most smaller violations when you rerun the implementation flow. You must identify the main timing characteristic contributing to each violation and then apply the corresponding resolution techniques described throughout this chapter.

The following figure shows the basic process for analyzing and resolving timing violations.

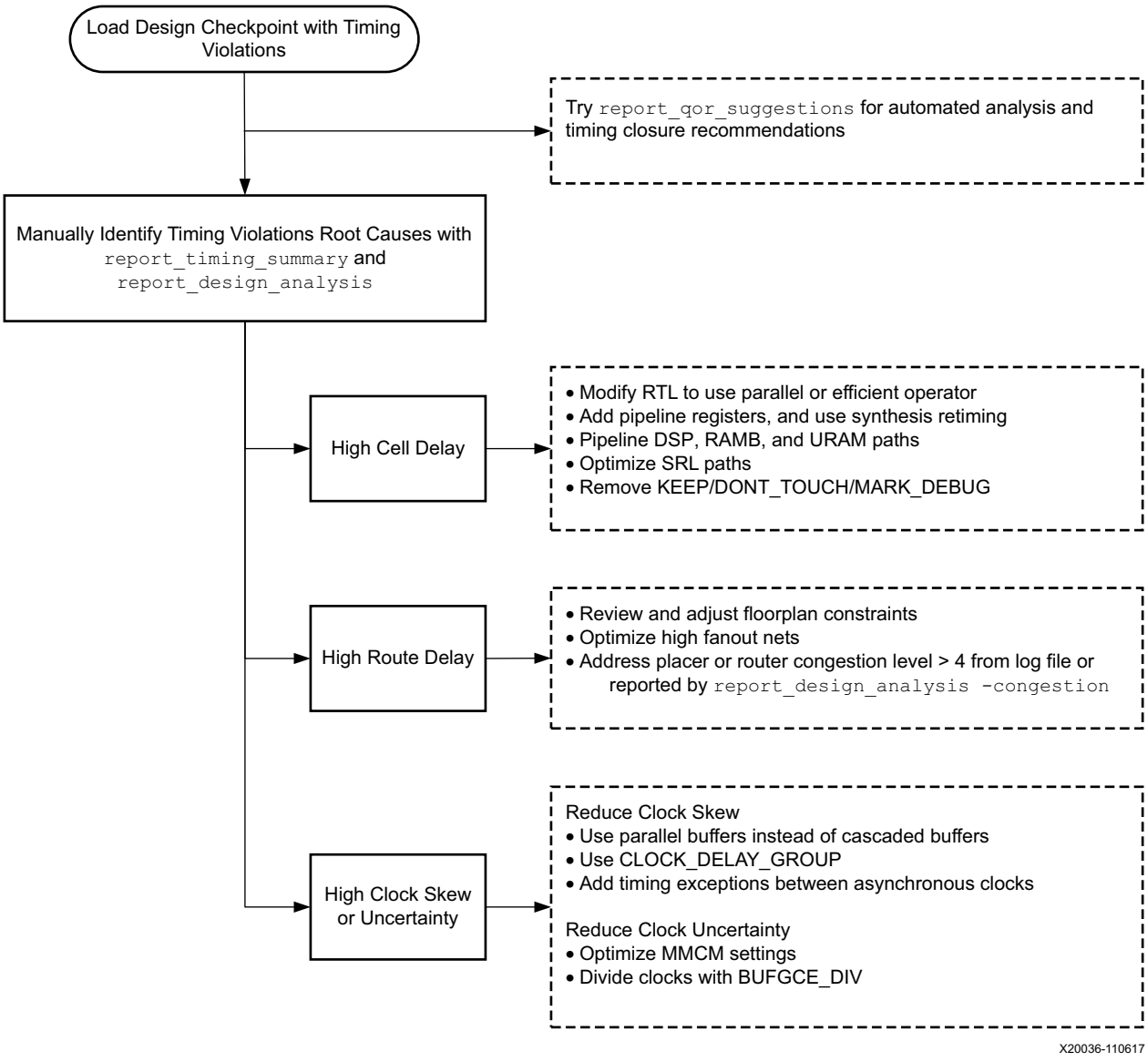



*Figure 5-7:* **Analyzing and Resolving Timing Violations**

# Identifying Timing Violations Root Cause

For setup, you must first analyze the worst violation of each clock group. A clock group refers to all intra, inter, and asynchronous paths captured by a given clock.

For hold, all violations must be reviewed as follows:

- Before routing, review only violations over 0.5ns.
- After routing, start with the worst violation.

## *Reviewing Timing Slack*

Several factors can impact the setup and hold slacks. You can easily identify each factor by reviewing the setup and hold slack equations when written in the following simplified form:

| | | |
|---|---|---|
| Slack (setup/recovery) | = | setup path requirement<br>- datapath delay(max)<br>+ clock skew<br>- clock uncertainty<br>- setup/recovery time |
| Slack (hold/removal) | = | hold path requirement<br>+ datapath delay(min)<br>- clock skew<br>- clock uncertainty<br>- hold/removal time |

For timing analysis, clock skew is always calculated as follows:

```
Clock Skew = destination clock delay - source clock delay (after the common node if any)
```

During the analysis of the violating timing paths, you must review the relative impact of each variable to determine which variable contributes the most to the violation. Then you can start analyzing the main contributor to understand what characteristic of the path influences its value the most and try to identify a design or constraint change to reduce its impact. If a design or constraint change is not practical, you must do the same analysis with all other contributors starting with the worst one. The following list shows the typical contributor order from worst to least.

For setup/recovery:

- Datapath delay: Subtract the timing path requirement from the datapath delay. If the difference is comparable to the (negative) slack value, then either the path requirement is too tight or the datapath delay is too large.
- Datapath delay + setup/recovery time: Subtract the timing path requirement from the datapath delay plus the setup/recovery time. If the difference is comparable to the

(negative) slack value, then either the path requirement is too tight or the setup/recovery time is larger than usual and noticeably contributes to the violation.

- Clock skew: If the clock skew and the slack have similar negative values and the skew absolute value is over a few 100 ps, then the skew is a major contributor and you must review the clock topology.
- Clock uncertainty: If the clock uncertainty is over a few 100 ps, then you must review the clock topology and jitter numbers to understand why the uncertainty is so high.

For hold/removal:

- Clock skew: If the clock skew is over 300 ps, you must review the clock topology.
- Clock uncertainty: If the clock uncertainty is over 200 ps, then you must review the clock topology and jitter numbers to understand why the uncertainty is so high.
- Hold/removal time: If the hold/removal time is over a few 100 ps, you can review the primitive data sheet to validate that this is expected.
- Hold path requirement: The requirement is usually zero. If not, you must verify that your timing constraints are correct.

Assuming all timing constraints are accurate and reasonable, the most common contributors to timing violations are usually the datapath delay for setup/recovery timing paths, and skew for hold/removal timing paths. At the early stage of a design cycle, you can fix most timing problems by analyzing these two contributors. However, after improving and refining design and constraints, the remaining violations are caused by a combination of factors, and you must review all factors in parallel to identify which to improve.

See this link for more information on timing analysis concepts, and see this link for more information on timing reports (`report_timing_summary/report_timing`) in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21].

### *Using the Design Analysis Report*

When timing closure is difficult to achieve or when you are trying to improve the overall performance of your application, you must review the main characteristics of your design after running synthesis and after any step of the implementation flow. The QoR analysis usually requires that you look at several global and local characteristics at the same time to determine what is suboptimal in the design and the constraints, or which logic structure is not suitable for the target device architecture and implementation tools. The `report_design_analysis` command gathers logical, timing, and physical characteristics in a few tables to simplify the QoR root cause analysis.

***Note:*** The `report_design_analysis` command does not report on the completeness and correctness of timing constraints. See Checking That Your Design is Properly Constrained for more information on reviewing and fixing timing constraints.

**TIP:** *Run the Design Analysis Report in the Vivado IDE for improved visualization, automatic filtering, and convenient cross-probing.*

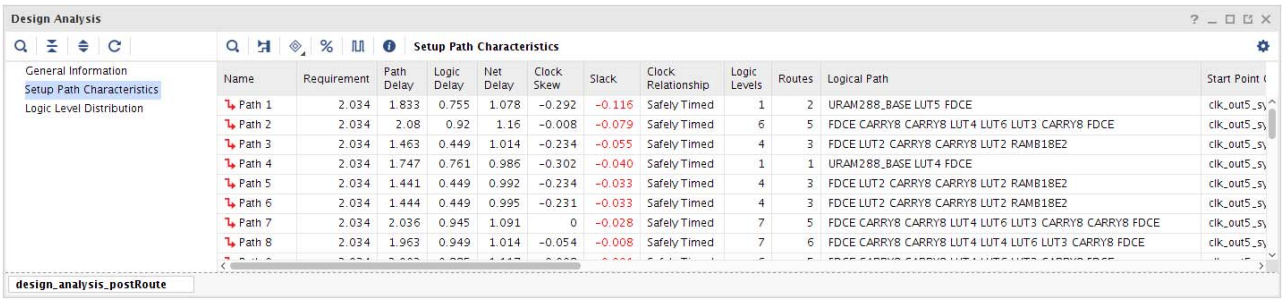
The following sections only cover timing path characteristics analysis. The Design Analysis report also provides useful information about congestion and design complexity, as described in Identifying Congestion.

### Analyze Path Characteristics

You can use the following command to report the 50 worst setup timing paths:

```
report_design_analysis -max_paths 50 -setup -name design_analysis_postRoute
```

The following figure shows an example of the Setup Path Characteristics table generated by this command. To see additional columns in the window, scroll horizontally.




| Name | Requirement | Path Delay | Logic Delay | Net Delay | Clock Skew | Slack | Clock Relationship | Logic Levels | Routes | Logical Path | Start Point |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | 2.034 | 1.833 | 0.755 | 1.078 | -0.292 | -0.116 | Safely Timed | 1 | 2 | URAM288_BASE LUT5 FDCE | clk_out5_s |
| Path 2 | 2.034 | 2.08 | 0.92 | 1.16 | -0.008 | -0.079 | Safely Timed | 6 | 5 | FDCE CARRY8 CARRY8 LUT4 LUT6 LUT3 CARRY8 FDCE | clk_out5_s |
| Path 3 | 2.034 | 1.463 | 0.449 | 1.014 | -0.234 | -0.055 | Safely Timed | 4 | 3 | FDCE LUT2 CARRY8 CARRY8 LUT2 RAMB18E2 | clk_out5_s |
| Path 4 | 2.034 | 1.747 | 0.761 | 0.986 | -0.302 | -0.040 | Safely Timed | 1 | 1 | URAM288_BASE LUT4 FDCE | clk_out5_s |
| Path 5 | 2.034 | 1.441 | 0.449 | 0.992 | -0.234 | -0.033 | Safely Timed | 4 | 3 | FDCE LUT2 CARRY8 CARRY8 LUT2 RAMB18E2 | clk_out5_s |
| Path 6 | 2.034 | 1.444 | 0.449 | 0.995 | -0.231 | -0.033 | Safely Timed | 4 | 3 | FDCE LUT2 CARRY8 CARRY8 LUT2 RAMB18E2 | clk_out5_s |
| Path 7 | 2.034 | 2.036 | 0.945 | 1.091 | 0 | -0.028 | Safely Timed | 7 | 5 | FDCE CARRY8 CARRY8 LUT4 LUT6 LUT3 CARRY8 CARRY8 FDCE | clk_out5_s |
| Path 8 | 2.034 | 1.963 | 0.949 | 1.014 | -0.054 | -0.008 | Safely Timed | 7 | 6 | FDCE CARRY8 CARRY8 LUT4 LUT4 LUT6 LUT3 CARRY8 FDCE | clk_out5_s |

*Figure 5-8:* **Report Design Analysis Timing Path Characteristics Post-Route**

Following are tips for working with this table:

- Toggle between numbers and % by clicking on the "%" (Show Percentage) button. This is particularly helpful to review proportion of cell delay and net delay.
- By default, columns with only null or empty values are hidden. Click on the "Hide Unused" button to turn off filtering and show all columns, or right click on the table header to select which columns to show or hide.

From this table, you can isolate which characteristics are introducing the timing violation for each path:

- High logic delay percentage (Logic Delay)
  - Are there many levels of logic? (LOGIC_LEVELS)
  - Are there any constraints or attributes that prevent logic optimization? (DONT_TOUCH, MARK_DEBUG)
  - Does the path include a cell with high logic delay such as block RAM or DSP? (Logical Path, Start Point Pin Primitive, End Point Pin Primitive)
  - Is the path requirement too tight for the current path topology? (Requirement)
- High net delay percentage (Net Delay)
  - Are there any high fanout nets in the path? (High Fanout, Cumulative Fanout)
  - Are the cells assigned to several Pblocks that can be placed far apart? (Pblocks)
  - Are the cells placed far apart? (Bounding Box Size, Clock Region Distance)
  - For SSI technology devices, are there nets crossing SLR boundaries? (SLR Crossings)
  - Are one or several net delay values a lot higher than expected while the placement seems correct? Select the path and visualize its placement and routing in the Device window.
  - Is there a missing pipeline register in a block RAM or DSP cell? (Comb DSP, MREG, PREG, DOA_REG, DOB_REG)
- High skew (<-0.5 ns for setup and >0.5 ns for hold) (Clock Skew)
  - Is it a clock domain crossing path? (Start Point Clock, End Point Clock)
  - Are the clocks synchronous or asynchronous? (Clock Relationship)
  - Is the path crossing I/O columns? (IO Crossings)

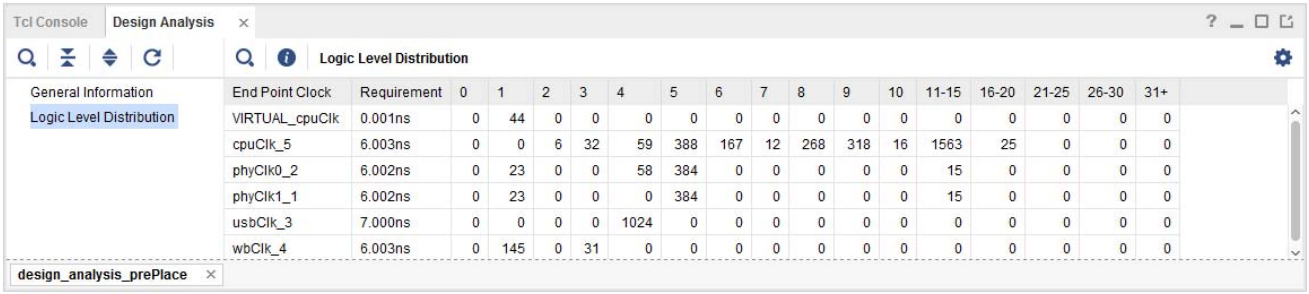
**TIP:** *For visualizing the details of the timing paths in the Xilinx Vivado IDE, select the path in the table, and go to the Properties tab.*

### Review the Logic Level Distribution

The `report_design_analysis` command also generates a Logic Level Distribution table for the worst 1000 paths (default) that you can use to identify the presence of longer paths in the design. The longest paths are usually optimized first by the placer in order to meet timing, which will potentially degrade the placement quality of shorter paths. You must always try to eliminate the longer paths to improve the overall QoR. For this reason, Xilinx recommends reviewing the longest paths before placement.

The following figure shows an example of the Logic Level Distribution for a design where the worst 5000 paths include difficult paths with 17 logic levels while the clock period is 7.5 ns. Run the following command to obtain this report:

```
report_design_analysis -logic_level_distribution -logic_level_dist_paths 5000 -name
design_analysis_prePlace
```




| End Point Clock | Requirement | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11-15 | 16-20 | 21-25 | 26-30 | 31+ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| VIRTUAL_cpuClk | 0.001ns | 0 | 44 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| cpuClk_5 | 6.003ns | 0 | 0 | 6 | 32 | 59 | 388 | 167 | 12 | 268 | 318 | 16 | 1563 | 25 | 0 | 0 | 0 |
| phyClk0_2 | 6.002ns | 0 | 23 | 0 | 0 | 58 | 384 | 0 | 0 | 0 | 0 | 0 | 15 | 0 | 0 | 0 | 0 |
| phyClk1_1 | 6.002ns | 0 | 23 | 0 | 0 | 0 | 384 | 0 | 0 | 0 | 0 | 0 | 15 | 0 | 0 | 0 | 0 |
| usbClk_3 | 7.000ns | 0 | 0 | 0 | 0 | 1024 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| wbClk_4 | 6.003ns | 0 | 145 | 0 | 31 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Figure 5-9:* **Report Design Analysis Timing Path Characteristics Pre-Place**

For logic levels above 10, you can use the `-min_level` and `-max_level` options to provide more distribution information for paths between the min and max level you identify. For example:

```
report_design_analysis -logic_level_distribution -min_level 16 -max_level 20
-logic_level_dist_paths 5000 -name design_analysis_1
```

Run the following command to generate the timing report of the longest paths:

```
report_timing -name longPaths -of_objects [get_timing_paths -setup -to [get_clocks
cpuClk_5] -max_paths 5000 -filter {LOGIC_LEVELS>=16 && LOGIC_LEVELS<=20}]
```

Based on what you find, you can improve the netlist by changing the RTL or using different synthesis options, or you can modify the timing and physical constraints.

## *Datapath Delay and Logic Levels*

In general, the number of LUTs and other primitives in the path is most important factor in contributing to the delay. Because LUT delays are reported differently in 7 series and UltraScale devices, separate cell delay and route delay ranges must be considered, as explained below.

If the path delay is dominated by:

- **Cell delay is >25% in 7 series devices and >50% in UltraScale devices**

  Can the path be modified to be shorter or to use faster logic cells? See Reducing Logic Delay.

- **Route delay is >75% in 7 series devices and >50% in UltraScale devices**

  Was this path impacted by hold fixing? You can determine this by running `report_design_analysis -show_all` and examining the **Hold Detour** column. Use the corresponding analysis technique.

  - Yes - Is the impacted net part of a CDC path?
    - Yes - Is the CDC path missing a constraint?
    - No - Do the startpoint and endpoint of that hold-fixed path use a balanced clock tree? Look at the skew value.
  - No - See the following information on congestion.

  Was this path impacted by congestion? Look at each individual net delay, the fanout and observe the routing in the Device view with routing details enabled (post-route analysis only). You can also turn on the congestion metrics to see if the path is located in or near a congested area. Use the following analysis steps for a quick assessment or review Identifying Congestion for a comprehensive analysis.

  - Yes - For the nets with the highest delay value, is the fanout low (<10)?
    - Yes - If the routing seems optimal (straight line) but driver and load are far apart, the sub-optimal placement is related to congestion. Review Addressing Congestion to identify the best resolution technique.
    - No - Try to use physical logic optimization to duplicate the driver of the net. Once duplicated, each driver can automatically be placed closer to its loads, which will reduce the overall datapath delay. Review Optimizing High Fanout Nets for more details and to learn about alternate techniques.
  - No - The design is spread out too much. Try one of the following techniques to improve the placement:
    - Reducing Control Sets
    - Tuning the Compilation Flow
    - Considering Floorplan

### *Clock Skew and Uncertainty*

Xilinx devices use various types of routing resources to support most common clocking schemes and requirements such as high fanout clocks, short propagation delays, and

extremely low skew. Clock skew affects any register-to-register path with either a combinational logic or interconnect between them.

**RECOMMENDED:** *Run a design analysis report (`report_design_analysis`) to generate a timing report, which includes information on clock skew data. Verify that the clock nets do not contain excessive clock skew.*

Clock skew in high performance clock domains (+300 MHz) can impact performance. In general, the clock skew should be no more than 500 ps. For example, 500 ps represents 15% of a 300 MHz clock period, which is equivalent to the timing budget of 1 or 2 logic levels. In cross domain clock paths the skew can be higher, because the clocks use different resources and the common node is located further up the clock trees. SDC-based tools time all clocks together unless constraints specify that they should not be, for example:

```
set_clock_groups/set_false_path/set_max_delay -datapath_only)
```

If you suspect high clock skew, review Reducing Clock Skew and Reducing Clock Uncertainty.

## Reducing Logic Delay

Vivado implementation focuses on the most critical paths first, which often makes less difficult paths become critical after placement or after routing. Xilinx recommends identifying and improving the longest paths after synthesis or after `opt_design`, because it will have the biggest impact on QoR and will usually dramatically reduce the number of place and route iterations to reach timing closure.

Before placement, timing analysis uses estimated delays that correspond to ideal placement and typical clock skew. By using `report_timing`, `report_timing_summary`, or `report_design_analysis`, you can quickly identify the paths with too many logic levels or with high cell delays, because they usually fail timing or barely meet timing before placement. Use the methodology proposed in Identifying Timing Violations Root Cause to find the long paths which need to be improved before implementing the design.

### Optimizing Regular Fabric Paths

Regular fabric paths are paths between fabric registers or shift registers, and traversing a mix of LUTs, MUXFs and CARRYs. The report_design_analysis Timing Path Characteristics table provides the best logic path topology summary, where the following issues can be identified:

- Several small LUTs are cascaded: mapping to LUTs is impacted by hierarchy, presence of KEEP_HIERARCHY, DONT_TOUCH or MARK_DEBUG attributes, intermediate signals with some fanout (10 and higher). Try running the `opt_design -remap` option or the AddRemap directive to try collapsing smaller LUTs and reduce the number of logic levels. If `opt_design` is not able to optimize the longest paths, analyze the path in the Elaborated view and restructure the RTL description.
- Single CARRY cell is present in the path. CARRY primitives are most beneficial for QoR when cascaded. CARRY cells are more difficult to place than LUTs, and forcing synthesis to use LUTs rather than a single CARRY allows for better LUTs structuring and more flexible placement in many cases. Try the FewerCarryChains synthesis directive or the PerfThresholdCarry strategy (project mode only) to eliminate most single CARRY cells.
- Path ends at shift register (SRL): pull the first register out of the shift register by using the SRL_STYLE attribute in RTL. See this link in *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 16] for more details.
- Path ends at a fabric register (FD) clock enable or synchronous set/reset: if the path ending at the data pin (D) has more margin and fewer logic levels, use the EXTRACT_ENABLE or EXTRACT_RESET attribute and set it to "no" on the signal in RTL. See Pushing the Logic from the Control Pin to the Data Pin in Chapter 3 section for more details.

**TIP:** *To cross-probe from a post-synthesis path to the corresponding RTL view and source code, see this link in the Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906) [Ref 21].*

### Optimizing Paths with Dedicated Blocks and Macro Primitives

Paths from/to/between dedicated blocks and macro primitives, such as DSP, block RAM, FIFO or GT_CHANNEL, need special attention as these primitives usually have the following timing characteristics:

- Higher setup/hold/clock-to-output timing arc values for some pins. For example, a block RAM has a clock-to-output delay around 1.5 ns without the optional output register and 0.4 ns with the optional output register. Review the data sheet of your target device architecture for complete details.
- Higher routing delays than regular FD/LUT connections
- Higher clock skew variation than regular FD-FD paths

Also, their availability and site locations are restricted compared to CLB slices, which usually makes their placement more challenging and often incurs some QoR penalty.

For these reasons, Xilinx recommends:

- Pipelining paths from and to Dedicated Blocks and Macro Primitives as much as possible
- Restructuring the combinational logic connected to these cells to reduce the logic levels by at least 1 or 2 cells if latency incurred by pipelining is a concern
- Meeting setup timing by at least 500 ps on these paths before placement
- Replicating cones of logic connected to too many Dedicated Blocks or Macro Primitives if they need to be placed far apart

## Reducing Net Delay

Sub-optimal net delay is the consequence of a challenging design or inappropriate constraints on the placer and router algorithms. The most common root causes are:

- Presence of physical constraints forcing logic to be placed far apart
- Lower placement quality (spread logic) due to very high device utilization
- Difficulty to route high fanout nets
- Presence of congested area due to the netlist complexity and device obstacles, such as I/O columns and device boundaries

The following sections present several analysis and resolution techniques.

### *Reviewing Physical Constraints*

All designs come with a minimum set of physical constraints, especially for I/O location, and sometimes for clocking and logic placement. While I/O location cannot be modified at the time of the design is ready for timing closure, physical constraints such Pblocks and LOC must be analyzed. Use the `report_design_analysis` Timing Path Characteristics table to identify the presence of several Pblocks constraints on each critical path.

In the Vivado IDE Properties window, you can select the path in the Timing Path Characteristic table to review which Pblocks are constraining cells in the path. Consider removing one or several Pblock constraints if the constraints force logic spreading.

### *Identifying Congestion*

Device congestion can potentially lead to difficult timing closure if the critical paths are placed inside or next to a congested area or if the device utilization is high and the placed design is hardly routable. In many cases, congestion will significantly increase the router runtime. If a path shows routed delays that are longer than expected, Xilinx recommends

analyzing the congestion of the design and identifying the best congestion alleviation technique.

### Congestion Area and Level Definition

Xilinx FPGA routing architecture comprises interconnect resources of various lengths in each direction: North, South, East and West. A congested area is reported as the smallest square that covers adjacent interconnect tiles (INT_XnYm) or CLB tiles (CLE_M_XnYm) where interconnect resource utilization in a specific direction is close to or over 100%. The congestion level is the positive integer which corresponds to the side length of the square. The following figure shows the relative size of congestion areas on an UltraScale device versus clock regions.

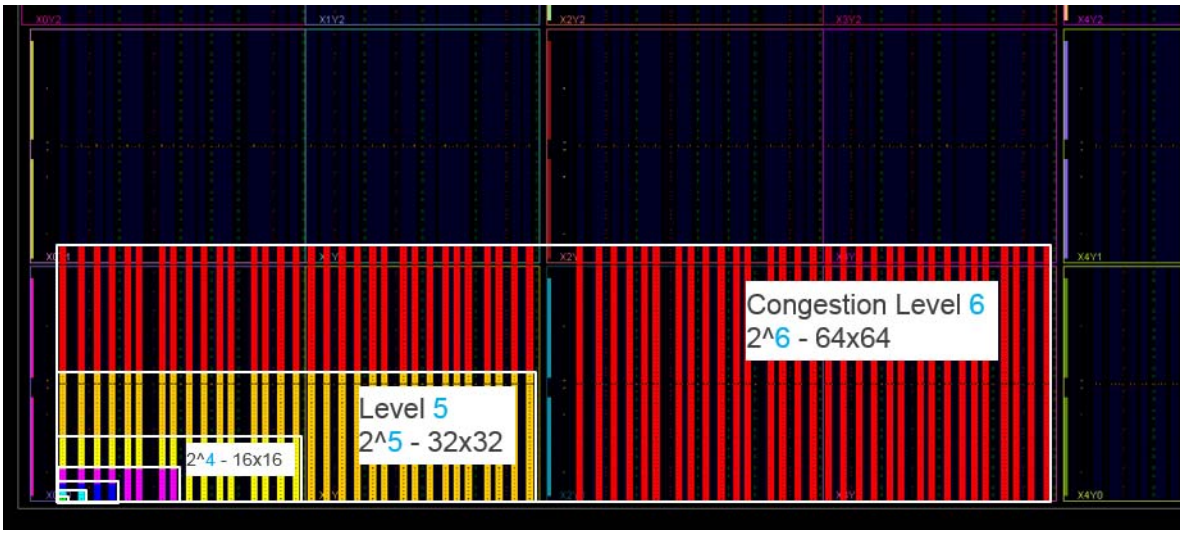



*Figure 5-10:* **Congestion Levels and Areas in an UltraScale Device View**

### Congestion Level Ranges

When analyzing congestion, the level reported by the tools can be categorized as shown in the following table.

***Note:*** Congestion levels of 5 or higher often impact QoR and always lead to longer router runtime.

*Table 5-4:* **Congestion Level Ranges**

| Level | Area | Congestion | QoR Impact |
|---|---|---|---|
| 1, 2 | 2x2, 4x4 | None | None |
| 3, 4 | 8x8, 16x16 | Mild | Possible QoR degradation |
| 5 | 32x32 | Moderate | Likely QoR degradation |
| 6 | 64x64 | High | Difficulty routing |
| 7, 8 | 128x128, 256x256 | Impossible | Likely unroutable |

### Routing Congestion per CLB in the Device Window

The Routing Congestion per CLB is based on estimation and not actual routing. After placement or after routing, you can display this congestion metric by right-clicking in the Device window, selecting **Metric**, and then selecting **Vertical and Horizontal Routing Congestion per CLB**. This provides a quick visual overview of any congestion hotspots in the device. The following figure shows a placed design with several congested areas due to high utilization and netlist complexity.

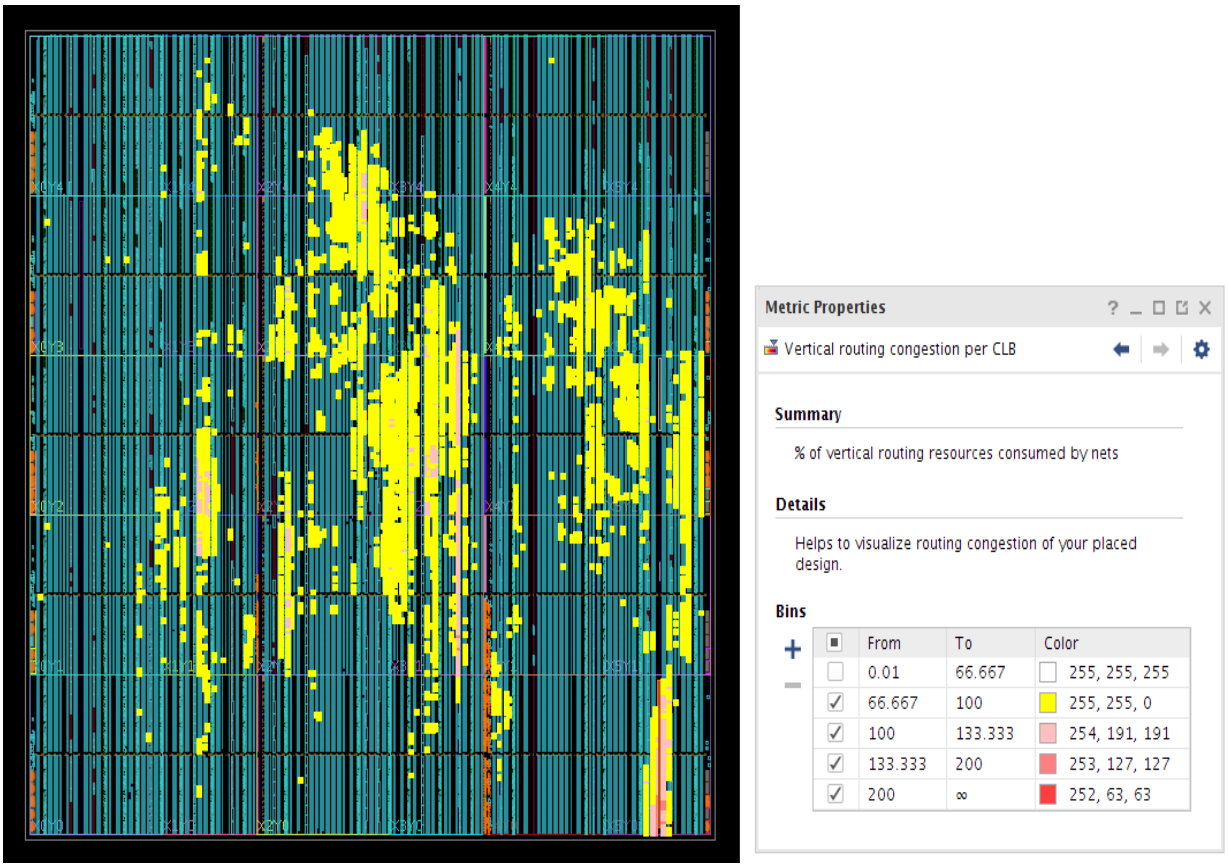



*Figure 5-11:* **Example of Congestion in the Device Window**

### Congestion in the Placer Log

The placer estimates congestion throughout the placement phases and spreads the logic in congested areas. This helps reducing the interconnect utilization to improve routability, and also the estimated versus routed delays correlation. However, when the congestion cannot be reduced due to high utilization or other reasons, the placer does not print congestion details but issues the following warning:

```
WARNING: [Place 46-14] The placer has determined that this design is highly congested
and may have difficulty routing. Run report_design_analysis -congestion for a
detailed report.
```

In that case the QoR is very likely impacted and it is prudent to address the issues causing the congestion before continuing on to the router. As stated in the message, use the `report_design_analysis` command to report the actual congestion levels, as well as identify their location and the logic placed in the same area.

### Congestion in the Router Log

The router issues additional messages depending on the congestion level and the difficulty to route certain resources. The router also prints several intermediate timing summaries. The first one comes after routing all the clocks and usually shows WNS/TNS/WHS/TNS numbers similar to post-place timing analysis. The next router intermediate timing summary is reported after initial routing. If the timing has degraded significantly, the QoR has been impacted by hold fixing and/or congestion.

When congestion level is 4 or higher, the router prints an initial estimated congestion table which gives more details on the nature of the congestion:

- Global Congestion is similar to how the placer congestion is estimated and is based on all types of interconnects.
- Long Congestion only considers long interconnect utilization for a given direction.
- Short Congestion considers all other interconnect utilization for a given direction.

Any congestion area greater than 32x32 (level 5) will likely impact QoR and routability (highlighted in yellow in the table below). Congestion on Long interconnects increases usage of Short interconnects which results in longer routed delays. Congestion on Short interconnects usually induce longer runtimes and if their tile % is more than 5%, it will also likely cause QoR degradation (highlighted in red in the table below).

```
INFO: [Route 35-449] Initial Estimated Congestion
```

| | Global Congestion | | Long Congestion | | Short Congestion | |
|---|---|---|---|---|---|---|
| Direction | Size | % Tiles | Size | % Tiles | Size | % Tiles |
| NORTH | 16x16 | 1.95 | 32x32 | 1.68 | 32x32 | 11.58 |
| SOUTH | 8x8 | 1.90 | 16x16 | 2.00 | 32x32 | 9.23 |
| EAST | 8x8 | 0.93 | 2x2 | 0.20 | 32x32 | 9.14 |
| WEST | 8x8 | 1.37 | 2x2 | 0.15 | 32x32 | 14.50 |

*Figure 5-12:* **Initial Estimated Congestion Table**

During Global Iterations, the router first tries to find a legal solution with no overlap and also meet timing for both setup and hold, with higher priority for hold fixing. When the

router does not converge during a global iteration, it stops optimizing timing until a valid routed solution has been found, as shown on the example below:

```
Phase 4.1 Global Iteration 0
 Number of Nodes with overlaps = 1157522
 Number of Nodes with overlaps = 131697
 Number of Nodes with overlaps = 28118
 Number of Nodes with overlaps = 10971
 Number of Nodes with overlaps = 7324
WARNING: [Route 35-447] Congestion is preventing the router from routing all nets.
The router will prioritize the successful completion of routing all nets over timing
optimizations.
```

After a valid routed solution has been found, timing optimizations are re-enabled.

The route also flags CLB routing congestion and provides the name of the top most congested CLBs. See INFO message below.

```
INFO: [Route 35-443] CLB routing congestion detected. Several CLBs have high routing
utilization, which can impact timing closure. Top ten most congested CLBs are:
CLEL_L_X29Y384 CLEL_R_X29Y384 CLE_M_X43Y107 CLEL_R_X43Y107 CLEL_L_X31Y389
CLEL_R_X31Y389
```

Finally, when the router cannot find a legally routed solution, several Critical Warning messages, as shown below, indicate the number of nets that are not fully routed and the number of interconnect resources with overlaps.

```
CRITICAL WARNING: [Route 35-162] 44084 signals failed to route due to routing
congestion. Please run report_route_status to get a full summary of the design's
routing.
…
CRITICAL WARNING: [Route 35-2] Design is not legally routed. There are 91566 node
overlaps.
```

**TIP:** *During routing, nets are spread around the congested areas, which usually reduces the final congestion level reported in the log file when the design is successfully routed.*

### Report Design Analysis Congestion Report

To help you identify congestion, the Report Design Analysis command allows you to generate a congestion report that shows the congested areas of the device and the name of design modules present in these areas. The congestion tables in the report show the congested area seen by the placer and router algorithms. The following figure shows an example of the congestion table.

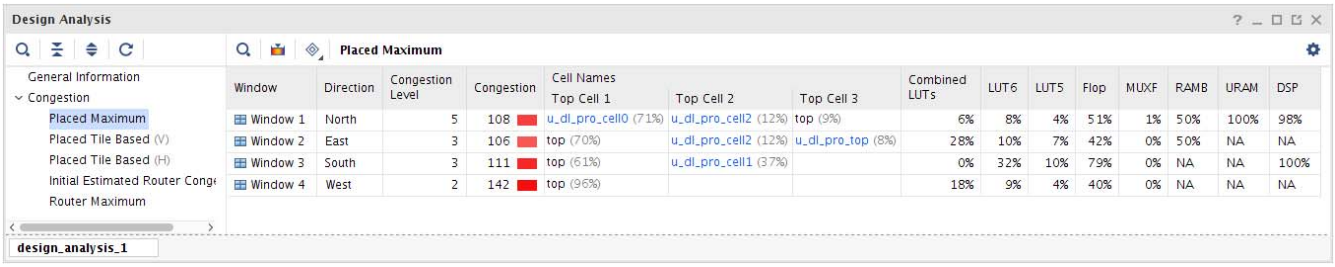



*Figure 5-13:* **Congestion Table**

The Placed Maximum, Initial Estimated Router Congestion, and Router Maximum congestion tables provide information on the most congested areas in the North, South, East, and West direction. When you select a window in the table, the corresponding congested area is highlighted in the Device window. For information on the congestion and QoR impact, see Table 5-4.

The tables show the congestion at different stages of the design flow:

- Placed Maximum: Shows congestion based on the location of the cells and a model of routing.
- Initial Estimated Router Congestion: Shows congestion after a quick router iteration. This is the most useful stage to analyze congestion because it gives an accurate picture of congestion due to placement.
- Router Maximum: Shows congestion after the router has worked extensively to reduce congestion.

The Congestion percentages in the Congestion Table show the routing utilization in the congestion window. The top three hierarchical cells located in the congested window are listed and can be selected and cross-probed to the Device window or Schematic window. The cell utilization percentage in the congestion window is also shown.

With the hierarchical cells present in the congested area identified, you can use the congestion alleviating techniques discussed later in this guide to try reducing the overall design congestion.

For more information on generating and analyzing the Report Design Analysis Congestion report, see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21].

### Report Design Analysis Complexity Report

The Complexity Report shows the Rent Exponent, Average Fanout, and distribution per type of leaf cells for the top-level design and/or for hierarchical cells. The Rent exponent is the relationship between the number of ports and the number of cells of a netlist partition when recursively partitioning the design with a min-cut algorithm. It is computed with similar algorithms as the ones used by the placer during global placement. Therefore, it can

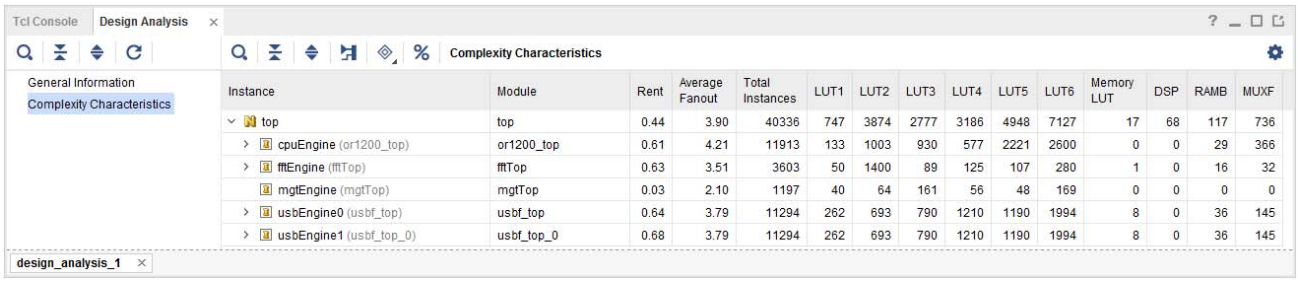
provide a good indication of the challenges seen by the placer, especially when the hierarchy of the design matches well the physical partitions found during global placement.

A design with higher Rent exponent corresponds to a design where the groups of highly connected logic also have strong connectivity with other groups. This usually translates into a higher utilization of global routing resources and an increased routing complexity. The Rent exponent provided in this report is computed on the unplaced and unrouted netlist. After placement, the Rent exponent of the same design can differ as it is based on physical partitions instead of logical partitions.

Report Design Analysis runs in Complexity Mode when you do either of the following:

- Check the Complexity option in the Report Design Analysis dialog box Options tab.
- Execute the `report_design_analysis` Tcl command with the `-complexity` option.

The following figure shows the Complexity Report.




| Instance | Module | Rent | Average Fanout | Total Instances | LUT1 | LUT2 | LUT3 | LUT4 | LUT5 | LUT6 | Memory LUT | DSP | RAMB | MUXF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| top | top | 0.44 | 3.90 | 40336 | 747 | 3874 | 2777 | 3186 | 4948 | 7127 | 17 | 68 | 117 | 736 |
| cpuEngine (or1200_top) | or1200_top | 0.61 | 4.21 | 11913 | 133 | 1003 | 930 | 577 | 2221 | 2600 | 0 | 0 | 29 | 366 |
| fftEngine (fftTop) | fftTop | 0.63 | 3.51 | 3603 | 50 | 1400 | 89 | 125 | 107 | 280 | 1 | 0 | 16 | 32 |
| mgtEngine (mgtTop) | mgtTop | 0.03 | 2.10 | 1197 | 40 | 64 | 161 | 56 | 48 | 169 | 0 | 0 | 0 | 0 |
| usbEngine0 (usbf_top) | usbf_top | 0.64 | 3.79 | 11294 | 262 | 693 | 790 | 1210 | 1190 | 1994 | 8 | 0 | 36 | 145 |
| usbEngine1 (usbf_top_0) | usbf_top_0 | 0.68 | 3.79 | 11294 | 262 | 693 | 790 | 1210 | 1190 | 1994 | 8 | 0 | 36 | 145 |

*Figure 5-14:* **Complexity Report**

The following table shows the typical ranges for the Rent Exponent.

*Table 5-5:* **Rent Exponent Ranges**

| Range | Meaning |
|---|---|
| 0.0 to 0.65 | This range is low to normal. |
| 0.65 to 0.85 | This range is high, especially when the total number of instances is above 15,000. |
| Above 0.85 | This range is very high, indicating that the design might fail during implementation if the number of instances is also high. |

The following table shows the typical ranges for the Average Fanout.

*Table 5-6:* **Average Fanout Ranges**

| Range | Meaning |
|---|---|
| Below 4 | This range is normal. |
| 4 to 5 | This range is high, indicating that placing the design without congestion might be difficult.<br>***Note:*** When using SSI technology devices, if the total number of instances is above 100,000, it might be difficult for the placer to find a solution that fits in 1 SLR or is spread over 2 SLRs. |
| Above 5 | This range is very high, indicating that the design might fail during implementation. |

You must treat high Rent exponents and high Average Fanouts for larger modules with higher importance. Smaller modules, especially under 15,000 total instances, can have high Rent exponent and high Average Fanout and still be easy to place and route successfully. Therefore, you must review the Total Instances column along with the Rent exponent and Average Fanout.

**TIP:** *Top-level modules do not necessarily have high complexity metrics even though some of the lower-level modules have high Rent exponents and high Average Fanouts. Use the* `-hierarchical_depth` *option to refine the analysis to include the lower-level modules.*

For more information on generating and analyzing the Report Design Analysis Complexity report see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21].

## Reducing Clock Skew

To meet requirements such as high fanout clocks, short propagation delays, and low clock skew, Xilinx devices use dedicated routing resources to support most common clocking schemes. Clock skew can severely reduce timing budget on high frequency clocks. Clock skew can also add excessive stress on implementation tools to meet both setup and hold when the device utilization is high.

The clock skew is typically less than 300 ps for intra-clock timing paths and less than 500 ps for timing paths between balanced synchronous clocks. When crossing I/O columns or SLR boundaries, clock skew shows more variation, which is reflected in the timing slack and optimized by the implementation tools. For timing paths between unbalanced clock trees or with no common node, clock skew can be several nanoseconds, making timing closure almost impossible.

To reduce clock skew:

1. Review all clock relationships to ensure that only synchronous clock paths are timed and optimized, as described in Defining Clock Groups and CDC Constraints in Chapter 3.
2. Review the clock tree topologies and placement of timing paths impacted by higher clock skew than expected, as described in the following sections.
3. Identify the possible clock skew reduction techniques, as described in the following sections.

### *Using Intra-Clock Timing Paths*

Timing paths with the same source and destination clocks that are driven by the same clock buffer typically exhibit very low skew. This is because the common node is located on the dedicated clock network, close to the leaf clock pins, as shown in the following figure.

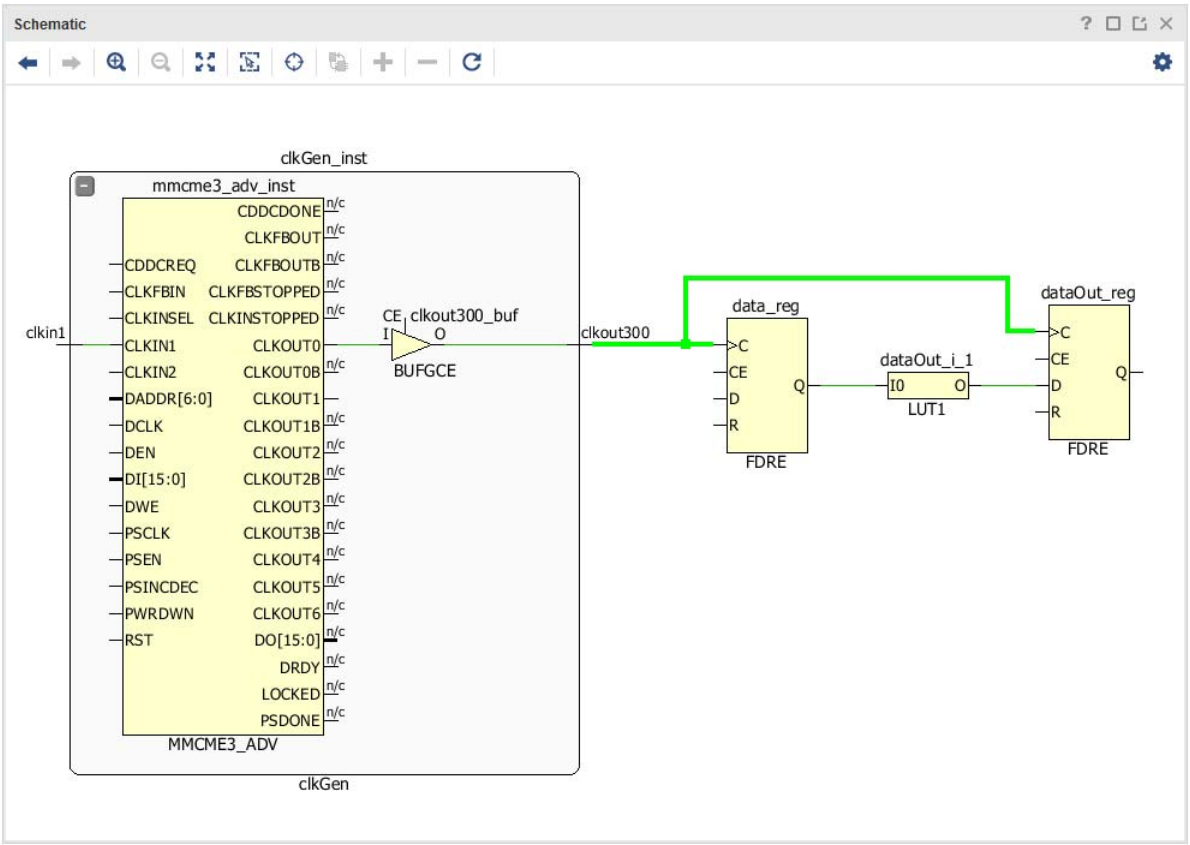



*Figure 5-15:* **Typical Synchronous Clocking Topology with Common Node Located on Green Net**

When analyzing the clock path in the timing report, the delays before and after the common node are not provided separately because the common node only exists in the physical database of the design and not in the logical view. For this reason, you can see the common node in the Device window of the Vivado IDE when the Routing Resources are turned on but not in the Schematic window. The timing report only provides a summary of skew calculation with source clock delay, destination clock delay, and credit from clock pessimism removal (CPR) up to the common node.

## *Limiting Synchronous Clock Domain Crossing Paths*

Timing paths between synchronous clocks driven by separate clock buffers exhibit higher skew, because the common node is located before the clock buffers. That is, the common node is farther from the leaf clock pins, resulting in higher pessimism in the timing analysis. The clock skew is even worse for timing paths between unbalanced clock trees due the delay difference between the source and destination clock paths. Although positive skew helps with meeting setup time, it hurts hold time closure, and vice versa.

In the following figure, three clocks have several intra and inter clock paths. The common node of the two clocks driven by the MMCM is located at the output of the MMCM (red markers). The common node of the paths between the MMCM input clock and MMCM output clocks is located on the net before the MMCM (blue marker). For the paths between

the MMCM input clock and MMCM output clocks, the clock skew can be especially high depending on the clkin_buf BUFGCE location and the MMCM compensation mode.

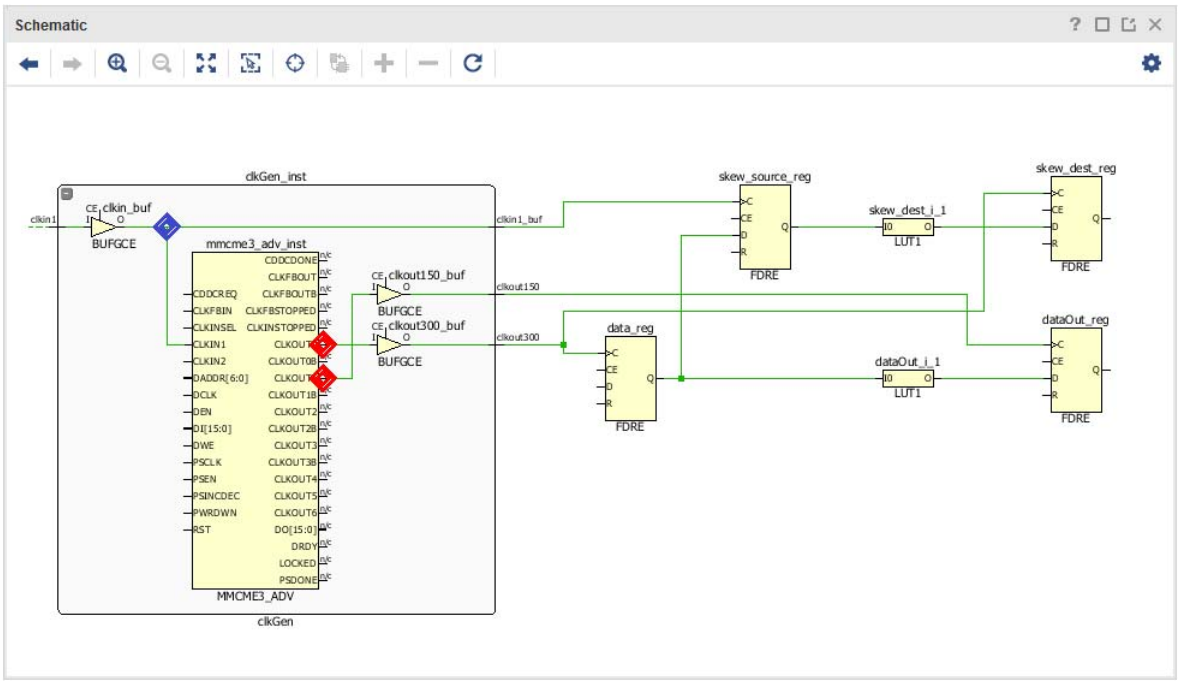



*Figure 5-16:* **Synchronous CDC Paths with Common Nodes on Input and Output of a MMCM**

Xilinx recommends limiting the number of synchronous clock domain crossing paths even when clock skew is acceptable. Also, when skew is abnormally high and cannot be reduced, Xilinx recommends treating these paths as asynchronous by implementing asynchronous clock domain crossing circuitry and adding timing exceptions.

### *Adding Timing Exceptions between Asynchronous Clocks*

Timing paths in which the source and destination clocks originate from different primary clocks or have no common node must be treated as asynchronous clocks. In this case, the skew can be extremely large, making it impossible to close timing.

You must review all timing paths between asynchronous clocks to ensure the following:

- Proper asynchronous clock domain crossing circuitry (`report_cdc`)
- Timing exception definitions that ignore timing analysis (`set_clock_groups`, `set_false_path`) or ignore skew (`set_max_delay -datapath_only`)

You can use the Clock Interaction Report (`report_clock_interaction`) to help identify clocks that are asynchronous and are missing proper timing exceptions. For more information on CDC path constraints, see Defining Clock Groups and CDC Constraints in Chapter 3.

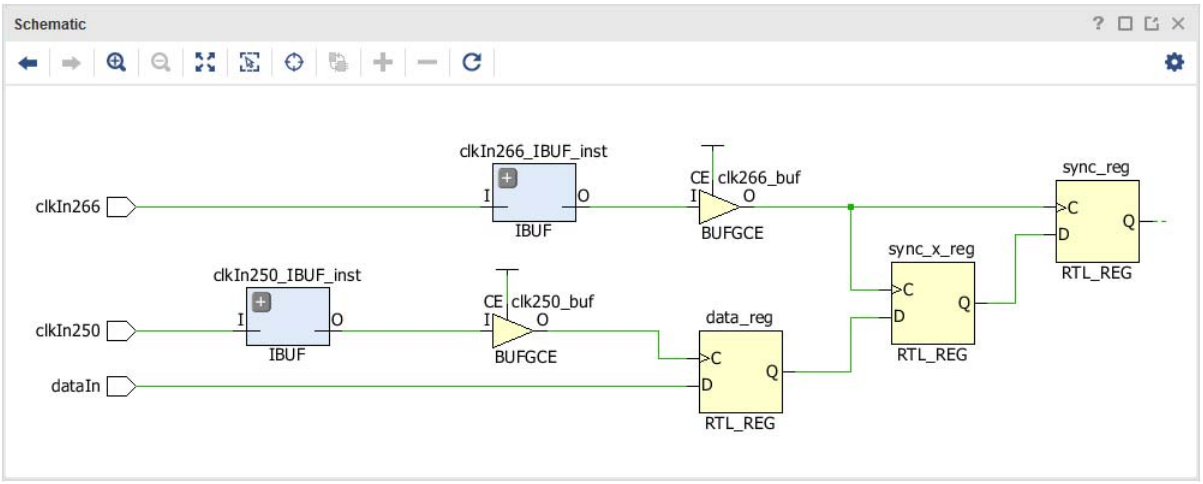



*Figure 5-17:* **Asynchronous CDC Paths with Proper CDC Circuitry and No Common Node**

## *Applying Common Techniques for Reducing Clock Skew*

Because the 7 series and UltraScale device clocking architectures differ, review the Clocking Guidelines in Chapter 3 to learn the best practice for each architecture, and verify that your design complies.

**TIP:** *Given the flexibility of the UltraScale device clocking architecture, the* `report_methodology` *command contains checks to aid you in creating an optimal clocking topology.*

The following techniques cover the most common scenarios:

- Avoid timing paths between cascaded clock buffers by eliminating unnecessary buffers or connecting them in parallel as shown in the following figure.

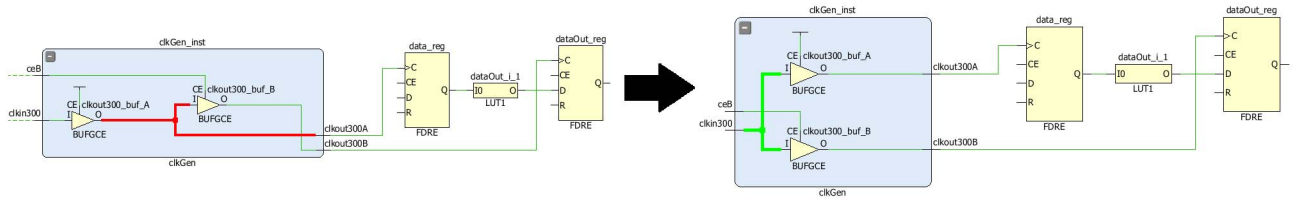



*Figure 5-18:* **Synchronous Clocking Topology with Cascaded BUFG Reconnected in Parallel**

- Combine parallel clock buffers into a single clock buffer and connect any clock buffer clock enable logic to the corresponding sequential cell enable pins, as shown on figure below. If some of the clocks are divided by the buffers built-in divider, implement the equivalent division with clock enable logic and apply multicycle path timing exceptions as needed. When both rising and falling clock edges are used by the downstream logic or when power is an important factor, this technique might not be applicable.

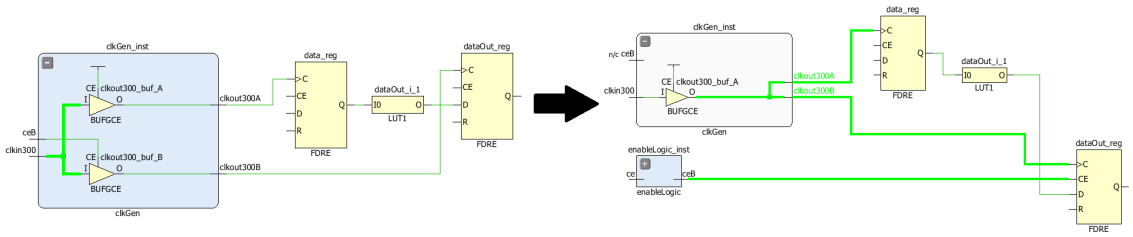



*Figure 5-19:* **Synchronous Clocking Topology with Parallel Clock Buffer Recombined into a Single Buffer**

- Remove LUTs or any combinatorial logic in clock paths as they make clock delays and clock skew unpredictable during placement, resulting in lower quality of results. Also, a portion of the clock path is routed with general interconnect resources which are more sensitive to noise than global clocking resources. Combinatorial logic usually comes from sub-optimal clock gating conversion and can usually be moved to clock enable logic, either connected to the clock buffer or to the sequential cells.

  In the following figure, the first BUFG (clk1_buf) is used in LUT3 to create a gated clock condition.

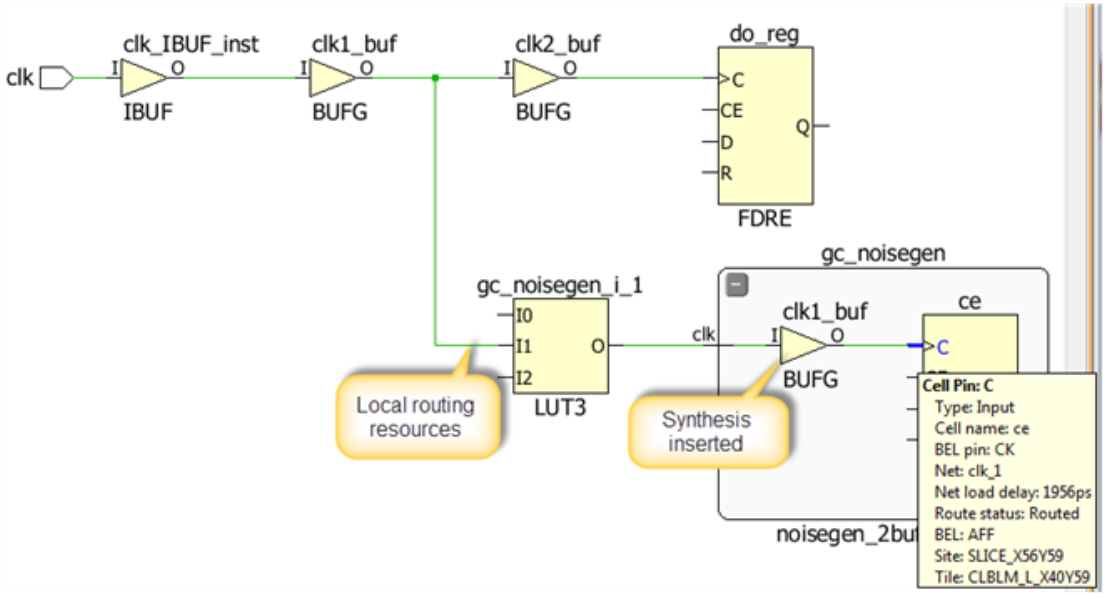



*Figure 5-20:* **Skew Due to Local Routing on Clock Network**

### *Applying Techniques for Improving Skew in 7 Series Devices*

Although the 7 series and UltraScale architectures differ in terms of clock architectures, some general clock considerations apply to both families:

- Do not use the CLOCK_DEDICATED_ROUTE=FALSE constraint in a production 7 series design. Use CLOCK_DEDICATED_ROUTE=FALSE only as a temporary workaround to a clock failure ONLY to produce an implemented design in order to view the clocking topology for debugging. Clock paths routed with fabric interconnect can have high clock skew and be impacted by switching noise, leading to poor performance or non-functional designs. In the following figure, the right side has a dedicated clock route, while on the left side, the dedicated route is disabled for clock.

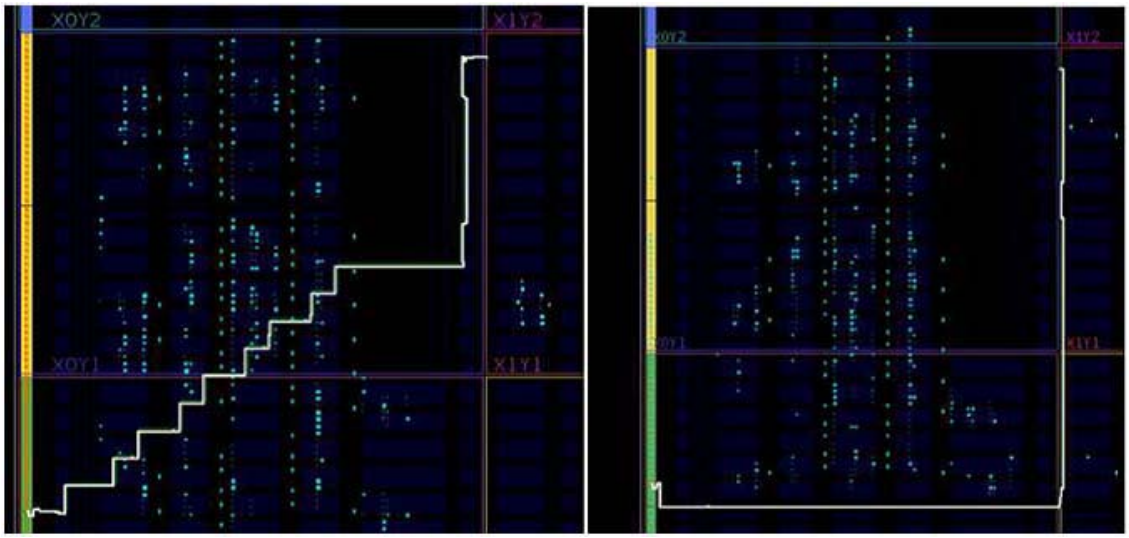



*Figure 5-21:* **Comparison of Fabric Clock Route versus Dedicated Clock Route**

- Do not allow regional clock buffers (BUFR/BUFIO/BUFH) to drive logic in several clock regions as the skew between the clock tree branches in each region will be very high. Remove inappropriate LOC or Pblock constraints to resolve this situation.

### *Improving Skew in UltraScale and UltraScale+ Devices*

- Avoid using an MMCM or PLL to perform simple division of a BUFG_GT clock. BUFG_GT cells have the ability to divide down the input clock. The following figure shows how to save an MMCM resource and implement balanced clock trees for two clocks originating from a GTHE3_CHANNEL cell.

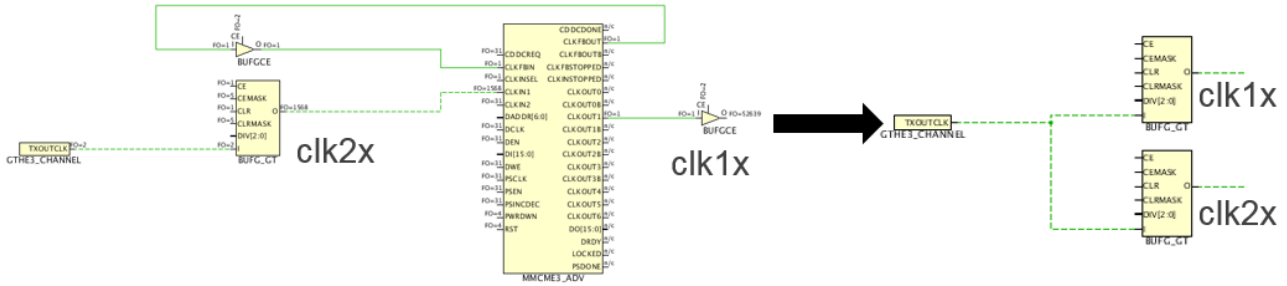



*Figure 5-22:* **Implementing Balanced Clock Trees using UltraScale BUFG_GTs**

- Use the CLOCK_DELAY_GROUP on the driver net of critical synchronous clocks to force CLOCK_ROOT and route matching during placement and routing. The buffers of the clocks must be driven by the same cell for this constraint to be honored. For details, see Synchronous CDC in Chapter 3.
- If a timing path is having difficulty meeting timing and the skew is larger than expected, it is possible that the timing path is crossing an SLR or an I/O column. If this is the case, physical constraints such as Pblocks may be used to force the source and destination into a single SLR or to prevent the crossing of an I/O column.
- When working with high speed synchronous clock domain crossing timing paths, LOCing the clock modifying blocks such as the MMCM/PLL to the center of the clock loads can aid in meeting timing. The decreased delay on the clock networks will result in less timing pessimism on the clock domain crossing paths.
- Verify that clock nets with CLOCK_DEDICATED_ROUTE=FALSE constraint are routed with global clocking resources. Use ANY_CMT_COLUMN instead of FALSE to ensure the clock nets with routing waivers are routed with dedicated clocking resources only. See Clock Constraints in Chapter 3 for more details on CLOCK_DEDICATED_ROUTE support. If the clock net is routed with fabric interconnect, identify the design change or clocking placement constraint needed to resolve this situation and make the implementation tools use global clocking resources instead. Clock paths routed with fabric interconnect can have high clock skew or be impacted by switching noise, leading to poor performance or non-functional designs.

## Reducing Clock Uncertainty

Clock uncertainty is the amount of uncertainty relative to an ideal clock. Uncertainty can come from user-specified external clock uncertainty (`set_clock_uncertainty`), system jitter or duty cycle distortion. Clock Modifying Blocks such as the MMCM and PLL also contribute to clock uncertainty in the form of Discrete Jitter, and Phase Error if multiple related clocks are used.

The Clocking Wizard provides accurate uncertainty data for the specified device and can generate various MMCM clocking configurations for comparing different clock topologies. To achieve optimal results for the target architecture, Xilinx recommends regenerating clock

generation logic using the Clocking Wizard rather than using legacy clock generation logic from prior architectures.

### *Using MMCM Settings to Reduce Clock Uncertainty*

When configuring an MMCM for frequency synthesis, the target frequency may have several possible M (multiplier) and D (divider) values to achieve the same goal. The M and D values that result in the highest VCO frequency that does not exceed the maximum VCO frequency for the device will minimize the clock uncertainty.

The MMCM frequency synthesis example below uses an input clock of 62.5 MHz to generate an output clock of approximately 40 MHz. There are two solutions, but the MMCM_2 with a higher VCO frequency generates less clock uncertainty due to reduced jitter and phase error.

*Table 5-7:* **MMCM Frequency Synthesis Example**

| | MMCM_1 | MMCM_2 |
|---|---|---|
| Input clock | 62.5 MHz | 62.5 MHz |
| Output clock | 40.0 MHz | 39.991 MHz |
| CLKFBOUT_MULT_F | 16 | 22.875 |
| CLKOUT0_DIVIDE_F | 25 | 35.750 |
| VCO Frequency | 1000.000 MHz | 1429.688 |
| Jitter (ps) | 167.542 | 128.632 |
| Phase Error (ps) | 384.432 | 123.641 |

**TIP:** *When using the Clocking Wizard from the IP Catalog, make sure that Jitter Optimization Setting is set to the Minimum Output Jitter, which provides the higher VCO frequency.*

### *Using BUFGCE_DIV to Reduce Clock Uncertainty*

In UltraScale devices, BUFGCE_DIV cells can be used to reduce clock uncertainty on synchronous clock domain crossings by eliminating MMCM Phase Error. For example, consider a path between a 300 MHz and 150 MHz clock domains, where both clocks are generated by the same MMCM.

In this case, the clock uncertainty includes 120 ps of Phase Error for both Setup and Hold analysis. Instead of generating the 150 MHz clock with the MMCM, a BUFGCE_DIV can be connected to the 300 MHz MMCM output and divide the clock by 2. For optimal results, the 300 MHz clock needs to also use a BUFGCE_DIV, with not active division, in order to match the 150 MHz clock delay accurately, as shown in the following figure.

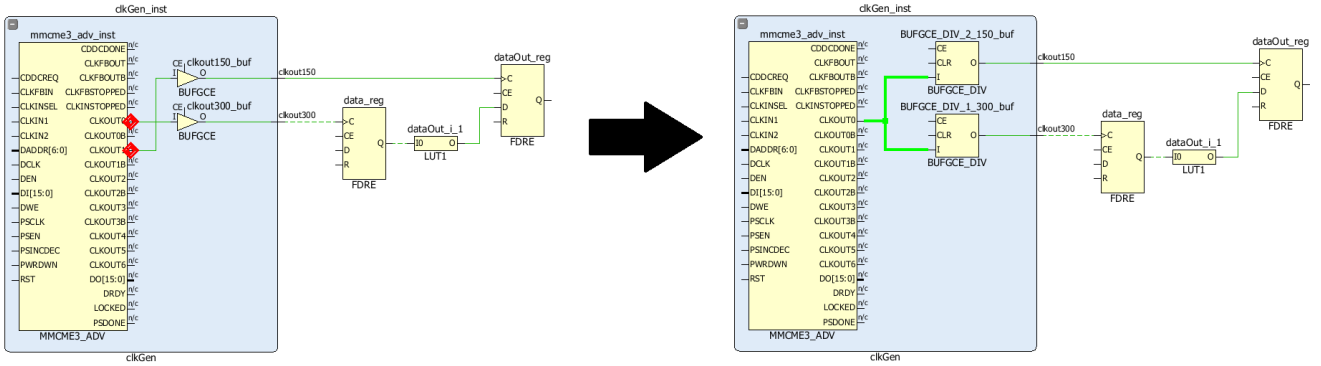



*Figure 5-23:* **Improving the Clock Topology for an UltraScale Synchronous CDC Timing Path**

With the new topology:

- For setup analysis, clock uncertainty does not include the MMCM phase error and is reduced by 120 ps.
- For hold analysis, there is no more clock uncertainty (only for same edge hold analysis).
- The common node moves closer to the buffers, which saves some clock pessimism.

By applying the CLOCK_DELAY_GROUP constraint on the two clock nets, the clock paths will have matched routing. For more information, see Synchronous CDC in Chapter 3.

The following table compares the clock uncertainty for setup and hold analysis of an UltraScale synchronous CDC timing path.

*Table 5-8:* **Comparison of Clock Uncertainty for Setup and Hold Analysis of an UltraScale Synchronous CDC Timing Path**

| **Setup Analysis** | **MMCM Generated 150 MHz Clock** | | **BUFGCE_DIV 150 MHz Clock** |
|---|---|---|---|
| | Total System Jitter (TSJ) | 0.071 ns | 0.071 ns |
| | Discrete Jitter (DJ) | 0.115 ns | 0.115 ns |
| | Phase Error (PE) | 0.120 ns | 0.000 ns |
| | Clock Uncertainty | 0.188 ns | 0.068 ns |
| **Hold Analysis** | **MMCM Generated 150 MHz Clock** | | **BUFGCE_DIV 150 MHz Clock** |
| | Total System Jitter (TSJ) | 0.071 ns | 0.000 ns |
| | Discrete Jitter (DJ) | 0.115 ns | 0.000 ns |
| | Phase Error (PE) | 0.120 ns | 0.000 ns |
| | Clock Uncertainty | 0.188 ns | 0.000 ns |

# Applying Common Timing Closure Techniques

The following techniques can help with design closure on challenging designs. Before attempting these techniques, ensure that the design is properly constrained and that you identify the main issue that affects the top violating paths.

**RECOMMENDED:** *Xilinx recommends running the `report_qor_suggestions` Tcl command to identify and apply many of these techniques automatically. For more information, see this link in the Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906) [Ref 21].*

## Improving the Netlist with Block-Level Synthesis Strategies

While most designs can meet timing requirements with the default Vivado synthesis settings, larger and more complex designs usually require a mix of synthesis strategies for different hierarchies to close timing.

### *Reducing MUXF Mapping to Lower Congestion*

**TIP:** *This optimization technique is automatically applied by the `report_qor_suggestions` Tcl command.*

Using MUXF cells helps critical paths with many logic levels or a tight clock requirement while also reducing power. However, this approach limits placement flexibility when the netlist connectivity is complex, leading to potential higher routing congestion and timing degradation. To reduce MUXF utilization in a congested module, you can use the block-level synthesis strategy with the BLOCK_SYNTH.MUXF_MAPPING constraint. For more information on using this strategy, see Block-Level Synthesis Strategy in Chapter 4.

As described in Identifying Congestion, you can determine whether timing closure is impacted by routing congestion by reviewing the Router Initial Estimated Congestion table in the log files or in the Design Analysis report (`report_design_analysis -congestion`) after place or route is complete.

In the following figure, the Design Analysis report shows that 7% of the device is impacted by Short congestion level 5 (32x32 CLBs) in the South direction while 26% MUXF are utilized in the corresponding congested area.

| Direction | Type | Congestion Level | Percentage Tiles | Congestion Window | Cell Names | Combined LUTs | LUT6 |...| MUXF |
|---|---|---|---|---|---|---|---|---|---|
| ... | | | | | | | | | |
| North | Short | 4 | 6.983% | (CLEL_L_X48Y73,CLEL_R_X63Y88) | inst_name1(92%) | 0% | 22% |...| 4% |
| South | Short | 5 | 7.136% | (CLEL_L_X32Y297,CLEL_R_X63Y328) | inst_name2(99%) | 2% | 49% |...| 26% |
| East | Short | 4 | 6.005% | (CLEL_L_X48Y109,CLE_M_X63Y125) | inst_name3(94%) | 0% | 38% |...| 10% |
| West | Short | 4 | 6.541% | (CLEL_L_X32Y273,CLE_M_X47Y320) | inst_name4(92%) | 1% | 48% |...| 23% |

*Figure 5-24:* **South Short Congestion in the report_design_analysis Congestion Table**

In the Vivado IDE, you can select a row in the table of the Design Analysis congestion report to highlight the corresponding congested area in the Device window. The following figure shows that the congestion overlaps with a higher MUXF density area. The MUXF cells are highlighted in magenta using the following command in the Vivado IDE Tcl Console:

```
highlight_objects -color magenta [get_cells -hier -filter REF_NAME=~MUXF*]
```

MUXF* includes MUXF7/MUXF8/MUXF9, which are dedicated multiplexer resources located within the CLB. These resources are grouped with up to 8 LUTs during placement, forcing high CLB input utilization with higher routing demand and limiting placement flexibility. The estimated congestion per CLB is displayed using the Vivado IDE metrics.

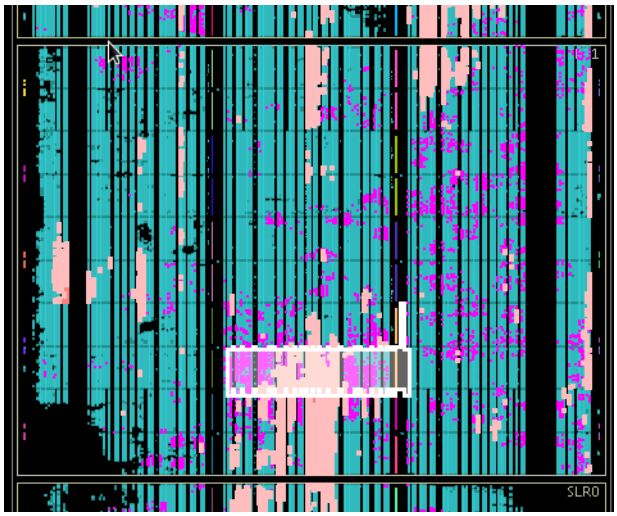



*Figure 5-25:* **MUXF Congestion Highlighted in the Vivado IDE Device Window**

In this situation, Xilinx recommends reducing the number of MUXF* by mapping their corresponding functionality to LUTs, which have higher placement and routing flexibility. You can use the following command in the XDC synthesis constraints to modify the netlist:

```
set_property BLOCK_SYNTH.MUXF_MAPPING 0 [get_cells inst_name4]
```

After rerunning synthesis, place, and route, the updated congestion table in the Design Analysis report now shows that the South Short congestion is lower (level 4), which typically improves the timing quality of results.

| Direction | Type | Congestion Level | Percentage Tiles | Congestion Window | Cell Names | Combined LUTs | LUT6 | ... | MUXF |
|---|---|---|---|---|---|---|---|---|---|
| ... | | | | | | | | | |
| North | Short | 4 | 6.983% | (CLEL_L_X48Y73,CLEL_R_X63Y88) | inst_name1(92%) | 0% | 22% | ... | 4% |
| South | Short | 4 | 9.040% | (CLEL_L_X34Y297,CLEL_R_X49Y312) | inst_name2(99%) | 2% | 54% | ... | 4% |
| East | Short | 4 | 6.005% | (CLEL_L_X48Y109,CLE_M_X63Y125) | inst_name3(94%) | 0% | 38% | ... | 10% |
| West | Short | 4 | 6.541% | (CLEL_L_X32Y273,CLE_M_X47Y320) | inst_name4(92%) | 1% | 48% | ... | 23% |

*Figure 5-26:* **Initial Router Congestion Table after Reducing MUXF Usage on a Module**

### Improving Logic Levels

Throughout the design cycle, you must verify that the logic level distribution fits the clock frequency goals for the target Xilinx FPGA family and device speed grade. Although a limited number of paths with a high number of logic levels do not always introduce a timing closure challenge, you can improve the timing QoR by optimizing the longest paths in the design with the Vivado synthesis retiming option.

Using the retiming option globally is usually runtime intensive and can negatively impact power. Therefore, Xilinx recommends that you identify a specific hierarchy with violations on paths with a high number of logic levels after synthesis or with optimal placement. When the paths in the fanin or fanout of the longest paths have fewer logic levels and are contained within a small or medium hierarchical module, you can use the BLOCK_SYNTH.RETIMING block-level synthesis strategy.

The following figure shows a critical paths with 5 LUTs, constrained by a 600 MHz clock. The REG2 destination flop drives a timing path with a single LUT that is included one hierarchy up from REG2.

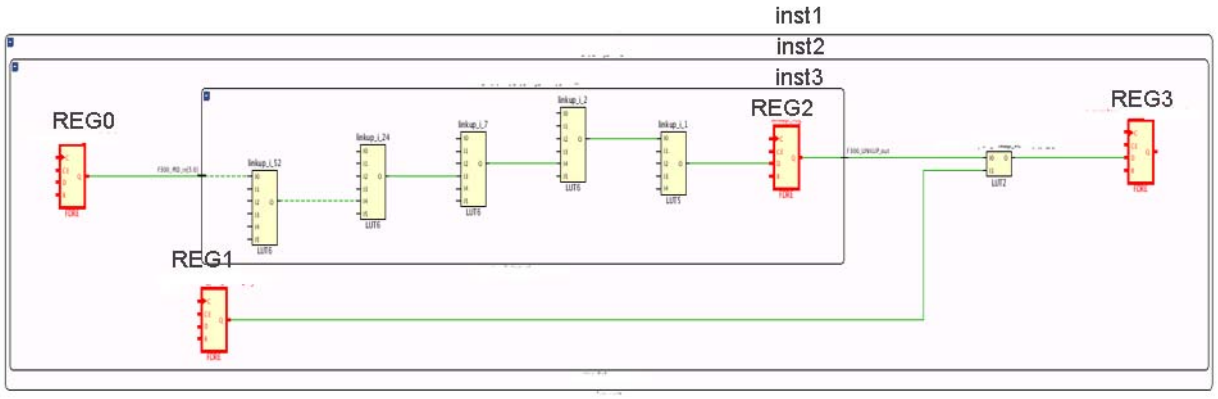



*Figure 5-27:* **Schematic Showing Critical Path with 5 Logic Levels**

In addition to using the Schematic window in the Vivado IDE, you can use the `report_design_analysis -logic_level_distribution` command to review the distribution of logic levels for specific paths. This allows you to determine how many paths need to be rebalanced to improve the timing QoR.

You can use the `retiming_forward` and `retiming_backward` attributes available in Vivado synthesis to control the optimization on a specific register or a path. Using these attributes applies retiming optimization on a specific set of paths rather than on the top module or submodules, which reduces the area overhead. You can apply these attributes in the RTL or in the XDC file. For more information, including usage and restrictions, see the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 16].

The following figure shows 58 paths with 5 logic levels within the inst1/inst2 hierarchy constrained with the 600 MHz clock and 32 paths with only 1 logic level.

```
current_instance inst1/inst2

report_design_analysis -timing -logic_level_distribution -of_timing_paths [get_timing_paths -max_paths 100 -group core_clk_600]

+------------------+-------------+---+----+----+---+---+----+---+---+---+---+----+-------+-------+-------+-------+-----+
|  End Point Clock | Requirement | 0 | 1  | 2  | 3 | 4 | 5  | 6 | 7 | 8 | 9 | 10 | 11-15 | 16-20 | 21-25 | 26-30 | 31+ |
+------------------+-------------+---+----+----+---+---+----+---+---+---+---+----+-------+-------+-------+-------+-----+
| core_clk_600     | 1.667ns     | 0 | 32 | 10 | 0 | 0 | 58 | 0 | 0 | 0 | 0 |  0 |     0 |     0 |     0 |     0 |   0 |
+------------------+-------------+---+----+----+---+---+----+---+---+---+---+----+-------+-------+-------+-------+-----+
```

*Figure 5-28:* **Logic Level Distribution with Default Synthesis Optimization**

Vivado synthesis can rebalance the logic levels by moving the registers in the low logic level paths into the high logic level paths. In this example, you can add the following constraint to the synthesis XDC file to perform retiming on the inst1/inst2 hierarchy:

```
set_property BLOCK_SYNTH.RETIMING 1 [get_cells inst1/inst2]
```

After rerunning synthesis with the same global settings and the updated XDC file, you can run regular timing analysis on the inst1/inst2 timing paths or rerun the `report_design_analysis` command to verify that the longest paths have fewer logic levels, as shown in the following figure. The critical path is now REG0 > 3 LUTs > REG2 (backward retimed), and the path from REG2 to REG4 has 3 logic levels.

```
current_instance inst1/inst2

report_design_analysis -timing -logic_level_distribution -of_timing_paths [get_timing_paths -max_paths 100 -group core_clk_600]

+------------------+-------------+---+---+---+-----+---+---+---+---+---+---+----+-------+-------+-------+-------+-----+
|  End Point Clock | Requirement | 0 | 1 | 2 |  3  | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11-15 | 16-20 | 21-25 | 26-30 | 31+ |
+------------------+-------------+---+---+---+-----+---+---+---+---+---+---+----+-------+-------+-------+-------+-----+
| core_clk_600     | 1.667ns     | 0 | 0 | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 |  0 |     0 |     0 |     0 |     0 |   0 |
+------------------+-------------+---+---+---+-----+---+---+---+---+---+---+----+-------+-------+-------+-------+-----+
```

*Figure 5-29:* **Logic Level Distribution with Retiming Enabled for Synthesis Optimization**

## Reducing Control Sets

**TIP:** *This optimization technique is automatically applied by the `report_qor_suggestions` Tcl command.*

Often not much consideration is given to control signals such as resets or clock enables. Many designers start HDL coding with "if reset" statements without deciding whether the reset is needed or not. While all registers support resets and clock enables, their use can significantly affect the end implementation in terms of performance, utilization, and power.

The first factor to consider is the number of control sets. A control set is the group of clock, enable, and set/reset signals used by a sequential cell. For example, two cells connected to the same clock have different control sets if only one cell has a reset or if only one cell has

a clock enable. Constant or unused enable and set/reset register pins also contribute to forming control sets.

The second factor to consider is the targeted architecture. The number of control sets that can be packed together depends on the architecture:

- A 7 series device slice (or half-CLB) comprises 8 registers, which all share 1 clock, 1 set/reset, and 1 clock enable. Only 1 control set can be used per group of 8 registers.
- An UltraScale device half-CLB comprises 2 groups of 4 registers, which share 1 clock and 1 set/reset. In addition, each group of 4 registers has 1 clock enable and can ignore the set/reset. A constant set/reset signal is not routed and can be ignored. A constant enable signal is treated like a dynamic enable signal and needs to be routed. Under optimal conditions, up to 2 control sets can be used per group of 8 registers.

CLB packing restrictions caused by control sets force the placer to move some registers, including their input LUT. In some cases, the registers are moved to less optimal locations. The additional distance can negatively impact not only utilization but also placement QoR and power consumption, due to logic spreading (longer net delays) and higher interconnect resources utilization. This is mainly of concern in designs with many low fanout control signals, such as clock enables that feed single registers.

Despite the higher UltraScale device CLB control set capacity, typical designs show a control set utilization similar to 7 series designs. Therefore, Xilinx recommendations are the same for both architectures. The following table provides a guideline for the recommended number of control sets, depending on the target device size, for both Xilinx 7 series and UltraScale devices.

*Table 5-9:* **Control Set Guideline for 7 Series and UltraScale Devices**

| | Percentage of Control Sets |
|---|---|
| **Acceptable** | Less than 7.5% of the total number of control sets in the device |
| **Reduction Recommended** | Between 7.5% and 15% of the total number of control sets in the device |
| **Reduction Required** | Greater than 15% of the total number of control sets in the device |

These guidelines assume the following:

- Typical control set capacity: 1 per 8 CLB registers
- Total number of control sets in a device: CLB registers / 8

To determine the number of control sets in a design:

- Before placement: Use `report_control_sets -verbose`
- After placement: Use `report_utilization` (text mode only)

**TIP:** *The number of unique control sets can be a problem in a small portion of the design, resulting in longer net delays or congestion in the corresponding device area. Identifying the high local density of unique control sets requires detailed placement analysis in the Vivado IDE Device window, which includes highlighted control signals in different colors.*

If the number of control sets is high, use one of the following strategies to reduce their number:

- Remove the MAX_FANOUT attributes that are set on control signals in the HDL sources or constraint files. Replication on control signals dramatically increases the number of unique control sets. Xilinx recommends using `place_design -fanout_opt` to replicate in the placer and `phys_opt_design -directive Explore` for finer replication after placer. This prevents unnecessary replication and equivalent control sets from crossing each other, which can lead to routing congestion.
- Increase the control set threshold of Vivado synthesis (or other FPGA synthesis tool). Review the control sets fanout distribution table in `report_control_sets -verbose` to determine a more appropriate control sets threshold to use during synthesis. For example:

```
synth_design -control_set_opt_threshold 16
```

**TIP:** *Use the BLOCK_SYNTH synthesis constraints to change the control sets threshold on modules that are the most impacted by placement spreading or congestion.*

- Use `opt_design -control_set_merge` or `opt_design -merge_equivalent_drivers` to merge equivalent control sets after synthesis.
- Avoid low fanout asynchronous set/reset (preset/clear), because they can only be connected to dedicated asynchronous pins and cannot be moved to the datapath by synthesis. For this reason, the synthesis control set threshold option does not apply to asynchronous set/reset.
- Avoid using both active-High and active-Low of a control signal for different sequential cells.
- Only use clock enable and set/reset when necessary. Often data paths contain many registers that automatically flush uninitialized values, and where set/reset or enable signals are only needed on the first and last stages.

Additional synthesis attributes and recommendations on control signals are available in Control Signals and Control Sets in Chapter 3.

## Optimizing High Fanout Nets

High fanout nets often lead to implementation issues. As die sizes increase with each FPGA family, fanout problems also increase. It is often difficult to meet timing on nets that have

many thousands of endpoints, especially if there is additional logic on the paths, or if they are driven from non-sequential cells, such as LUTs or distributed RAMs.

### *Use Register Replication*

Most synthesis tools provide a fanout threshold limit to force high fanout driver replication. The Vivado synthesis option is `synth_design -fanout_limit 5000`. Adjusting this global threshold does not allow you to control which registers can be replicated. A better method is to apply attributes on specific registers or levels of hierarchy to specify which registers can or cannot be replicated. For example, if a LUT1 rather than a register is being used for replication, it indicates that an attribute or constraint is preventing the optimization, such as a DONT_TOUCH attribute on a hierarchical cell or net segment in a different hierarchy.

Sometimes, designers address the high fanout nets in RTL or synthesis by using a global fanout limit or a MAX_FANOUT attribute on a specific net. This does not always result in the most optimal routing resource usage, especially if the MAX_FANOUT attribute is set too low. In addition, if the high fanout signal is a register control signal and is replicated more than necessary, this can lead to a higher number of control sets.

Often, a better approach to reducing fanout is to use a balanced tree for the high fanout signals. Consider manually replicating registers based on the design hierarchy, because the cells included in a hierarchy are often placed together. Review the recommendations in Use Register Replication in Chapter 3.

To structure control set trees as described in the preceding example, you can use the `opt_design` Tcl command with one of the following options:

- `-control_set_merge`: This option reduces the drivers of logically-equivalent control signals to a single driver.
- `-merge_equivalent_drivers`: This option reduces the drivers of logically-equivalent signals, including control signals, to a single driver.

These options are the reverse of fanout replication and result in nets that are better suited for module-based replication. This merge also works across multi-stage reset trees as shown in the following figure.

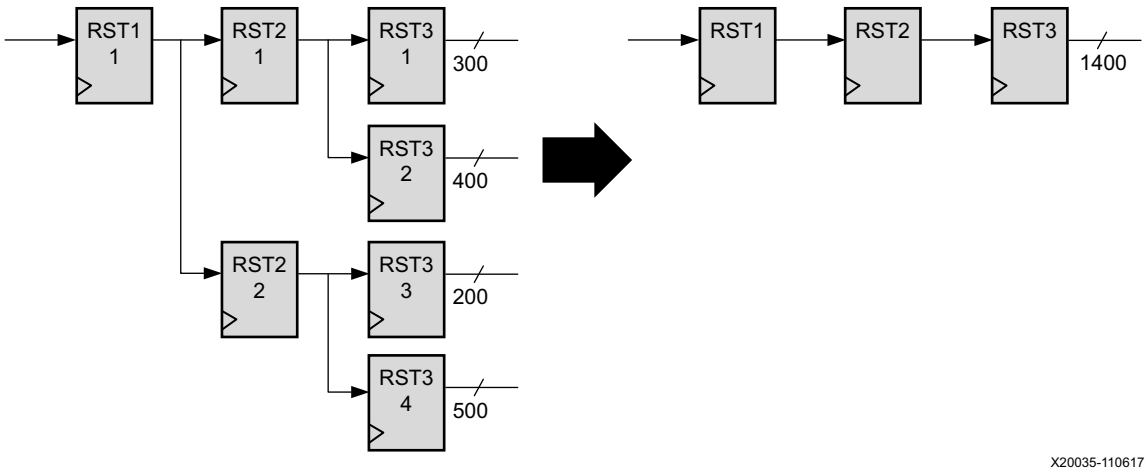



*Figure 5-30:* **Control Set Merging Using opt_design -control_set_merge**

- `-hier_fanout_limit <arg>`: This option replicates registers according to hierarchy where `<arg>` represents the fanout limit for the replication according to the logical hierarchy. For each hierarchical instance driven by the high fanout net, if the fanout within the hierarchy is greater than the specified limit, the net within the hierarchy is driven by a replica of the driver of the high fanout net. The replicated driver is placed in the same level of hierarchy as the original driver, and replication is not limited to control set registers. The following figure shows replication on a clock enable net with a fanout of 60,000 using `opt_design -hier_fanout_limit 1000`. Because each module SR_1K contains 1000 loads, the driver is replicated 59 times.

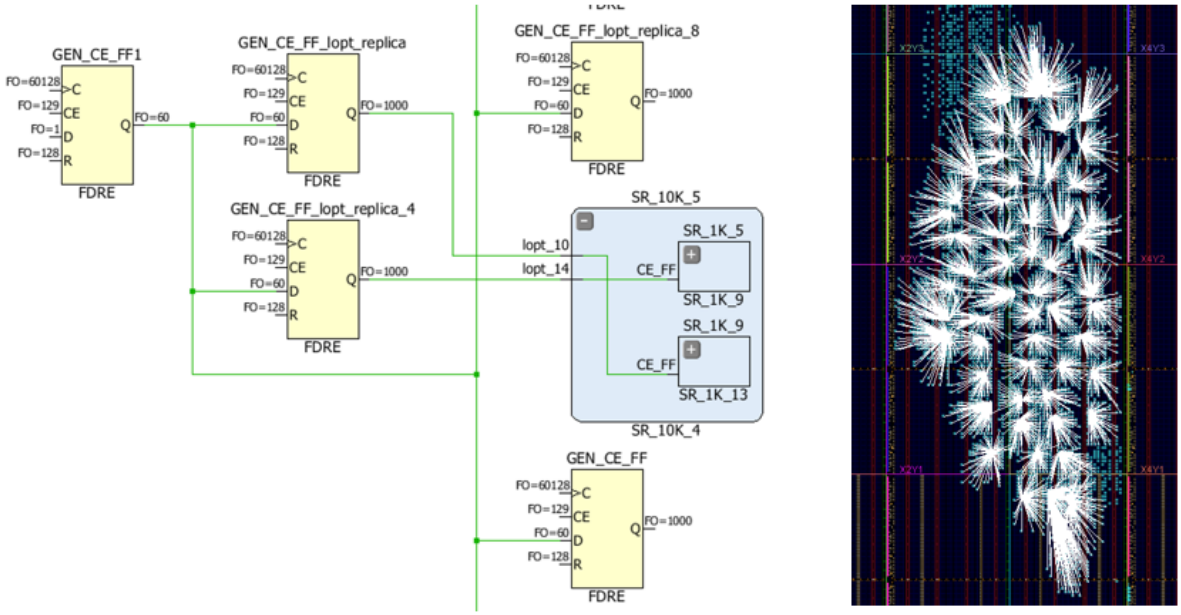



*Figure 5-31:* **Module-Based Replication on a High-Fanout Clock Enable Net**

You can also enable fanout optimization in `place_design` with the `-fanout_opt` option. Replication occurs early in the placer flow and is based on placement information. Registers that drive more than 1000 loads and registers that drive DSPs, block RAMs, FIFOs, and URAMs are considered for replication and are co-located with the loads if replication occurs.

For SSI technology devices, high-fanout drivers can be replicated for each SLR and optionally assigned to SLR-aligned Pblocks along with their loads. This technique helps reduce the impact of the SLR crossing delay and gives more freedom to place the replicated high fanout nets independently in each SLR.

## *Promote High Fanout Nets to Global Routing*

**TIP:** *This optimization technique is automatically applied by the `report_qor_suggestions` Tcl command.*

Lower performance high fanout nets can be moved onto the global routing by inserting a clock buffer between the driver and the loads. This optimization is automatically performed in `opt_design` for nets with a fanout greater than 25000 only when a limited number of clock buffers are already used. You can force `synth_design` and `opt_design` to insert a clock buffer when setting the CLOCK_BUFFER_TYPE attribute on a net in the RTL file or in the constraint file (XDC). For example:

```
set_property CLOCK_BUFFER_TYPE BUFG [get_nets netName]
```

Using global clocking insures optimal routing at the cost of higher net delay. For best performance, clock buffers must drive sequential loads directly, without intermediate combinatorial logic. In most cases, `opt_design` reconnects non-sequential loads in parallel to the clock buffer. If needed, you can prevent this optimization by applying a DONT_TOUCH on the clock buffer output net. Also, if the high fanout net is a control signal, you must identify why some loads are not dedicated clock enable or set/reset pins. Review the use of dedicated synthesis attribute to control local clock enable and set/reset optimizations in Control Signals and Control Sets in Chapter 3.

The placer also automatically routes high fanout nets (fanout > 1000) on any global routing tracks available after clock routing is performed. This optimization occurs towards the end of the placer flow and is only performed if timing does not degrade. You can disable this feature using the `-no_bufg_opt` option.

### *Use Physical Optimization*

Physical optimization (`phys_opt_design`) automatically replicates the high fanout net drivers based on slack and placement information, and usually significantly improves timing. Xilinx recommends that you drive high fanout nets with a fabric register (FD*), which is easier to replicate and relocate during physical optimization.

In some cases, the default `phys_opt_design` command does not replicate all critical high fanout nets. Use a different directive to increase the command effort: Explore, AggressiveExplore or AggressiveFanoutOpt. Also, when a high fanout net becomes critical during routing, you can add an iteration of `phys_opt_design` to force replication on specific nets before trying to route the design again. For, example:

```
phys_opt_design -force_replication_on_nets [get_nets [list netA netB netC]]
```

## Prioritize Critical Logic Using the group_path Option

You can use the `group_path` command with the `-weight` option to give higher priority to the path endpoints defined in the group. For example, to assign a higher priority to group of logic clocked by a specific clock, use the following command:

```
group_path -name [get_clocks clock] -weight 2
```

In this example, the implementation tools give higher priority to the paths with a weight of 2 over other paths in the design.

## Fixing Large Hold Violations Prior to Routing

For paths that have large hold violations (> 0.4 ns), it is advantageous to reduce the hold violations prior to routing the design, making it easier for the router to fix the remaining smaller hold violations using route detours. By inserting negative-edge triggered registers between sequential elements, the timing path can be split into two half period paths and the hold violation can be reduced.

You can insert the negative-edge triggered registers using the `-insert_negative_edge_ffs` option during the `phys_opt_design` implementation step. Only paths with flip-flop drivers and at most one LUT in between the sequential elements are considered for this optimization. The setup slack on the paths must be sufficiently positive after the optimization or else the optimization is discarded.

The following figure shows a negative-edge triggered register inserted after a flip-flop driving a CMAC block. Before the optimization, the hold slack between the flip-flop and the driver was -0.492 ns. After the insertion of the negative-edge triggered register (highlighted in blue), the setup and hold slack are both positive.

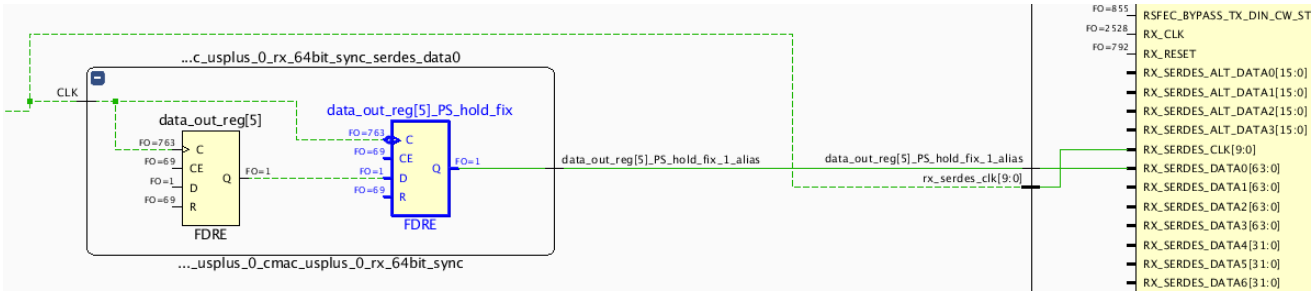



*Figure 5-32:* **Fixing Hold Violation with Negative Edge Register Insertion**

## Addressing Congestion

Congestion can be caused by a variety of factors and is a complex problem that does not always have a straightforward solution. Complexity and congestion have the same resolution techniques. Check to see if complex modules are placed in the congested regions of the device. The `report_design_analysis` congestion report helps you identify the congested regions and the top modules that are contained within the congestion window. Various techniques exist to optimize the modules in the congested region.

**TIP:** *Before you try to address congestion with the techniques being discussed below, make sure that you have clean constraints. Overlapping Pblocks can cause congestion and should be avoided. Excessive hold time failures or negative hold slack require the router to detour which can lead to congestion.*

### *Lower Device Utilization*

When several fabric resource utilization percentages are high (on average > 75%), placement becomes more challenging if the netlist complexity is also high (high top-level connectivity, high Rent exponent, high average fanout). High performance designs also come with additional placement challenges. In such situations, revisit the design features and consider removing non-essential modules until only one or two fabric resource utilization percentages are high. If logic reduction is not possible, review the other congestion alleviation techniques presented in this chapter.

**TIP:** *Review resource utilization after* `opt_design` *in order to get more accurate numbers, once unused logic has been trimmed instead of after synthesis.*

## *Balance SLR Utilization for SSI Devices*

When targeting SSI technology devices it is important to analyze the utilization per SLR region. Overall utilization might be low, but high utilization in one SLR might lead to a congestion.

In the following figure, the overall utilization for the design is low. However, the utilization in SLR2 is high and the logic requires more routing resources than logic in the other SLRs. The logic in this area is a wide bus MUX that saturates the routing resources.

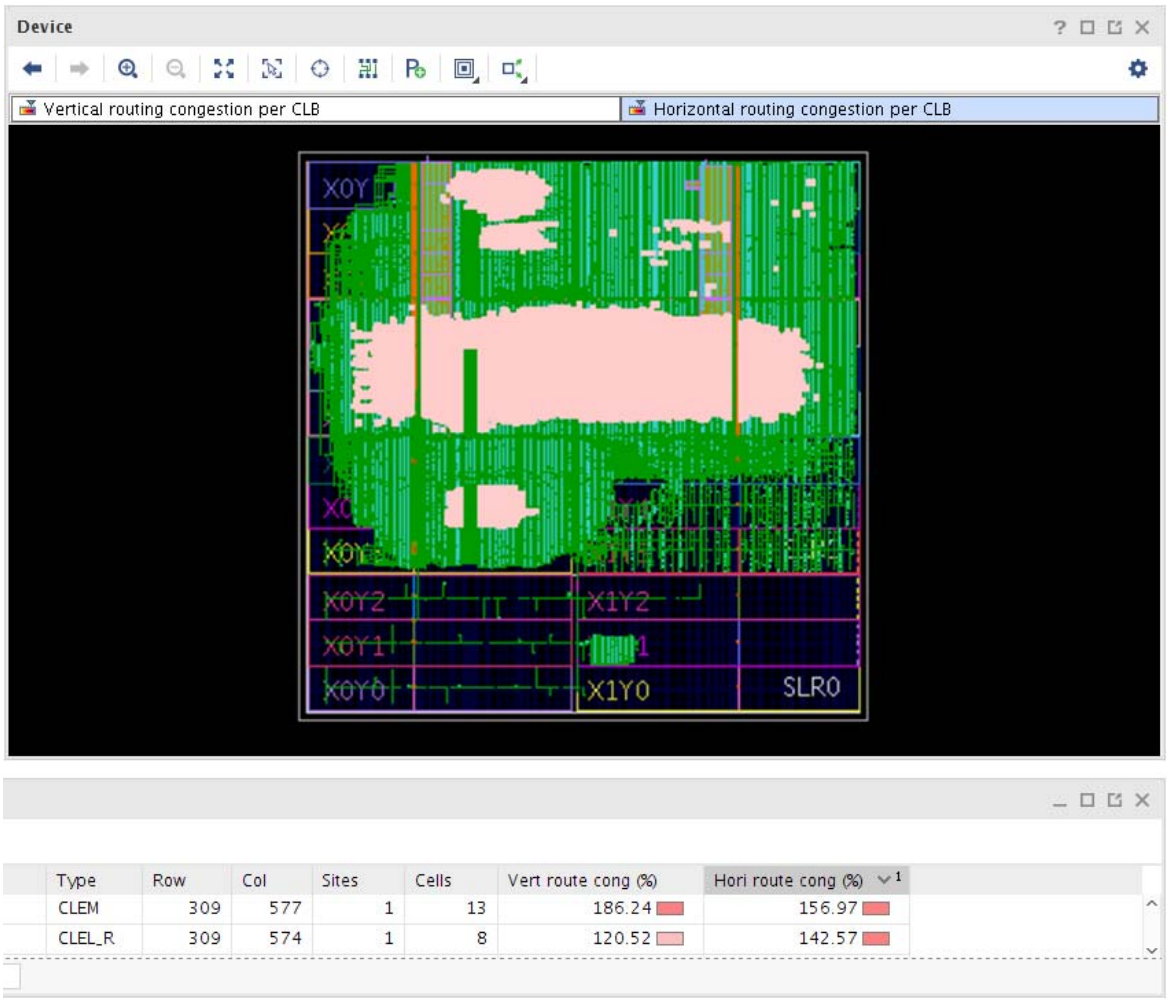



*Figure 5-33:* **Utilization Analysis per SLR Region**

To balance utilization, try the following:

- Use different placer directives for spreading the design.
- Use floorplanning constraints, such as Pblocks to keep some modules out of the highly utilized and congested SLR.

### *Use Alternate Placer and Router Directives*

Because placement typically has the greatest impact on overall design performance, applying different placer directives is one of the first techniques that should be tried to reduce congestion. Consider running the alternate placer directives without any existing Pblock constraints in order to give more freedom to the placer to spread the logic as needed.

Several placer directives exist that can help alleviate congestion by spreading logic throughout the device to avoid congested regions. The SpreadLogic placer directives are:

- AltSpreadLogic_high
- AltSpreadLogic_medium
- AltSpreadLogic_low
- SSI_SpreadLogic_high
- SSI_SpreadLogic_low

When congestion is detected on SLR crossing, consider using:

- SSI_BalanceSLLs placer directive which helps with partitioning the design across SLRs while attempting to balance SLLs between SLRs.
- SSI_SpreadSLLs placer directive which allocates extra area for regions of higher connectivity when partitioning across SLRs.

Other placer directives or implementation strategies might also help with alleviating congestions and should also be tried after the placer directives mentioned above.

To compare congestion for different placer directives either run the Design Analysis Congestion report after `place_design`, or examine the initial estimated congestion in the router log file. Review the results against the congestion level ranges shown in Table 5-4.

Routing has less impact on congestion than placer directives. However, in some cases it is useful to attempt different routing directives. The following directive ensures that the router works harder to access more routing and relieve congestion in the interconnect tiles:

- AlternateCLBRouting

***Note:*** The AlternateCLBRouting routing directive is most effective when there is short congestion or both short and long congestion. This directive only applies to UltraScale devices.

For more information, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

### Turn Off Cross-Boundary Optimization

Prohibiting cross-boundary optimization in synthesis prevents additional logic getting pulled into a module. This reduces the complexity of the modules but can also lead to higher overall utilization. This can be done globally with the `-flatten_hierarchy none` option in `synth_design`. This same technique can be applied on specific modules with the KEEP_HIERARCHY attribute in RTL.

### Disable LUT Combining and MUXF Inference

If utilization of MUXF* primitives or LUT combining is high in the congested area (> 40%), you can try using a synthesis strategy that eliminates MUXF* usage and LUT combining to help alleviate congestion. MUXF* usage and LUT combining potentially increase congestion because they tend to increase the input connectivity for the slices. The Flow_AlternateRoutability synthesis strategy and directive instructs the synthesis tool to avoid MUXF* structures and not generate any additional LUT combining.

***Note:*** Disabling these resources can result in increased power. Use this method only when needed to achieve timing closure.

In addition, the `opt_design` command provides an optional MUX optimization phase to remap MUXF* structures to LUT3 primitives to improve routability. You can use the `-muxf_remap` option to remap all of the MUXF* cells. Alternatively, set the MUXF_REMAP property to TRUE on a select number of cells in the congested region to limit the scope of the MUX remapping. Any MUXF* cells with the MUXF_REMAP property set to TRUE automatically trigger the MUX optimization phase during `opt_design` and are remapped to LUT3s.

The following figure shows a 16-1 MUX before and after the MUXF* optimization.

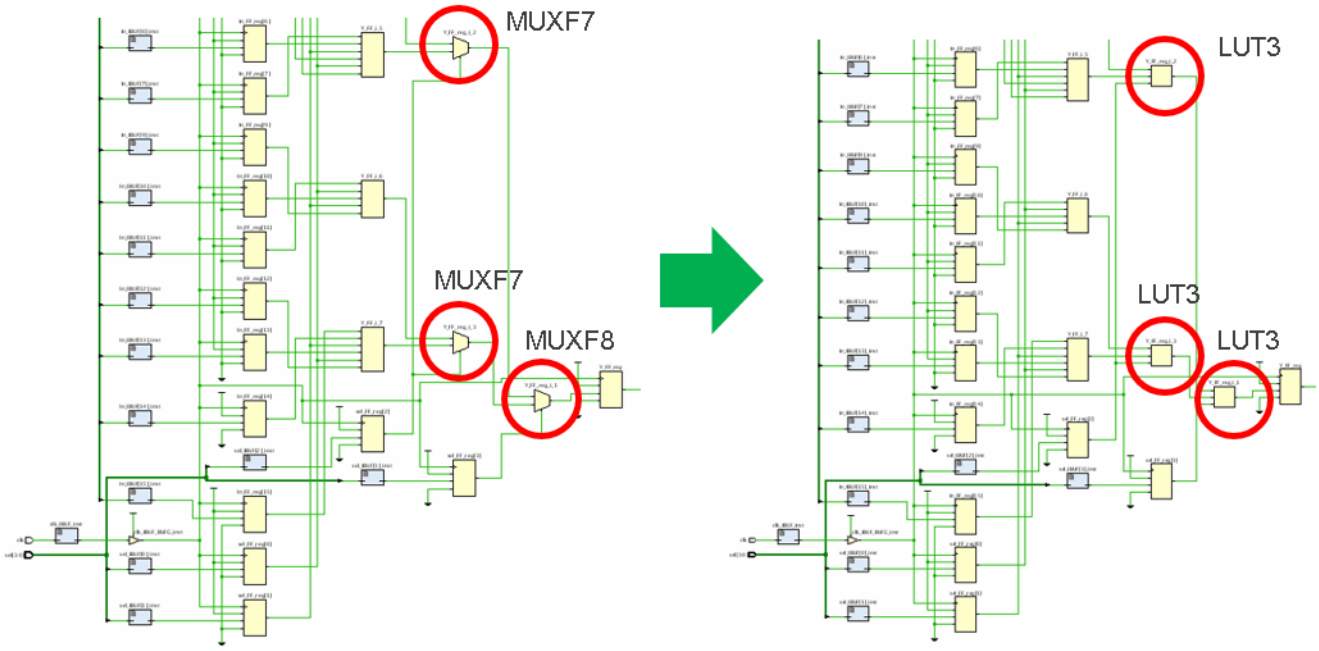



*Figure 5-34:* **Netlist Before and After MUX Optimization**

To further optimize the netlist after performing MUX optimization, use the `-remap` option with the `-muxf_remap` option. This combines the LUT3 primitives that are generated by the MUXF* optimization with connected logic if possible.

***Note:*** If you are using Synplify Pro for synthesis, you can use the **Enable Advanced LUT Combining** option in the Implementation Options under the Device tab. This option is ON by default. If you are modifying the Synplify Pro project file (*prj), the following is specified `set_option -enable_prepacking 0|1` (default is "1").

You can use the following command to highlight LUT combining in your design:

```
highlight_objects [get_cells -hier -filter {SOFT_HLUTNM != "" || HLUTNM != ""}]
```

The following figure shows the placer estimated congestion (purple) and the MUXF* usage (yellow) in those areas. In this case, reducing the MUXF* usage can help alleviate congestion.

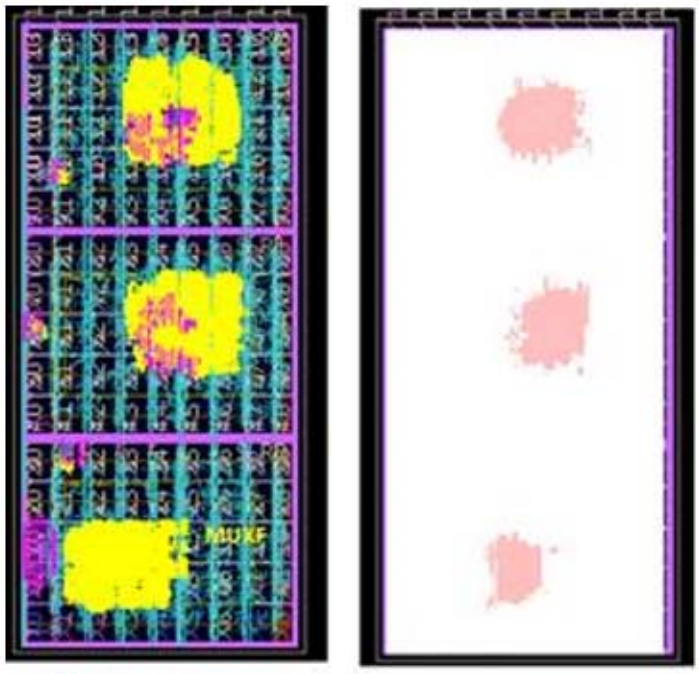



*Figure 5-35:* **Placer Estimated Congestion and MUXF* Usage**

The following figure shows the horizontal congestion of a design with and without LUT combining. The cells utilizing LUT combining are highlighted in purple.

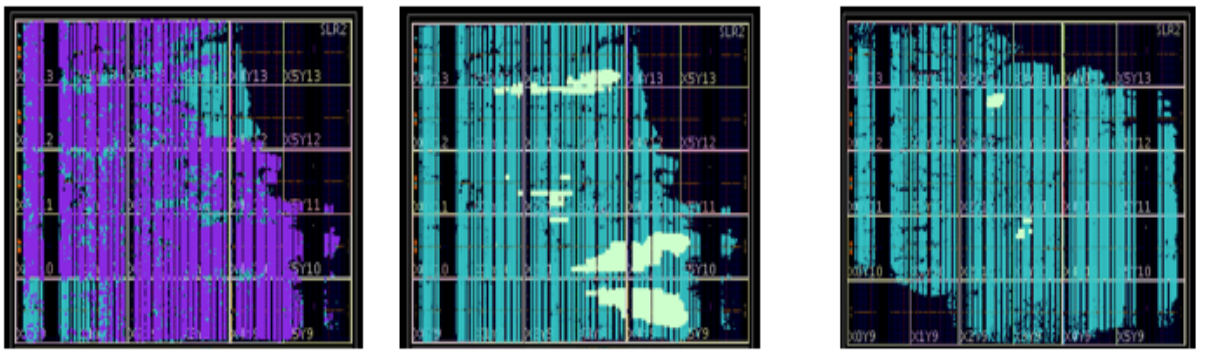



*Figure 5-36:* **Effect of LUT Combining on Horizontal Congestion**

### *Use Block-Level Synthesis Strategies*

In some cases it might be advantageous to try different synthesis options or strategies for different modules in the design. For example, one module might require the use of MUXF* resources to implement a timing critical function, while the rest of the design might benefit from implementation of logic in LUTs rather than MUXF* to reduce congestion. In that case, set the PERFORMANCE_OPTIMIZED strategy for the timing critical module while the rest of the design is being synthesized using the Flow_AlternateRoutability strategy to reduce congestion. For more information, see Block-Level Synthesis Strategy in Chapter 4.

### *Limit High-Fanout Nets in Congested Areas*

**TIP:** *This optimization technique is automatically applied by the* `report_qor_suggestions` *Tcl command.*

High fanout nets that have tight timing constraints require tightly clustered placement to meet timing. This can cause localized congestion as shown in the following figure. High fanout nets can also contribute to congestion by consuming routing resources that are no longer available for other nets in the congestion window.

To analyze the impact of high fanout non-global nets on routability in the congestion window you can:

- Select the leaf cells of the top hierarchical modules in the congestion window
- Use the find command (**Edit > Find**) to select all of the nets of the selected cell objects (filter out Global Clocks, Power, and Ground nets)
- Sort the nets in decreasing Flat Pin Count order
- Select the top fan-out nets to show them in relation to the congestion window

This can quickly help you identify high-fanout nets which potentially contribute to congestion.

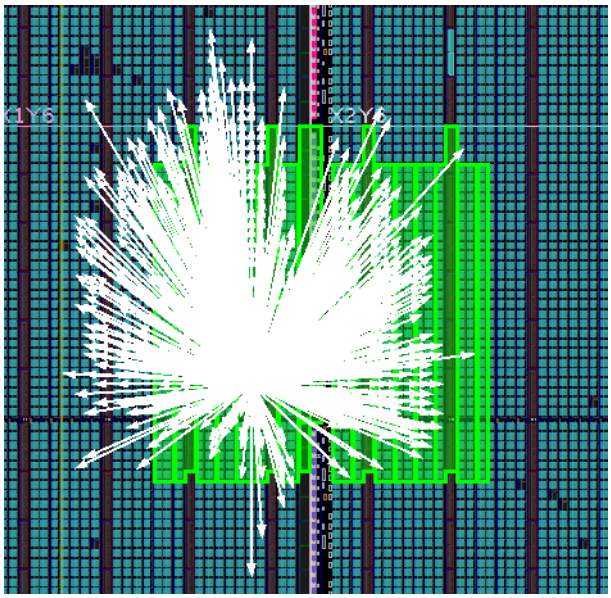



*Figure 5-37:* **High-Fanout Nets in Congestion Window**

For high fanout nets with tight timing constraints in the congestion window, replicating the driver will help relaxing the placement constraints and alleviate congestion.

High fanout nets (fanout > 5000) with sufficient positive timing slack can be routed on global clock resources instead of fabric resources. The placer automatically routes high fanout nets with fanout > 1000 on global routing resources if those resources are available towards the end of the placer step. This optimization only occurs if it does not degrade timing.

You can also set the property CLOCK_BUFFER_TYPE=BUFG on the net and let synthesis or logic optimization automatically insert the buffer prior to the placer step. Review the new inserted buffer placement along with its driver and loads placement after `place_design` to verify that it is optimal. If it is not optimal, use the CLOCK_REGION constraint (UltraScale devices only) or LOC constraint (7 series devices only) on the clock buffer to control its placement.

## Tuning the Compilation Flow

The default compilation flow provides a quick way to obtain a baseline of the design and start analyzing the design if timing is not met. After initial implementation, tuning the compilation flow might be required to achieve timing closure.

### *Strategies and Directives*

Strategies and directives can be used to increase the implementation solution space and find the optimal solution for your design. The strategies are applied globally to a project implementation run, while the directives can be set individually on each step of the

implementation flow in both project and non-project modes. The pre-defined strategies should be tried first before trying to customize the flow with directives. Xilinx does not recommend running the SSI technology strategies for a non-SSI technology device.

If timing cannot be met with the predefined strategies, you can manually explore a custom combination of directives. Because placement typically has a large impact on overall design performance, it can be beneficial to try various placer directives with only the I/O location constraints and with no other placement constraints. By reviewing both WNS and TNS of each placer run (these values can be found in the placer log), you can select two or three directives that provide the best timing results as a basis for the downstream implementation flow.

**TIP:** *For a list of directives and a short description their functions, enter the implementation command followed by the* `-help` *option (for example,* `place_design -help`*), or see this link in the Vivado Design Suite User Guide: Implementation (UG904) [Ref 19].*

For each of these checkpoints, several directives for `phys_opt_design` and `route_design` can be tried and again only the runs with the best estimated or final WNS/TNS should be kept. In Non-Project Mode, you must explicitly describe the flow with a Tcl script and save the best checkpoints. In Project Mode, you can create individual implementation runs for each placer directive, and launch the runs up to the placement step. You would continue implementation for the runs that have the best results after the placer step (as determined by the Tcl-post script).

Physical constraints (Pblocks and DSP and RAM macro constraints) can prevent the placer from finding the most optimal solution. Xilinx therefore recommends that you run the placer directives without any Pblock constraints. The following Tcl command can be used to delete any Pblocks before placement with directives commences:

```
delete_pblock [get_pblocks *]
```

Running `place_design -directive <directive>` and analyzing placement of the best results can also provide a template for floorplanning the design or reusing the placement of block RAM macros or DSP macros, which can stabilize the flow from run to run.

### Optimization Iterations

Sometimes it is advantageous to iterate through a command multiple times to obtain the best results. For example, it might be helpful to first run `phys_opt_design` with the `force_replication_on_nets` option in order to optimize some critical nets that appear to have an impact on WNS during route.

Next run `phys_opt_design` with any of the directives to improve the overall WNS of the design.

In Non-Project Mode, use the following commands:

```
phys_opt_design -force_replication_on_nets [get_nets -hier *phy_reset*]
phys_opt_design -directive <directive name>
```

In Project Mode, the same results can be achieved by running the first `phys_opt_design` command as part of a Tcl-pre script for a `phys_opt_design` run step which will run using the `-directive` option.

### Overconstraining the Design

When the design fails timing by a small amount after route, it is usually due to a small timing margin after placement. It is possible to increase the timing budget for the router by tightening the timing requirements during placement and physical optimization. To accomplish this, Xilinx recommends using the `set_clock_uncertainty` constraint for the following reasons:

- It does not modify the clock relationships (clock waveforms remain unchanged).
- It is additive to the tool-computed clock uncertainty (jitter, phase error).
- It is specific to the clock domain or clock crossing specified by the `-from` and `-to` options.
- It can easily be reset by applying a null value to override the previous clock uncertainty constraint.

In any case, Xilinx recommends that you:

- Overconstrain only the clocks or clock crossing that cannot meet setup timing.
- Use the `-setup` option to tighten the setup requirement only

  ***Note:*** If you do not specify this option, both setup and hold requirements are tightened.
- Reset the extra uncertainty before running the router step.

See the following example:

A design misses timing by -0.2 ns on paths with the `clk1` clock domain and on paths from `clk2` to `clk3` by -0.3 ns before and after route.

1. Load netlist design and apply the normal constraints.
2. Apply the additional clock uncertainty to overconstrain certain clocks.
   a. The value should be at least the amount of violation.
   b. The constraint should be applied only to setup paths.

```
set_clock_uncertainty -from clk0 -to clk1 0.3 -setup
set_clock_uncertainty -from clk2 -to clk3 0.4 -setup
```

3. Run the flow up to the router step. It is best if the pre-route timing is met.
4. Remove the extra uncertainty.

```
set_clock_uncertainty -from clk0 -to clk1 0 -setup
set_clock_uncertainty -from clk2 -to clk3 0 -setup
```

5. Run the router.

After the router, you can review the timing results to evaluate the benefits of overconstraining. If timing was met after placement but still fails by some amount after route, you can increase the amount of uncertainty and try again.

**RECOMMENDED:** *Do not overconstrain beyond 0.5 ns. Overconstraining the design can result in increased power for the implementation as well as an increase in runtime.*

### *Using Incremental Compile*

You can use incremental compile to reduce implementation runtime and produce more predictable results. Xilinx recommends making incremental compile part of your standard timing closure strategies. For more information, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

This section covers recommendations for both high and low reuse modes. High reuse mode is enabled when cell reuse is equal to or greater than 75%. Low reuse mode is enabled when cell reuse is less than 75%.

### Choose a High Quality Reference Checkpoint

Choose a high quality reference checkpoint as follows:

- Use a reference checkpoint that meets timing or is close to meeting timing.

  If the reference checkpoint is close to meeting timing, Xilinx recommends running the `route_design -tns_cleanup` command to clean up paths that are not the worst case path.

- Ensure that the options used for `synth_design` and `opt_design` are the same for both the reference and incremental runs.
- When multiple reference checkpoints meet timing and have matching `synth_design` and `opt_design` options, choose the checkpoint with the least congestion.

  Examine the Vivado router log file and look for initial estimated congestion. For details, see Identifying Congestion.

### Limit Differences for High Reuse Mode

In high reuse mode, the amount of reuse depends on the differences between the reference and incremental runs, including differences in the following:

- Source code and IP
- Tool options used for `synth_design` and `opt_design`
- Tool versions

For best results, maximize reuse as follows:

- Run incremental flows in parallel with standard timing closure flows.
- If you have multiple runs with different `synth_design` and `opt_design` options, use a different reference checkpoint for each run.
- Use the same tool versions across runs if possible. Newer tool versions might include changes to algorithms and thresholds for existing options.
- Use `report_incremental_reuse -hierarchical` to show matching percentages per hierarchy area. If hierarchy areas that should match are not showing high matching percentages, compare the tool options and tool versions of the reference design and incremental design.

Avoid the following synthesis differences, which can impact reuse:

- Enabling register retiming
- Preserving or dissolving logical hierarchy
- Changing constraints

***Note:*** Options used with `place_design`, `route_design`, and `phys_opt_design` in the reference run do not affect reuse.

### Select Incremental Compile Directives for High Reuse Mode

In high reuse mode, the repeatability of the reference checkpoint results is the priority. You can fine tune the behavior of the tool with the following supported directives:

- **Default**: Targets the WNS from the reference run. This helps maintain consistency with the reference checkpoint and runtime.
- **Explore**: Targets WNS = 0.0 ns. Use this directive when the reference run is very close to meeting timing, and you are willing to trade off consistency in results and runtime with more effort to try to meet timing. This mode can improve WNS by up to 100 ps on difficult designs. There is usually a runtime hit with this option.
- **Quick**: This option is intended for designs that easily meet timing with > 99% reuse. Typically, this option is used for ASIC emulation and prototype designs with minor changes that do not impact timing.

### Improve Timing QoR for High Reuse Mode

In high reuse mode, improve timing QoR as follows:

- Ensure the reference checkpoint has already closed timing.

  In the reference checkpoint, if WNS is > 0.100 ns, increase effort on the reference runs to improve the WNS. After WNS is within 0.100 ns, you can use the Explore directive to try to close timing.

- Run multiple incremental runs with different reference checkpoints.

  This would normally mean the reference checkpoints are generated with different placer directives and all close timing.

- Do not floorplan incremental runs.

  Pblock placement is overridden by reference checkpoint placement.

- Do not overconstrain the placer as described in Overconstraining the Design.

  Overconstraining the design in the incremental run can severely impact reuse, because the tools try to meet a target WNS that is artificially altered.

### Reduce QoR Variability for Low Reuse Mode

In low reuse mode, you can reuse particular cells (for example, a hierarchical cell in the design) or cell types (for example, DSPs or block RAMs). This can be effective when both of the following are true:

- Some design runs are showing that a design can meet timing but many runs do not.
- It is early in the design the flow or significant changes are still being made.

Reusing hierarchical cells is effective when placement of a particular cell is influencing the WNS significantly. Reusing DSPs, block RAMs, or both is useful in designs that have a relatively high density of these blocks.

To reuse particular cell or cell types:

- Analyze the reference runs, such as checking failing checkpoints to identify areas to target.
- After determining the area to target, compare a set of runs against a baseline set of runs to evaluate effectiveness.
- Use different `place_design` directives.

  ***Note:*** In low reuse mode, all directives from the standard flow are supported, and target WNS is always 0.00 ns.

To reuse only block memory placement, use the following Tcl script:

```
read_checkpoint -incremental routed.dcp \
   -only_reuse [get_cells -hier -filter {PRIMITIVE_TYPE =~ BMEM.*.* }] -fix_reuse
```

To reuse only DSP placement, use the following Tcl script:

```
read_checkpoint -incremental routed.dcp \
   -only_reuse [get_cells -hier -filter {PRIMITIVE_TYPE =~ MULT.dsp.* }] -fix_reuse
```

To reuse both Block Memory and DSP placement, use the following Tcl script:

```
read_checkpoint -incremental routed.dcp \
   -only_reuse [get_cells -hier -filter {PRIMITIVE_TYPE =~ BMEM.*.* }] \
   -only_reuse [get_cells -hier -filter {PRIMITIVE_TYPE =~ MULT.dsp.* }] -fix_reuse
```

To reuse hierarchy in a particular hierarchical cell and all hierarchies below the cell, use the following Tcl script:

```
read_checkpoint -incremental routed.dcp \
   -only_reuse [get_cells <cell_name>] -fix_reuse
```

### *Considering Floorplan*

Floorplanning allows you to guide the tools, either through high-level hierarchy layout, or through detail placement. This can provide improved QoR and more predictable results. You can achieve the greatest improvements by fixing the worst problems or the most common problems. For example, if there are outlier paths that have significantly worse slack, or high levels of logic, fix those paths first by grouping them in a same region of the device through a Pblock. Limit floorplanning only to portions of design that need additional user intervention through floorplanning, rather than floorplanning the entire design.

Floorplanning logic that is connected to the I/O to the vicinity of the I/O can sometimes yield good results in terms of predictability from one compilation to the next. In general, it is best to keep the size of the Pblocks to a clock region. This provides the most flexibility for the placer. Avoid overlapping Pblocks, as these shared areas can potentially become more congested. Where there is a high number of connecting signals between two Pblocks consider merging them into a single Pblock. Minimize the number of nets that cross Pblocks.

**TIP:** *When upgrading to a newer version of the Vivado Design Suite, first try compiling without Pblocks or with minimal Pblocks (i.e. only SLR level Pblocks) to see if there are any timing closure challenges. Pblocks that previously helped to improve the QoR might prevent place and route from finding the best possible implementation in the newer version of the tools.*

### *Group Critical Logic*

Grouping critical logic to avoid crossing SLR or I/O columns can help improve the critical path of a design. The following figure shows two examples of a large FIFO implemented with 29 FIFO36E2 primitives. The critical path is from the WRRSTBUSY pin of every FIFO36E2 in the group through 5 LUTs to the WREN pin of every FIFO36E2 in the group.

- On the left, the example shows that the placer was unable to find the most optimal placement of the path, because block RAM utilization was high. FIFO36E2 primitives are marked in red.
- On the right, the example shows that the placer was able to meet timing, because the FIFO36E2 blocks were grouped in a rectangle that avoided the configuration column crossing. FIFO36E2 primitives are marked in green.

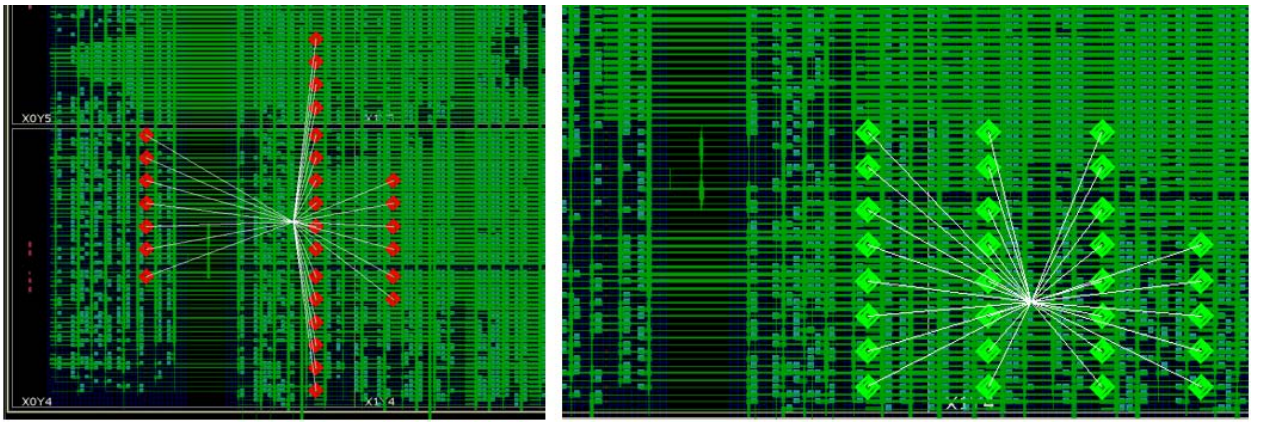


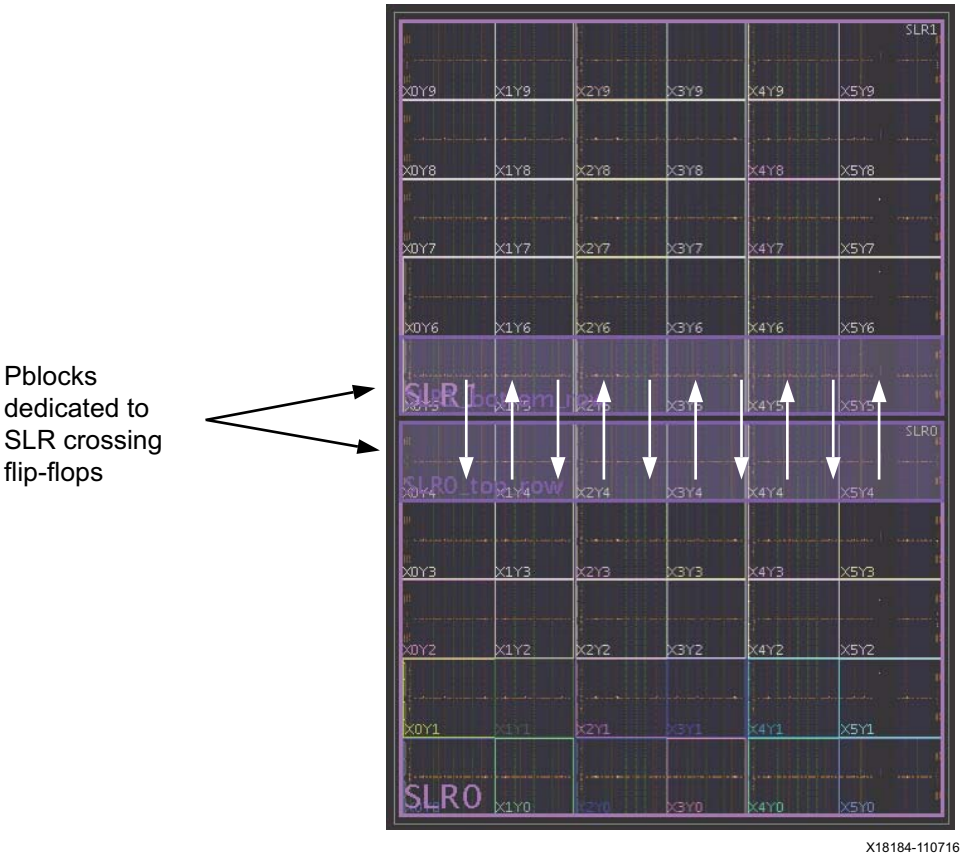

*Figure 5-38:* **FIFO Locations Avoiding the Configuration Column**

### *SSI Technology Considerations*

Stacked silicon interconnect (SSI) technology devices consist of multiple super logic regions (SLRs), joined by an interposer. The interposer connections are called super long lines (SLLs). There is some delay penalty when crossing from one SLR to another. To minimize the impact of the SLL delay on your design, floorplan the design so that SLR crossings are not part of the critical path. Minimizing SLR crossings through floorplanning by keeping a Pblock within one SLR only can also improve timing and routability of the design targeting SSI technology devices.

For high-performance designs, sufficient pipelining between the major hierarchies is required to ease global placement and SLR partitioning. When a design is challenging, SLR crossing points can change from run to run. In addition to defining SLR Pblocks, you can create additional Pblocks that are aligned to clock regions and located along the SLR boundary to constrain the crossing flip-flops. The following example shows an UltraScale ku115 SSI device with the following Pblocks:

- 2 SLR Pblocks: SLR0 and SLR1
- 2 SLR-crossing Pblocks: SLR0_top_row and SLR1_bottom_row




X18184-110716

*Figure 5-39:* **SLR-Crossing Pblock Example**

**IMPORTANT:** *Xilinx recommends using CLOCKREGION ranges instead of LAGUNA ranges for SLR-crossing Pblocks.*

For more information, see this link in *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) [Ref 21].

**VIDEO:** *For information on using floorplanning techniques to address design performance issues, see the* Vivado Design Suite QuickTake Video: Design Analysis and Floorplanning.

### *Reuse of Placement*

It is fairly easy to reuse the placement of block RAM macros and DSP macros. Reusing this placement helps to reduce the variability in results from one netlist revision to the next. These primitives generally have stable names. The placement is usually easy to maintain. Some placement directives result in better block RAM and DSP macro placement than others. You can try applying this improved macro placement from one placer run to others using different placer directives to improve QoR. A simple Tcl script that saves block RAM placement into an XDC file is shown below.

```
set_property IS_LOC_FIXED 1 \
  [get_cells -hier -filter {PRIMITIVE_TYPE =~ BMEM.bram.*}]
write_xdc bram_loc.xdc -exclude_timing
```

You can edit the `bram_loc.xdc` file to only keep block RAM location constraints and apply it for your consecutive runs.

**IMPORTANT:** *Do not reuse the placement of general slice logic. Do not reuse the placement for sections of the design that are likely to change.*

# Power Analysis and Optimization

Given the importance of power, the Vivado tools support methods for obtaining an accurate estimate for power, as well as providing some power optimization capabilities. For additional information refer to *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 22].

## Estimating Power Throughout the Flow

As your design flow progresses through synthesis and implementation, you must regularly monitor and verify the power consumption to be sure that thermal dissipation remains within budget. You can then take prompt remedial actions if power approaches your budget too closely.

The accuracy of the power estimates varies depending on the design stage when the power is estimated. To estimate power post-synthesis through implementation, run the `report_power` command, or open the Power Report in the Vivado IDE.

- Post Synthesis

  The netlist is mapped to the actual resources available in the target device.

- Post Placement

  The netlist components are placed into the actual device resources. With this packing information, the final logic resource count and configuration becomes available.

  This accurate data can be exported to the Xilinx Power Estimator spreadsheet. This allows you to:

  - Perform what-if analysis in XPE.
  - Provide the basis for accurately filling in the spreadsheet for future designs with similar characteristics.

- Post Routing

  After routing is complete all the details about routing resources used and exact timing information for each path in the design are defined.

In addition to verifying the implemented circuit functionality under best and worst case logic and routing delays, the simulator can also report the exact activity of internal nodes and include glitching. Power analysis at this level provides the most accurate power estimation before you actually measure power on your prototype board.

## Using the Power Constraints Advisor

The Power Constraint Advisor reports the tool-computed switching activity on all control signals in the design. For more information, see this link in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 22].

## Best Practices for Accurate Power Analysis

For accurate power analysis, make sure you have accurate clock constraints, I/O constraints, and signals. For more information, see this link in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 22].

## Reviewing the Design Power Distribution After Running Vivado Design Suite Power Analysis

You can review the total on-chip power and thermal properties as well as details of the power at the resource level to determine which parts of your design contribute most to the total power. For more information, see this link in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 22].

## Further Refining Control Signal Activity After Running Vivado Design Suite Power Analysis

When SAIF-based annotation has not been used for accurate power analysis, you can fine-tune the power analysis after doing the first level analysis. For more information, see this link in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 22].

## Power Optimization

If the power estimates are outside the budget, you must follow the steps described in the following sections to reduce power.

### *Analyzing Your Power Estimation and Optimization Results*

Once you have generated the power estimation report using `report_power`, Xilinx recommends the following:

- Examine the total power in the Summary section. Does the total power and junction temperature fit into your thermal and power budget?
- If the results are substantially over budget, review the power summary distribution by block type and by the power rails. This provides an idea of the highest power consuming blocks.
- Review the Hierarchy section. The breakdown by hierarchy provides a good idea of the highest power consuming module. You can drill down into a specific module to determine the functionality of the block. You can also cross-probe in the GUI to determine how specific sections of the module have been coded, and whether there are power efficient ways to recode it.

### *Running Power Optimization*

**TIP:** *To maximize the impact of power optimizations, see Coding Styles to Improve Power in Chapter 3.*

Power optimization works on the entire design or on portions of the design (when `set_power_opt` is used) to minimize power consumption.

Power optimization can be run either pre-place or post-place in the design flow, but not both. The pre-place power optimization step focusses on maximizing power saving. This can result (in rare cases) in timing degradation. If preserving timing is the primary goal, Xilinx recommends the post-place power optimization step. This step performs only those power optimizations that preserve timing.

In cases where portions of the design should be preserved due to legacy (IP) or timing considerations, use the `set_power_opt` command to exclude those portions (such as specific hierarchies, clock domains, or cell types) and rerun power optimization.

### *Using the Power Optimization Report*

To determine the impact of power optimizations, run the following command in the Tcl console to generate a power optimization report:

```
report_power_opt -file myopt.rep
```

### *Using the Timing Report to Determine the Impact of Power Optimization*

Power optimization works to minimize the impact on timing while maximizing power savings. However, in certain cases, if timing degrades after power optimization, you can employ a few techniques to offset this impact.

Where possible, identify and apply power optimizations only on non-timing critical clock domains or modules using the `set_power_opt` XDC command. If the most critical clock domain happens to cover a large portion of the design or consumes the most power, review critical paths to see if any cells in the critical path were optimized by power optimization.

Objects optimized by power optimization have an `IS_CLOCK_GATED` property on them. Exclude these cells from power optimization.

To locate clock gated cells, run the following Tcl command:

```
get_cells -hier -filter {IS_CLOCK_GATED==1}
```

# Configuration and Debug

After successfully completing the design implementation, the next step is to load the design into the FPGA and run it on hardware. Configuration is the process of loading application-specific data (a bitstream) into the internal memory of the FPGA device. Debug is required if the design does not meet expectations on the hardware.

See the following resources for details on configuration and debug software flows and commands:

- *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 24]
- *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 14]
- *7 Series FPGAs Configuration User Guide* (UG470) [Ref 38]
- *UltraScale Architecture Configuration User Guide* (UG570) [Ref 38]
- Vivado Design Suite QuickTake Video: How To Use the "write_bitstream" Command in Vivado

## Configuration

You must first successfully synthesize and implement your design in order to create a bitstream image. Once the bitstream has been generated and all DRCs are analyzed and corrected, you can load bitstream onto the device using one of the following methods:

- Direct Programming

  The bitstream is loaded directly to the device using a cable, processor, or custom solution.

- Indirect Programming

  The bitstream is loaded into an external flash memory. The flash memory then loads the bitstream into the device.

You can use the Vivado tools to accomplish the following:

- Create the FPGA bitstream (`.bit` or `.rbt`).
- Select **Tools > Edit Device Properties** to review the configuration settings for bitstream generation.
- Format the bitstream into flash programming files (`.mcs`).
- Program the device using either of the following methods:
  - Directly program the device.
  - Indirectly program the attached configuration flash device.

    Flash devices are non-volatile devices and must be erased before programming. Unless a full chip erase is specified, only the address range covered by the assigned MCS is erased.

**IMPORTANT:** *The Vivado Design Suite Device Programmer can use JTAG to read the Status register data on Xilinx devices. In case of a configuration failure, the Status register captures the specific error conditions that can help identify the cause of a failure. In addition, the Status register allows you to verify the Mode pin settings M[2:0] and the bus width detect. For details on the Status register, see the Configuration User Guide [Ref 38] for your device.*

**TIP:** *If configuration is not successful, you can use a JTAG readback/verify operation on the FPGA device to determine whether the intended configuration data was loaded correctly into the device.*

## Debugging

In-system debugging allows you to debug your design in real time on your target device. This step is needed if you encounter situations that are extremely difficult to replicate in a simulator.

For debug, you provide your design with special debugging hardware that allows you to observe and control the design. After debugging, you can remove the instrumentation or special hardware to increase performance and logic reduction.

Debugging an FPGA design is a multistep, iterative process. Like most complex problems, it is best to break the FPGA design debugging process down into smaller parts by focusing on getting smaller sections of the design working one at a time rather than trying to get the whole design to work at once.

Though the actual debugging step comes after you have successfully implemented your design, Xilinx recommends planning how and where to debug early in the design cycle. You can run all necessary commands to perform programming of the FPGA devices and in-system debugging of the design from the Program and Debug section of the Flow Navigator window in the Vivado IDE.

Following are the debug steps:

1. Probing: Identify the signals in your design that you want to probe and how you want to probe them.
2. Implementing: Implement the design that includes the additional debug IP attached to the probed nets.
3. Analyzing: Interact with the debug IP contained in the design to debug and verify functional issues.
4. Fixing phase: Fix any bugs and repeat as necessary.

For more information, see *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 24].

### Probing the Design

The Vivado tools provide several methods to add debug probes in your design. The table below explains the various methods, including the pros and cons of each method.

*Table 5-10:* **Debugging Flows**

| Debugging Flow Name | Flow Steps | Pros/Cons |
|---|---|---|
| HDL instantiation probing flow | Explicitly attach signals in the HDL source to an ILA debug core instance. | • You have to add/remove debug nets and IP from your design manually, which means that you will have to modify your HDL source<br>• This method provides the option to probe at the HDL design level.<br>• It is easy to make mistakes when generating, instantiating, and connecting debug cores. |
| Netlist insertion probing flow (recommended) | Use one of the following two methods to identify the signal for debug:<br>• Use the MARK_DEBUG attribute to mark signals for debug in the source RTL code.<br>• Use the MARK_DEBUG right-click menu option to select nets for debugging in the synthesized design netlist.<br>Once the signal is marked for debug, use the Set up Debug wizard to guide you through the Netlist Insertion probing flow. | • This method is the most flexible with good predictability.<br>• This method allows probing at different design levels (HDL, synthesized design, system design).<br>• This method does not require HDL source modification. |
| Tcl-based netlist insertion probing flow | Use the `set_property` Tcl command to set the MARK_DEBUG property on debug nets then use Netlist insertion probing Tcl commands to create debug cores and connect them to debug nets.<br>See Modifying the Implemented Netlist to Replace Existing Debug Probes for post-synthesis insertion of ILA core. | • This method provides fully automatic netlist insertion<br>• You can turn debugging on or off by modifying the Tcl commands.<br>• This method does not require HDL source modification. |

### *Choosing Debug Nets*

Xilinx makes the following recommendations for choosing debug nets:

- Probe nets at the boundaries (inputs or outputs) of a specific hierarchy. This method helps isolate problem areas quickly. Subsequently, you can probe further in the hierarchy if needed.
- Do not probe nets in between combinatorial logic paths. If you add MARK_DEBUG on nets in the middle of a combinatorial logic path, none of the optimizations applicable at the implementation stage of the flow are applied, resulting in sub-par QOR results.
- Probe nets that are synchronous in order to get cycle accurate data capture.

### *Retaining Names of Debug Probe Nets Using MARK_DEBUG*

You can mark a signal for debug either at the RTL stage or post-synthesis. The presence of the MARK_DEBUG attribute on the nets ensures that the nets are not replicated, retimed, removed, or otherwise optimized. You can apply the MARK_DEBUG attribute on top level ports, nets, hierarchical module ports and nets internal to hierarchical modules. This method is most likely to preserve HDL signal names post synthesis. Nets marked for debugging are shown in the Unassigned Debug Nets folder in the Debug window post synthesis.

Add the `mark_debug` attribute to HDL files as follows:

VHDL:

```
attribute mark_debug : string;
attribute keep : string;
attribute mark_debug of sine     : signal is "true";
```

Verilog:

```
(* mark_debug = "true" *) wire sine;
```

You can also add nets for debugging in the post-synthesis netlist. These methods do not require HDL source modification. However, there may be situations where synthesis might not have preserved the original RTL signals due to netlist optimization involving absorption or merging of design structures. Post-synthesis, you can add nets for debugging in any of the following ways:

- Select a net in any of the design views (such as the Netlist or Schematic windows), then right-click and select **Mark Debug**.
- Select a net in any of the design views, then drag and drop the net into the Unassigned Debug Nets folder.
- Use the net selector in the Set Up Debug Wizard.

- Set the MARK_DEBUG property using the properties window or the Tcl Console.

```
set_property mark_debug true [get_nets -hier [list {sine[*]}]]
```

  This applies the `mark_debug` property on the current, open netlist. This method is flexible, because you can turn MARK_DEBUG on and off through the Tcl command.

## *Using ILA Cores*

The Integrated Logic Analyzer (ILA) core allows you to perform in-system debugging of post-implementation designs on an FPGA device. Use this core when you need to monitor signals in the design. You can also use this feature to trigger on hardware events and capture data at system speeds.

### ILA Core and Timing Considerations

The configuration of the ILA core has an impact in meeting the overall design timing goals. Follow the recommendations below to minimize the impact on timing:

- Choose probe width judiciously. The bigger the probe width the greater the impact on both resource utilization and timing.
- Choose ILA core data depth judiciously. The bigger the data depth the greater the impact on both block RAM resource utilization and timing.
- Ensure that the clocks chosen for the ILA cores are free-running clocks. Failure to do so could result in an inability to communicate with the debug core when the design is loaded onto the device.
- Ensure that the clock going to the `dbg_hub` is a free running clock. Failure to do so could result in an inability to communicate with the debug core when the design is loaded onto the device. You can use the `connect_debug_port` Tcl command to connect the `clk` pin of the debug hub to a free-running clock.
- Close timing on the design prior to adding the debug cores. Xilinx does not recommend using the debug cores to debug timing related issues.
- If you still notice that timing has degraded due to adding the ILA debug core and the critical path is in the `dbg_hub`, perform the following steps:
  a. Open the synthesized design.
  b. Find the `dbg_hub` cell in the netlist.
  c. Go to the Properties of the `dbg_hub`.
  d. Find property C_CLK_INPUT_FREQ_HZ.
  e. Set it to frequency (in Hz) of the clock that is connected to the `dbg_hub`.
  f. Find property C_ENABLE_CLK_DIVIDER and enable it.
  g. Re-implement design.

- Make sure the clock input to the ILA core is synchronous to the signals being probed. Failure to do so results in timing issues and communication failures with the debug core when the design is programmed into the device.
- Make sure that the design meets timing before running it on hardware. Failure to do so results in unreliable results.

The following table shows the impact of using specific ILA features on design timing and resources.

***Note:*** This table is based on a design with one ILA and does not represent all designs.

*Table 5-11:* **Impact of ILA Features on Design Timing and Resources**

| ILA Feature | When to Use | Timing | Area |
|---|---|---|---|
| **Capture Control/ Storage Qualification** | • To capture relevant data<br>• To make efficient use of data capture storage (block RAM) | Medium to High Impact | • No additional block RAMs<br>• Slight increase in LUT/FF count |
| **Advanced Trigger** | • When BASIC trigger conditions are insufficient<br>• To use complex triggering to focus in on problem area | High Impact | • No additional block RAMs<br>• Moderate increase in LUT/FF count |
| **Number of Comparators per Probe Port**<br>***Note:*** Maximum is 4. | To use probe in multiple conditionals:<br>• 1-2 for Basic<br>• 1-4 for Advanced<br>• +1 for Capture Control | Medium to High Impact | • No additional block RAMs<br>• Slight to moderate increase in LUT/FF count |
| **Data Depth** | To capture more data samples | High Impact | • Additional block RAMs per ILA core<br>• Slight increase in LUT/FF count |
| **ILA Probe Port Width** | To debug a large bus versus a scalar | Medium Impact | • Additional block RAMs per ILA core<br>• Slight increase in LUT/FF count |
| **Number of Probes Ports** | To probe many nets | Low Impact | • Additional block RAMs per ILA core<br>• Slight increase in LUT/FF count |

**TIP:** *In the early stages of FPGA designs, there are usually a lot of spare resources in the FPGA that can be used for debugging.*

For designs with high-speed clocks, consider the following:

- Limit the number and width of signals being debugged.
- Pipeline the input probes to the ILA (C_INPUT_PIPE_STAGES), which enables extra levels of pipe stages.

For designs with limited MMCM/BUFG availability, consider clocking the debug hub with the lowest clock frequency in the design instead of using the clock divider inside the debug hub.

### *Debugging Designs in Vivado IP Integrator*

The Vivado IP integrator provides different ways to set up your design for debugging. You can use one of the following flows to add debug cores to your IP integrator design. The flow you choose depends on your preference and the types of nets and signals that you want to debug.

- Debug interfaces, nets, or both in the block design using the System ILA core

  Use this flow to:

  - Perform hardware-software co-verification using the cross-trigger feature of a MicroBlaze device or Zynq-7000 All Programmable (AP) SoC.
  - Verify the interface-level connectivity.
- Netlist insertion flow

  Use this flow to analyze I/O ports and internal nets in the post-synthesized design.

***Note:*** You can also use a combination of both flows to debug your design.

For more information on using System ILA in your IP integrator design, see the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 26].

### *Debugging AXI Interfaces in Vivado Hardware Manager*

The System ILA IP in IP integrator allows you to perform in-system debugging of post-implemented designs on a Xilinx device. Use this feature when there is a need to monitor interfaces and signals in the design.

If you inserted System ILA debug cores in your IP integrator block design, you can debug and monitor AXI transactions and read and write events in the Waveform window shown in the following figure. The Waveform window displays the interface slots, transactions, events, and signal groups that correspond to the interfaces probed by the System ILA IP.

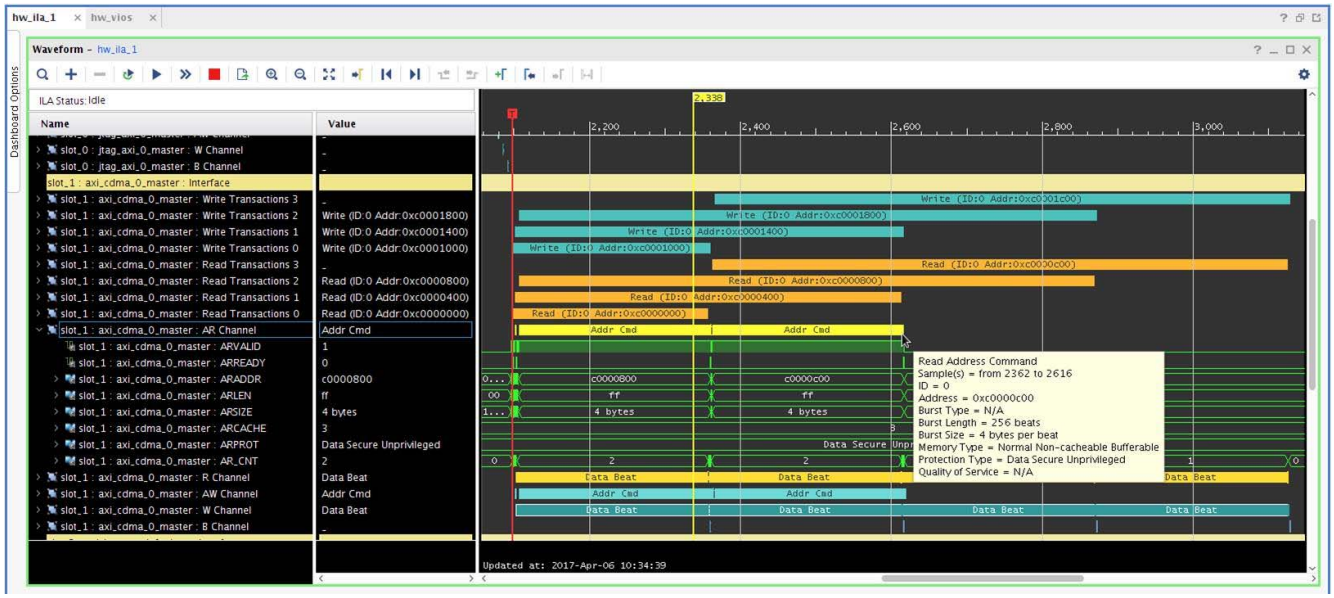



*Figure 5-40:* **Waveform Window**

For more information on System ILA and debugging AXI interfaces in the Vivado Hardware Manager, see this link and this link in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 24].

### *Using In-System IBERT*

The In-System IBERT core provides RX margin analysis through eye scan plots on the RX data of transceivers in UltraScale and UltraScale+™ devices. The core enables configuration and tuning of the GTH/GTY transceivers and is accessible through logic that communicates with the dynamic reconfiguration port (DRP) of the transceivers. You can use the core to change attribute settings as well as registers that control the values on the rxrate, rxlpmen, txdiffctrl, txpostcursor, and txprecursor ports.

The Vivado Serial I/O Analyzer in the Hardware Manager communicates with the core through JTAG when the design is programmed onto the device. There is only one instance of In-System IBERT required per design. In-System IBERT can work with all GTs used in the design. However, you must generate separate In-System IBERT cores according to the different GT types (for example, GTH, GTY).

Creating an In-System IBERT design with an internal system clock can prevent a scan from being performed. When creating an eye scan, the status changes from **In Progress** to **Incomplete**. Eye scan is incomplete when the internal system clock (MGTREFCLK) is connected to the clk/drpclk_i input port of In-System IBERT IP.

***Note:*** If needed, consider using an external clock, which does not exhibit this behavior. Alternatively, click any available link in the Vivado Serial I/O Analyzer. Go to the Properties window, and find the MB_RESET reg under the LOGIC field. Set it to 1 and then toggle back to 0. Rerun the eye scan or sweep.

For more information on this core, see the *In-System IBERT LogiCORE IP Product Guide* (PG246) [Ref 50].

### Running Debug-Related DRCs

The Vivado Design Suite provides debug-related DRCs, which are selected as part of the default rule deck when `report_drc` is run. The DRCs check for the following:

- Block RAM resources for the device are exceeded because of the current requirements of the debug core.
- Non-clock net is connected to the clock port on the debug core.
- Port on the debug core is unconnected.

### Generating AXI Transactions

Use the JTAG-to-AXI debug core to generate AXI transactions that interact with various AXI full and AXI lite slave cores in a system that is running on hardware. Instantiate this core in your design from the IP Catalog to generate AXI transactions and debug/drive AXI signals internal to your FPGA at run time. You also can use this core in designs without processors.

### Modifying the Implemented Netlist to Replace Existing Debug Probes

It is possible to replace debug nets connected to an ILA core in a placed and routed design checkpoint. You can do this by using the Engineering Change Order (ECO) flow. This is an advanced design flow used for designs that are nearing completion, where you need to swap nets connected to an existing ILA probe port. For information on using the ECO flow to modify nets on existing ILA cores, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904) [Ref 19].

### Inserting, Deleting, or Editing ILA cores on an Implemented Netlist

If you want to add, delete, or modify ILA cores (for example, resizing probe width, changing the data depth, etc.), Xilinx recommends that you use the Incremental Compile flow. The Incremental Compile flow for debug cores operates on a synthesized design or checkpoint (DCP) and uses a reference implemented checkpoint, ideally from a previous implementation run. This approach might save you time versus a complete re-implementation of the design. For information on using the Incremental Compile flow to insert, delete, or edit ILA cores, see this link in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 24].

### *Using Remote Debugging*

Xilinx provides multiple ways to debug or upgrade your design remotely:

- Use the Xilinx Hardware Server product to connect to a remote computer in the lab.
- Implement the Xilinx Virtual Cable (XVC) protocol to connect to a network-connected board.

For more information on using the Xilinx Virtual Cable protocol, see this link in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 24]. For more information on using an example design using the XVC flow with the PCIe core, see the *UltraScale+ Devices Integrated Block for PCI Express Product Guide* (PG213) [Ref 30].

Appendix A

# Additional Resources and Legal Notices

## Xilinx Resources

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

## Solution Centers

See the Xilinx Solution Centers for support on devices, tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

## Documentation Navigator and Design Hubs

Xilinx Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.
- At the Linux command prompt, enter `docnav`.

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

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note:*** For more information on Documentation Navigator, see the Documentation Navigator page on the Xilinx website.

# References

These documents provide supplemental material useful with this guide.

1. Vivado® Design Suite Documentation
2. UltraFast™ Design Methodology Quick Reference Guide (UG1231)
3. UltraFast Design Methodology Checklist (XTP301)

## Vivado Design Suite User and Reference Guides

4. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
5. *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899)
6. *Vivado Design Suite User Guide: Design Flows Overview* (UG892)
7. *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
8. *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894)
9. *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)
10. *Vivado Design Suite User Guide: Designing with IP* (UG896)
11. *Vivado Design Suite User Guide: Embedded Processor Hardware Design* (UG898)
12. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
13. *Vivado Design Suite User Guide: Getting Started* (UG910)
14. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
15. *Vivado Design Suite Properties Reference Guide* (UG912)
16. *Vivado Design Suite User Guide: Synthesis* (UG901)
17. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
18. *Vivado Design Suite User Guide: Using Constraints* (UG903)
19. *Vivado Design Suite User Guide: Implementation* (UG904)
20. *Vivado Design Suite User Guide: Hierarchical Design* (UG905)
21. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906)
22. *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907)
23. *Xilinx Power Estimator User Guide* (UG440)
24. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
25. *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909)
26. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994)

27. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
28. *Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide* (UG953)
29. *UltraScale Architecture Libraries Guide* (UG974)
30. *UltraScale+ Devices Integrated Block for PCI Express Product Guide* (PG213)

## Vivado Design Suite Tutorials

31. *Vivado Design Suite Tutorial: High-Level Synthesis* (UG871)
32. *Vivado Design Suite Tutorial: Design Flows Overview* (UG888)
33. *Vivado Design Suite Tutorial: Logic Simulation* (UG937)
34. *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* (UG940)
35. *Vivado Design Suite Tutorial: Partial Reconfiguration* (UG947)

## Other Xilinx Documentation

36. *7 Series FPGAs PCB Design Guide* (UG483)

    *UltraScale Architecture PCB Design User Guide* (UG583)

    *Zynq-7000 All Programmable SoC PCB Design Guide* (UG933)

37. *UltraFast Embedded Design Methodology Guide* (UG1046)
38. *7 Series FPGAs Configuration User Guide* (UG470)

    *UltraScale Architecture Configuration User Guide* (UG570)

39. *7 Series FPGAs SelectIO Resources User Guide* (UG471)

    *UltraScale Architecture SelectIO Resources User Guide* (UG571)

40. *7 Series Clocking Resources Guide* (UG472)

    *UltraScale Architecture Clocking Resources User Guide* (UG572)

41. *UltraScale Architecture GTH Transceivers User Guide* (UG576)

    *UltraScale Architecture GTY Transceivers Advance Specification User Guide* (UG578)

42. *UltraScale Architecture Gen3 Integrated Block for PCI Express® LogiCORE IP Product Guide* (PG156)
43. *7 Series FPGAs Memory Resources User Guide* (UG473)
44. *7 Series FPGAs DSP48E1 Slice User Guide* (UG479)

45. *UltraScale Architecture DSP Slice User Guide* (UG579)
46. *7 Series FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide* (UG480)
47. *Reference System: Kintex-7 MicroBlaze System Simulation Using IP Integrator* (XAPP1180)
48. *Zynq-7000 SoC and 7 Series FPGAs Memory Interface Solutions User Guide* (UG586)
49. *UltraScale Architecture FPGAs Memory IP LogiCORE IP Product Guide* (PG150)
50. *In-System IBERT LogiCORE IP Product Guide* (PG246)
51. Xilinx White Paper: *Simulating FPGA Power Integrity Using S-Parameter Models* (WP411)
52. 7 Series Schematic Review Recommendations (XMP277)

    UltraScale Architecture Schematic Review Checklist (XTP344)

    *UltraScale+ FPGA and Zynq UltraScale+ MPSoC Schematic Review Checklist* (XTP427)
53. *UltraScale FPGA BPI Configuration and Flash Programming* (XAPP1220)
54. *BPI Fast Configuration and iMPACT Flash Programming with 7 Series FPGAs* (XAPP587)
55. *Using SPI Flash with 7 Series FPGAs* (XAPP586)
56. *SPI Configuration and Flash Programming in UltraScale FPGAs* (XAPP1233)
57. *Using Encryption to Secure a 7 Series FPGA Bitstream* (XAPP1239)

**TIP:** *A complete set of Xilinx documents can be accessed from Documentation Navigator. For more information, see Accessing Additional Documentation and Training.*

# Training Resources

1. UltraFast Design Methodology Training Course
2. Vivado Design Suite QuickTake Video: UltraFast Vivado Design Methodology
3. Vivado Design Suite QuickTake Video: Vivado Design Flows Overview
4. Vivado Design Suite QuickTake Video: Targeting Zynq Using Vivado IP Integrator
5. Vivado Design Suite QuickTake Video: Partial Reconfiguration in Vivado Design Suite
6. Vivado Design Suite QuickTake Video: Creating Different Types of Projects
7. Vivado Design Suite QuickTake Video: Managing Sources With Projects
8. Vivado Design Suite QuickTake Video: Using Vivado Design Suite with Revision Control
9. Vivado Design Suite QuickTake Video: Managing Vivado IP Version Upgrades

10. Vivado Design Suite QuickTake Video: I/O Planning Overview
11. Vivado Design Suite QuickTake Video: Configuring and Managing Reusable IP in Vivado
12. Vivado Design Suite QuickTake Video: How To Use the "write_bitstream" Command in Vivado
13. Vivado Design Suite QuickTake Video: Design Analysis and Floorplanning
14. Vivado Design Suite QuickTake Video: Introducing the UltraFast Design Methodology Checklist
15. Vivado Design Suite Video Tutorials

# Please Read: Important Legal Notices