AMD
XILINX

UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs

Design Closure

- Timing Closure
- Power Closure
- Configuration and Debug

Design Closure

Design closure consists of meeting all system performance, timing, and power requirements, and successfully validating the functionality in hardware. During the design closure phase where you are starting to run the design through the implementation tools, both timing and power considerations should be your top priorities. At this stage of design closure, estimation of design utilization, timing and power gain more accuracy. This presents an opportunity to reaffirm that the timing and power goals are achievable. To confirm the design can meet its requirements, Xilinx recommends conducting both a timing and power baseline. A timing baseline is largely about evaluating timing paths after accurate timing constraints have been defined. A power baseline needs to provide Vivado with the right toggle information to determine accurate dynamic power information.
By combining the analysis of power requirements and timing requirements, if one item is off significantly, a measure taken to resolve it can significantly impact the other. For example:

- An extreme measure might be necessary to meet a power budget such as scaling back features. This will make timing closure significantly easier as the part is less congested.
- A less extreme measure might involve adding logic to reduce switching. This might make timing closure more difficult, particularly if in a congested area of the die.

While many power saving items do not impact timing closure, it is possible that other items might make timing closure harder. Applying the required power saving techniques early will help you understand the true magnitude of the timing closure task.
Once you start to iterate from the baseline, you should recheck the power numbers when you make an improvement to timing. This ensures that you understand what change caused a regression. Generally, turning on wholesale power saving features early and then scaling back on individual items that are causing timing issues helps to strike the right balance of meeting design closure goals.
Conducting both power and timing analysis together and early in the design closure implementation phase will save engineering time and enable more accurate project planning. In addition, it creates time to allow engineering solutions to be explored than when this is realized later in the design cycle.

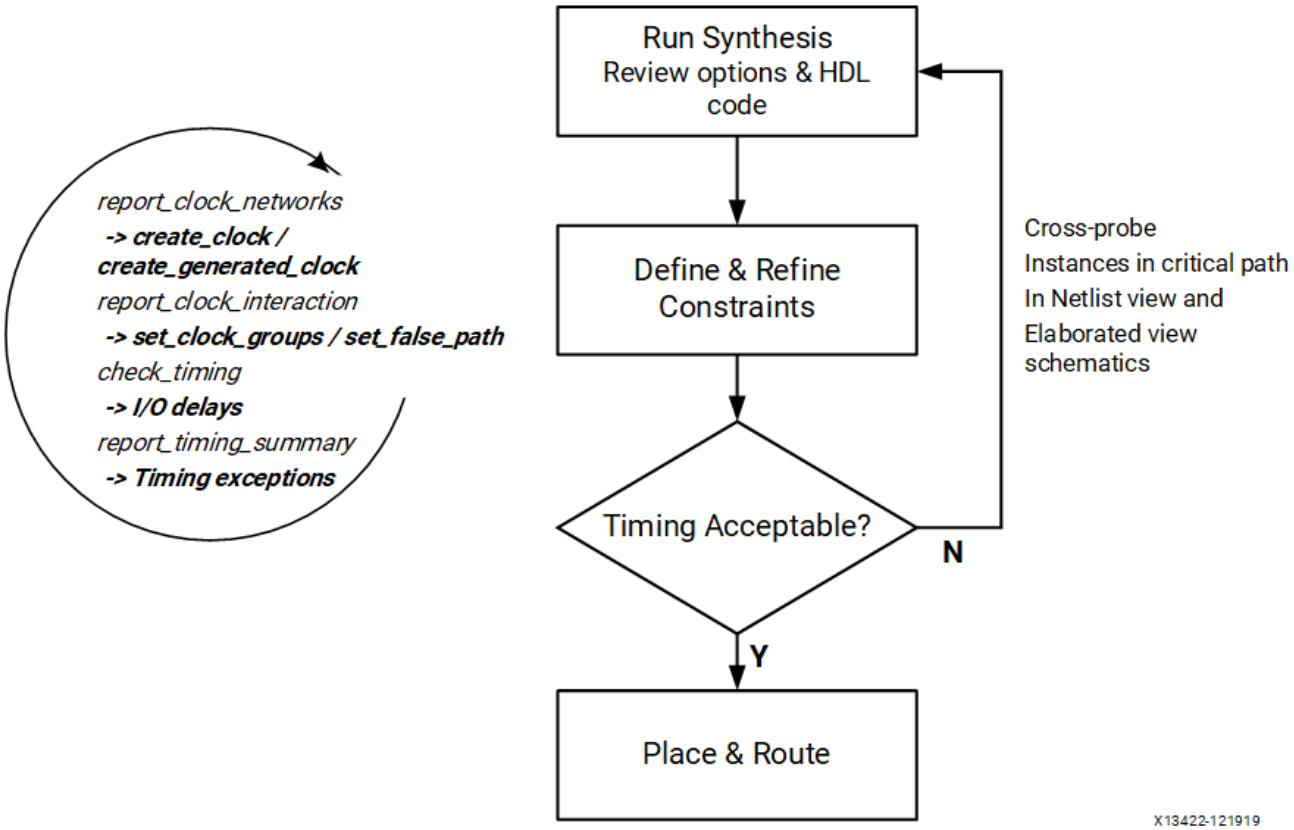
★ **Tip:** For more information on reports mentioned in this chapter, see *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906).

★ **Tip:** See the *UltraFast Design Methodology Timing Closure Quick Reference Guide* (UG1292) for a condensed version of the techniques described in this chapter, including running initial design checks, baselining the design, and resolving timing violations.

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: 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 worst negative slack (WNS) of each clock as the main way to improve total negative slack (TNS).
- Review large worst hold slack (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 Xilinx® design constraints (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 design rule checks (DRCs) and baselining, identifying the timing violation root causes, and addressing the violations using common techniques.

✎ **Note:** Timing results after synthesis use estimated net delays and not the actual routing delays. To get the final timing results, run implementation and then check the Report Timing Summary.

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

✎ **Power Tip:** When you have a design run with clean timing, consider using the Create Runs command in the Vivado® IDE to run multiple strategies. Run the `report_power` command on each design with accurate switching activity XDC constraints to find the best run from both a timing and power perspective.

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. For more information on baselining the design, see Baselining the Design.

Related Information

Baselining the Design

Checking for Positive Timing Slacks

The following timing metrics reflect the design timing score. Numbers must be positive to meet timing.

- Setup/Recovery (max delay analysis): WNS > 0 ns and TNS = 0 ns
- Hold/Removal (min delay analysis): WHS > 0 ns and THS = 0 ns
- Pulse Width: WPWS > 0 ns and TPWS = 0 ns

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:

Total Negative Slack (TNS)

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 worst negative slack (WNS).

Total Hold Slack (THS)

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 worst hold slack (WHS).

Total Pulse Width Slack (TPWS)

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)

Worst Pulse Width Slack (WPWS)

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 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).

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. To identify and correct unrealistic target clock frequencies or setup path requirement, use the `report_qor_assessment` command.

!! Important: To address missing or incomplete constraints, use the Timing Constraints wizard or see the *Vivado Design Suite User Guide: Using Constraints* (UG903).

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.

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 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.
For more information on some of these checks, see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906). Also, see the Adoption Of The Methodology Report blog series for more information on how `report_methodology` helps to resolve issues and save time.

!! Important: To increase visibility, the summary of the methodology violations is also included in the timing summary text report, because addressing these issues is critical for having proper signoff timing.

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.

!! Important: Carefully verify timing checks with a severity of Critical Warning.

For more information on timing methodology checks, see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906).

Table: Timing Closure Methodology DRCs

Check	Severity	Description
TIMING-6	Critical Warning	No common clock between related clocks
TIMING-7	Critical Warning	No common node between related clocks
TIMING-8	Critical Warning	No common period between related clocks
TIMING-14	Critical Warning	LUT on the clock tree
TIMING-15	Warning	Large hold violation on inter-clock path
TIMING-16	Warning	Large setup violation
TIMING-30	Warning	Sub-optimal master source pin selection for generated clock
TIMING-31	Critical Warning	Inappropriate multicycle path between phase shifted clocks

Check	Severity	Description
TIMING-32, TIMING-33, TIMING-34, TIMING-37, TIMING-38, TIMING-39	Warning	Non-recommended bus skew constraint
TIMING-36	Critical Warning	Missing master clock edge propagation for generated clock
TIMING-42	Warning	Clock propagation prevented by path segmentation
TIMING-44 TIMING-45	Warning	Unreasonable user intra and inter-clock uncertainty
TIMING-48	Advisory	Max Delay Datapath Only constraint on latch input
TIMING-49	Critical Warning	Unsafe enable or reset topology from parallel BUFGCE_DIV
TIMING-50	Warning	Unrealistic path requirement between same-level latches
XDCB-3	Warning	Same clock mentioned in multiple groups in the same `set_clock_groups` command
XDCH-1	Warning	Hold option missing in multicycle path constraint
XDCV-1	Warning	Incomplete constraint coverage due to missing original object used in replication
XDCV-2	Warning	Incomplete constraint coverage due to missing replicated objects

Methodology DRCs with Impact on Signoff Quality and Hardware Stability

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.

!! Important: Carefully verify timing checks with a severity of Critical Warning.

Table: Signoff Quality Methodology DRCs

Check	Severity	Description
TIMING-1, TIMING-2, TIMING-3, TIMING-4, TIMING-27	Critical Warning	Non-recommended clock source point definition
TIMING-5, TIMING-25, TIMING-19	Critical Warning	Unexpected clock waveform
TIMING-9, TIMING-10	Warning	Unknown or incomplete CDC circuitry
TIMING-11	Warning	Inappropriate `set_max_delay -datapath_only` command
TIMING-12	Warning	Clock Reconvergence Pessimism Removal disabled
TIMING-13, TIMING-23	Warning	Incomplete timing analysis due to broken paths
TIMING-17	Critical Warning	Non-clocked sequential cell
TIMING-18, TIMING-20, TIMING-26	Warning	Missing clock or input/output delay constraints
TIMING-21, TIMING-22	Warning	Issues with MMCM compensation
TIMING-24	Warning	Overridden `set_max_delay -datapath_only` command
TIMING-29	Warning	Inconsistent pair of multicycle paths

Check	Severity	Description
TIMING-35	Critical Warning	No common node in paths with the same clock
TIMING-40, TIMING-43	Warning	Inappropriate clock topologies or requirements
TIMING-41	Warning	Invalid forwarded clock defined on an internal pin
TIMING-46	Warning	Multicycle path with tied CE pins
TIMING-47	Warning	False path or asynchronous clock group between synchronous clocks
TIMING-51	Critical Warning	No common phase between related clocks from parallel MMCMs or PLLs
TIMING-52	Critical Warning	No common phase between related clocks from Spread Spectrum MMCM

Other Timing Methodology DRCs

Other TIMING and XDC checks identify constraints that can incur higher run time, 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.

Assessing the Maximum Frequency of the Design

You can define and assess the maximum frequency (FMAX) with a design that runs on a given architecture and speed grade by iteratively increasing the target clock frequency and re-running both synthesis and implementation until small setup slack violations (WNS < 0) are reported by timing analysis on the fully routed design. Xilinx recommends using the Default or PerformanceOptimized synthesis directives along with the Explore implementation directives and strategy to get the best achievable FMAX. In some cases, alternate strategies can show higher FMAX depending on the size of the design and the nature of the critical logic paths. For the implementation results with small setup violations, the maximum frequency is computed as follows:

- FMAX (MHz) = max(1000/(Ti - WNSi))

Where:

- Ti is the target clock period (ns) used during the implementation run "i"
- WNSi is the worst negative slack (ns) of the target clock used during the implementation run "i"

Additional important considerations:

- Using overly tight clock periods can lead to automatic effort reduction in the Vivado Implementation tools to avoid high compilation time due to unrealistic target and large timing violations. Use reasonably tight clock constraints instead.
- For designs with multiple clocks, you must proportionally decrease all synchronous clock periods until one of them starts failing timing after implementation (preferably the fastest clock or the clock with the most timing paths).

✎ **Note:** The FMAX value is not explicitly provided in the `report_timing` or `report_timing_summary` report.

For a given design implementation, the maximum operating frequency on hardware across temperature and voltage ranges supported by the target device speed grade is defined by 1000/(T - WNS), with WNS positive or negative. When operating under nominal temperature and voltage conditions, typically in a lab environment, it is usually possible to operate the design at a slightly higher frequency.

✎ **Note:** To increase the maximum frequency of the design, you can leverage the techniques described in this chapter or use Intelligent Design Runs.

Related Information

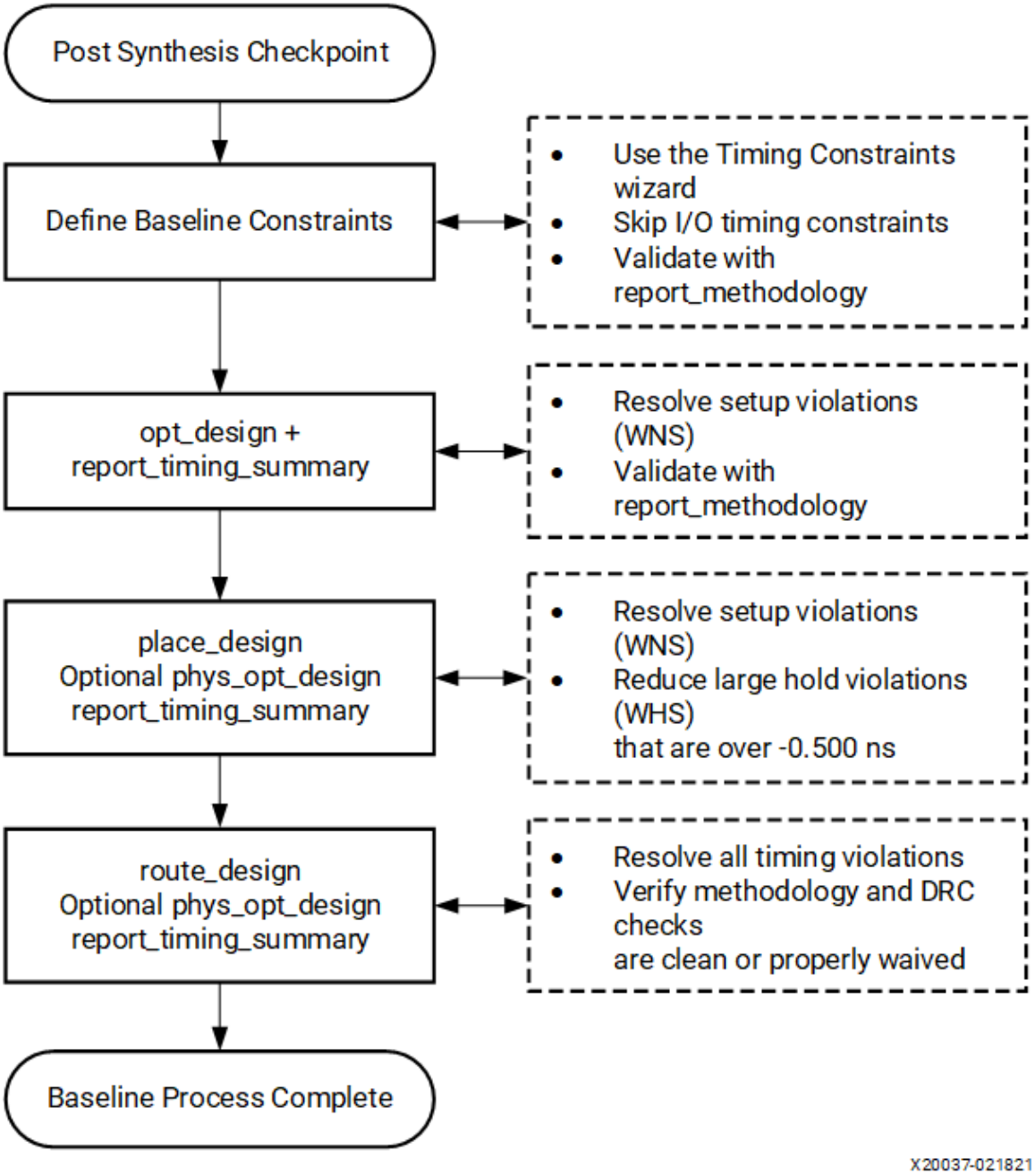
Using Intelligent Design Runs

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: 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) constraints
  CDC paths between synchronous clocks are safely timed by default, but you must use safe CDC circuitry and specify timing exceptions between asynchronous clocks.

After creating the constraints, identify the paths that cannot meet timing. Rewrite the corresponding RTL or relax the clock period.

!! Important: All Xilinx IP and 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 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).

Figure: Clock Report Shows the Clocks Generated from Primary Clocks

```
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}
```

★ **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.

Figure: 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

The following table explains the meaning of each color in this report.

Table: 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.

Color	Label	Meaning	What Next
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.

Color	Label	Meaning	What Next
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). Also, see this link in the *Vivado Design Suite Tcl Command Reference Guide* (UG835).

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. 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.

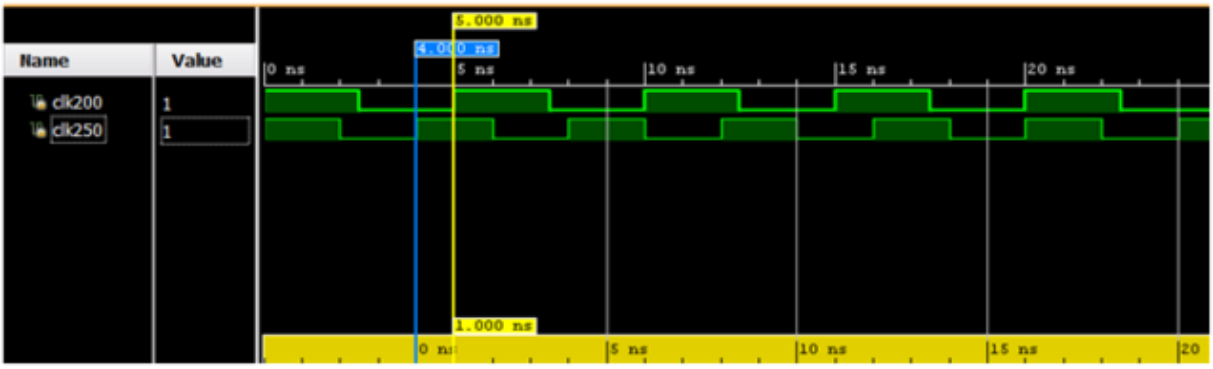
✎ **Note:** The simultaneous edge at 20 ns is not the tightest requirement in this example, because the capture edge cannot be the same as the launch edge.

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.

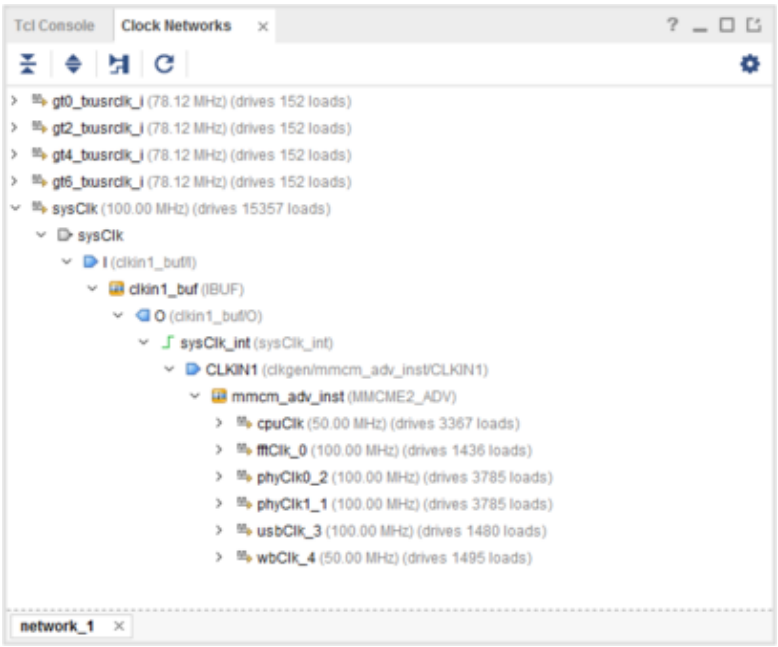
Figure: Clock Domain Crossing from 250 MHz to 200 MHz



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: 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:** 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 After Each Step

You must evaluate the design WNS after each synthesis and 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 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 -0.500 ns 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
- Digital signal processor (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.

Table: Timing Summary Report for Synthesized Design

	WNS	TNS	Num Failing Endpoints	WHS	THS	Num Failing Endpo
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` and `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 Critical 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.

Table: Timing Summary Report

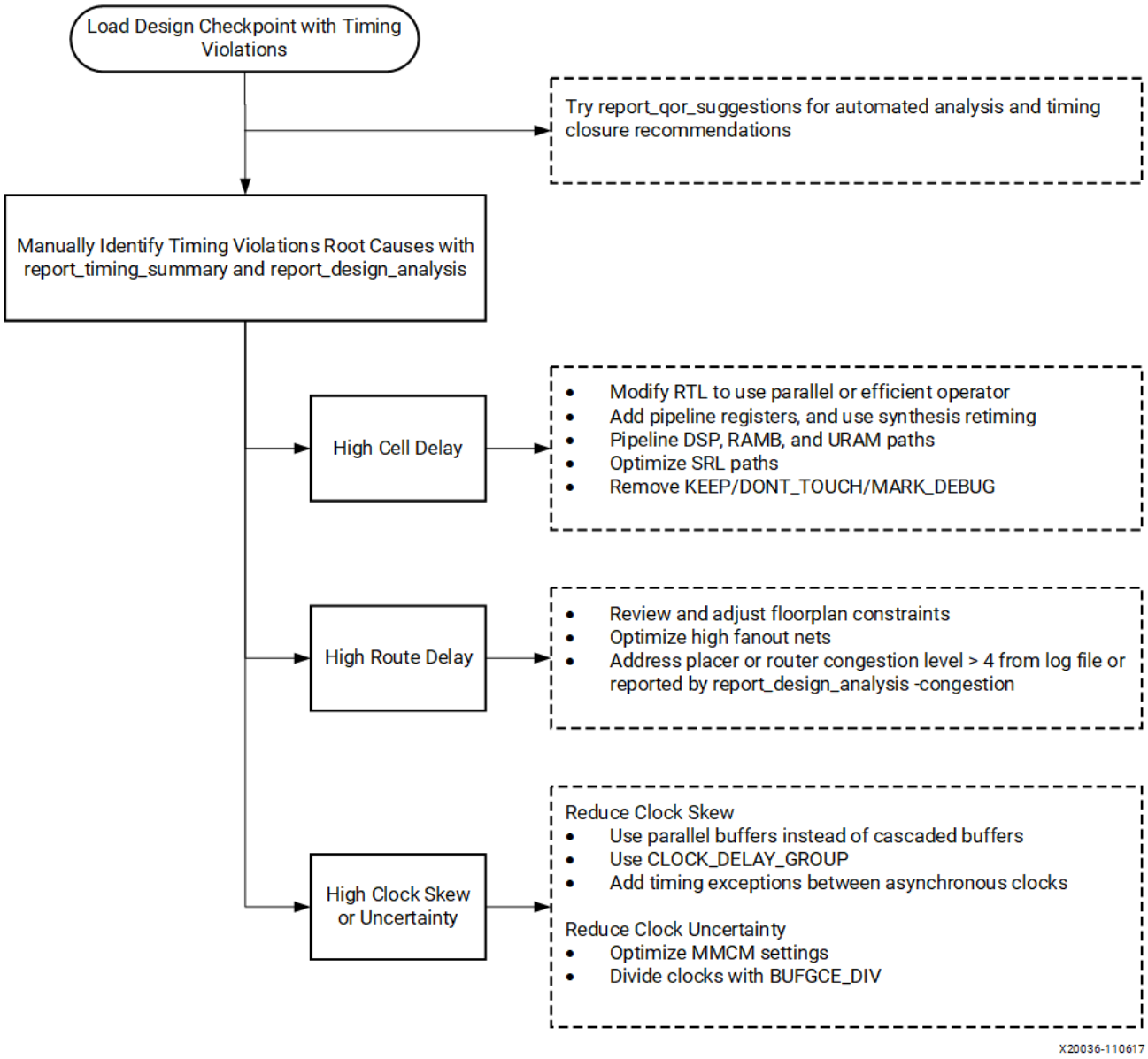
	WNS	TNS	Num Failing Endpoints	WHS	THS	Num Failing Endpo
Opt						
Place						
Physopt						
Route						

12. 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. When the worst violations are fixed, the tools typically resolve many of the less critical paths automatically when you rerun the implementation tools. You can assist in this process by focusing on resolutions that positively impact a high number of paths. For example, correcting suboptimal clocking typically impacts a high number of paths so Xilinx recommends focusing on these issues first before moving to path specific resolutions. The Report QoR Suggestions command automatically identifies issues and orders suggestions based on criticality. You can determine the progress made towards timing closure by running the Report QoR Assessment command both before and after applying the suggestions. An increase in the QoR Assessment Score and a decrease in the detailed table marked for review indicates improvements.
The following figure shows the basic process for analyzing and resolving timing violations.

Figure: 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 0.5 ns.
- 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

- datapath delay (max)
+ clock skew
- clock uncertainty
- setup/recovery time

Slack (hold/removal) = hold path requirement

+ datapath delay (min)
- clock skew
- clock uncertainty
- 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 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).

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.

★ **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.

Related Information

Checking That Your Design is Properly Constrained

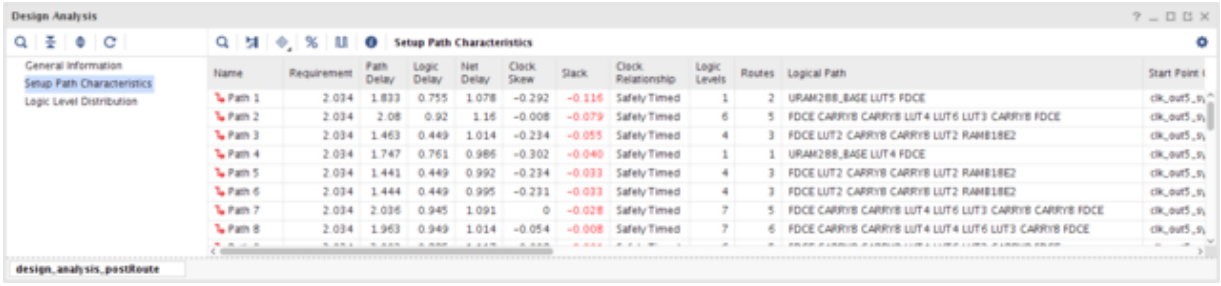
Analyze Path Characteristics

To report the 50 worst setup timing paths, you can use the Report Design Analysis dialog box in the Vivado IDE, or you can use the following command:

```
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.

Figure: Report Design Analysis Timing Path Characteristics Post-Route



Following are tips for working with this table:

- Toggle between numbers and % by clicking 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 the Hide Unused button to turn off filtering and show all columns, or right-click 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)

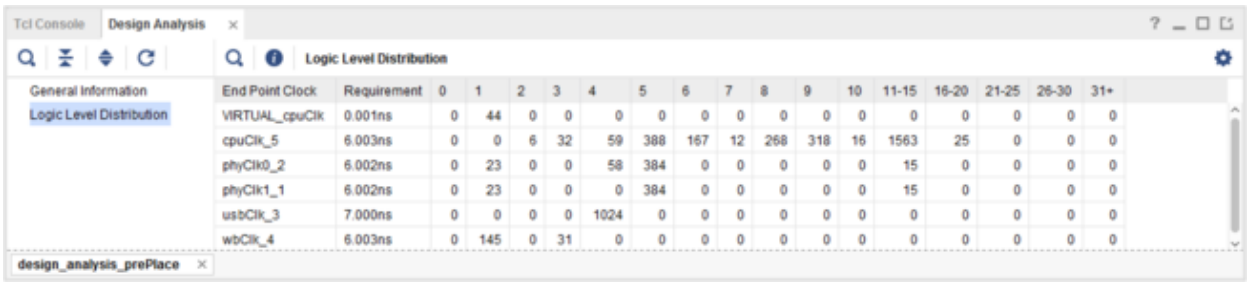
★ **Tip:** For visualizing the details of the timing paths in the Vivado IDE, select the path in the table, and go to the Properties tab.

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 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 timing 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
```

Figure: Report Design Analysis Timing Path Characteristics Pre-Place



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

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 different devices, separate cell delay and route delay ranges must be considered.
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 Reducing Net Delay Caused by 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`, or `set_max_delay -datapath_only`).
If the clock uncertainty is over 100 ps, then you must review the clock topology and jitter numbers to understand why the uncertainty is so high.

Related Information

Reducing Clock Skew
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 timing and power 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.

Related Information

Identifying Timing Violations Root Cause

Optimizing Regular Fabric Paths

Regular fabric paths are paths between fabric registers or shift registers that traverse a mix of resources, such as LUTs. 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, the presence of KEEP_HIERARCHY, DONT_TOUCH, or MARK_DEBUG attributes, or intermediate signals with some fanout (10 and higher). Run the `opt_design -remap` option or use the AddRemap or ExploreWithRemap directives to collapse smaller LUTs and reduce the number of logic levels. If `opt_design` is unable to optimize the longest paths due to a net fanout greater than one between the small LUTs, you can force the optimization by setting the LUT_REMAP property on the LUTs.
- Single CARRY cell is present in the path
  CARRY primitives are most beneficial for timing 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. Alternatively, use the CARRY_REMAP property to instruct `opt_design` to remap the tagged CARRY cells to LUTs.

  ✎ **Note:** This optimization technique is automatically applied by the `report_qor_suggestions` Tcl command.
- Path ends at shift register (SRL)
  Pull the first register out of the shift register by using the SRL_STYLE attribute in RTL. For details, see this link in the *Vivado Design Suite User Guide: Synthesis* (UG901). Alternatively, you can use the SRL_STAGES_TO_REG_INPUT property applied prior to `opt_design` to implement the same optimization. For details, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904).

  ✎ **Note:** This optimization technique is automatically applied by the `report_qor_suggestions` Tcl command.
- 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. Alternatively, you can instruct `opt_design` to perform the same optimization by setting the CONTROL_SET_REMAP property on the registers to optimize.

  ✎ **Note:** This optimization technique is automatically applied by the `report_qor_suggestions` Tcl command.

★ **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).

Related Information

Pushing the Logic from the Control Pin to the Data Pin

Optimizing Paths with Dedicated Blocks and Macro Primitives

Paths from/to/between dedicated blocks and macro primitives (e.g., DSP, block RAM, or UltraRAM) need special attention because 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 series 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 the following:

- Pipeline paths from and to dedicated blocks and macro primitives as much as possible.
- Restructure 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.
- Meet setup timing by at least 500 ps on these paths before placement.
- Replicate cones of logic connected to too many dedicated blocks or macro primitives if they need to be placed far apart.
- When the design has tight timing requirements to, within, or from a DSP block, run `opt_design -dsp_register_opt` to move registers to a more timing optimal position.

  ✎ **Note:** Because timing is approximate during `opt_design`, you might also need to run `phys_opt_design -dsp_register_opt` to correct movements where timing was not accurately represented at the pre-placement stage.

Reducing Net Delay Caused by 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 when the design is ready for timing closure, physical constraints such as 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.

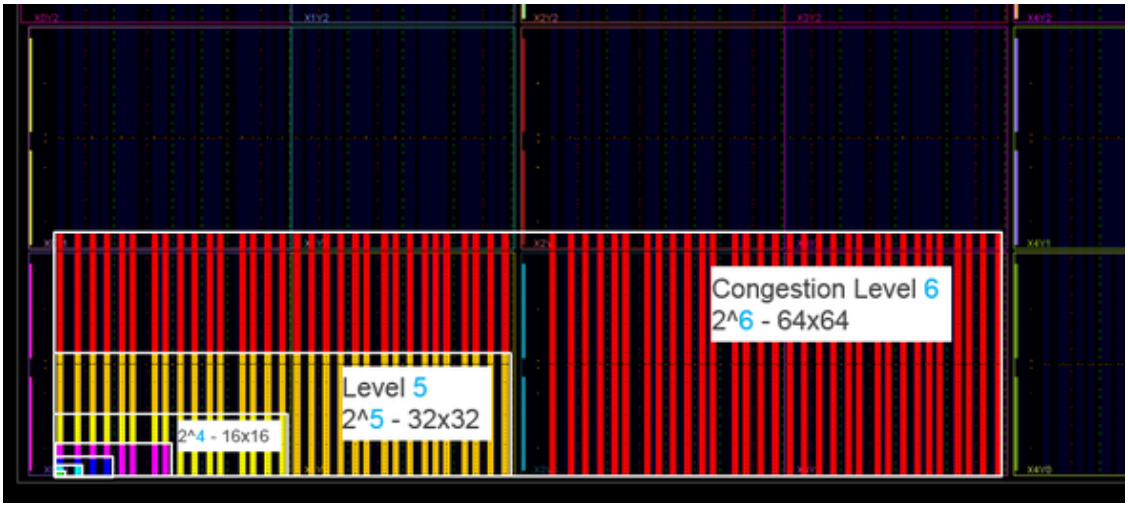
Reducing Net Delay Caused by Congestion

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, analyze the congestion of the design and identify the best congestion alleviation technique.

Congestion Area and Level Definition

Xilinx device 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 a Xilinx device versus clock regions.

Figure: Congestion Levels and Areas in the 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: Congestion Level Ranges

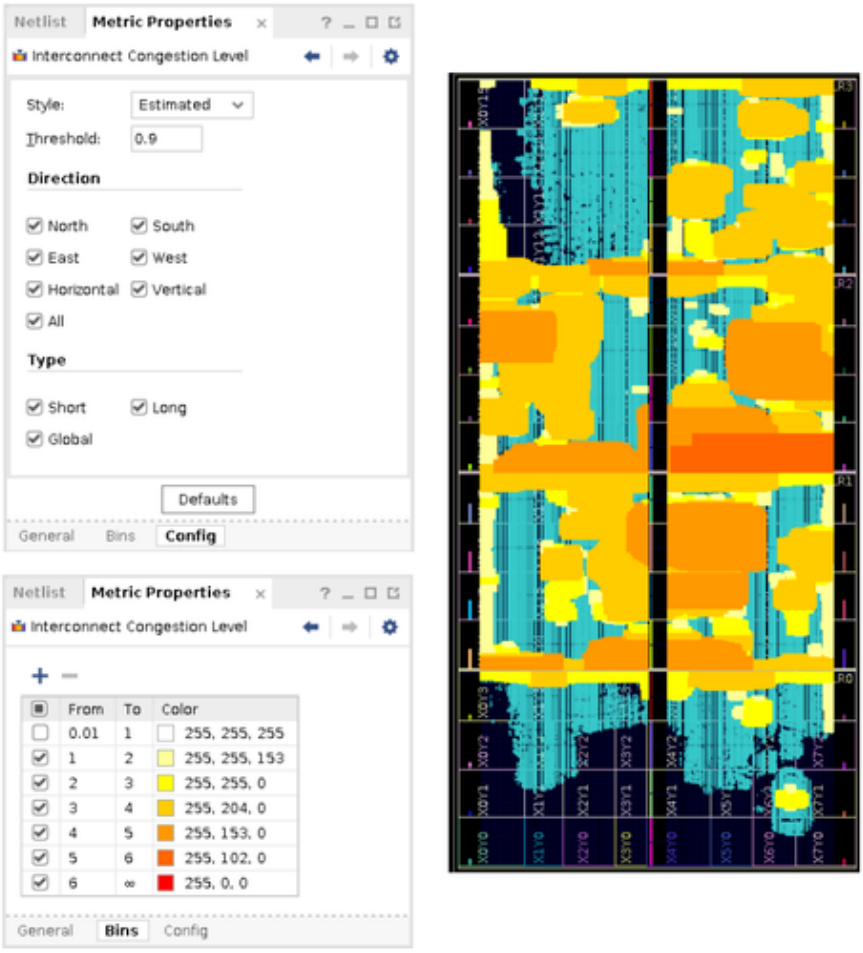
Level	Area	Congestion	QoR Impact
1, 2	2x2, 4x4	None	None
3, 4	8x8, 16x16	Mild	Possible QoR degradation
5	32x32	Moderate	Likely QoR degradation
6	64x64	High	Difficulty routing
7, 8	128x128, 256x256	Impossible	Likely unroutable

Interconnect Congestion Level in the Device Window

The Interconnect Congestion Level metric highlights the largest contiguous area in which routing resources are overused. By default, this metric is based on estimation, which is similar to the congestion level after initial routing. Actual routing can also be displayed if routing exists. After placement or after routing, you can display this congestion metric by right-clicking in the Device window and selecting Metric > Interconnect Congestion Level. The Interconnect Congestion Level metric provides a quick visual overview of any congestion hotspots in the device. The following figure shows a placed design with several congested areas. This metric is based on the current interconnect demand and availability with a threshold of 0.9 (that is, 90% routing usage). The range is 0.1 to 0.9. You can visualize congestion based on:

- Direction: North, South, East, West, Vertical, Horizontal
- Type: Short, Long, Global
- Style: Estimated, Routed, Mixed

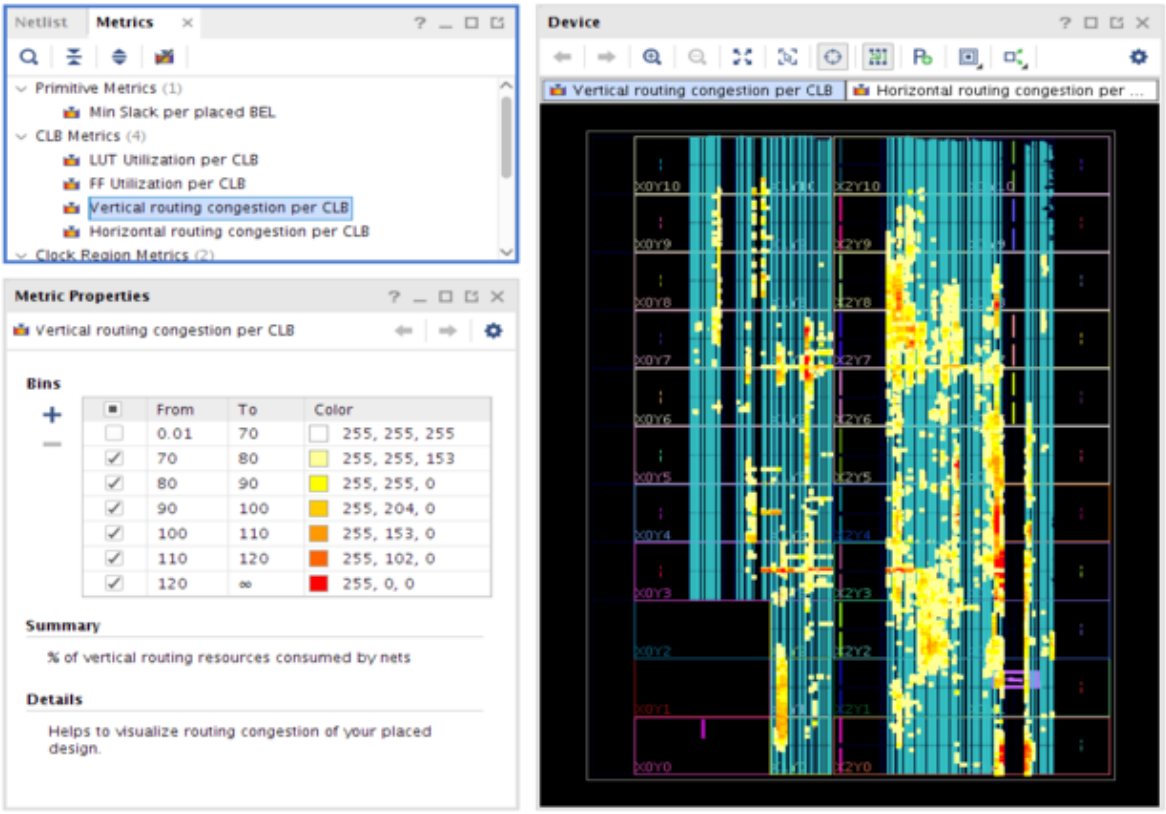
Figure: Example of Interconnect Congestion Level in the Device Window



Use the Routing Congestion per CLB, which 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 and selecting Metric > 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.

Note: Use this method for 7 series and UltraScale devices only.

Figure: Example of Congestion per CLB 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 timing 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).

Figure: Initial Estimated Congestion Table

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

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. An Info message is issued and the congested CLBs and nets are written to the text file listed in the message body. You can examine the text file for the list of CLB tiles and congested nets that are involved in the CLB pin-feed congestion, and use the congestion alleviation techniques listed in the Addressing Congestion section to resolve the CLB congestion before routing the design.

```
INFO: [Route 35-443] CLB routing congestion detected. Several
CLBs have high routing utilization, which can impact timing
closure. Congested CLBs and Nets are dumped in:
iter_200_CongestedCLBsAndNets.txt
```

★ **Tip:** Localized CLB routing congestion can lead to routing failures even when the reported congestion levels for Global, Long, or Short congestion are within the acceptable range (less than 5). Look for the message above and in generated text files for localized congestion hotspots.

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: 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.
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).

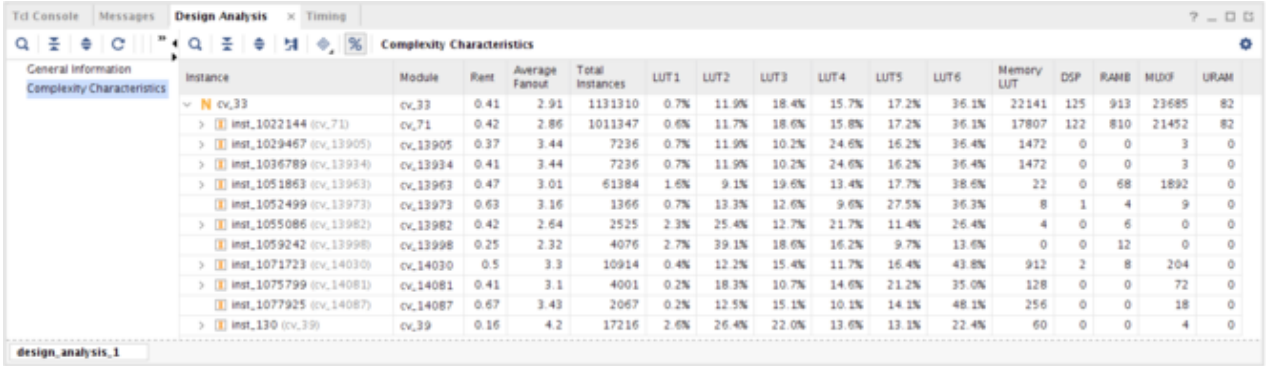
Report Design Analysis Complexity Report

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.

Figure: Complexity Report



The following table shows the typical ranges for the Rent Exponent.

Table: 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.

Range	Meaning
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: 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. 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).

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 the 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 resource columns, 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.
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.

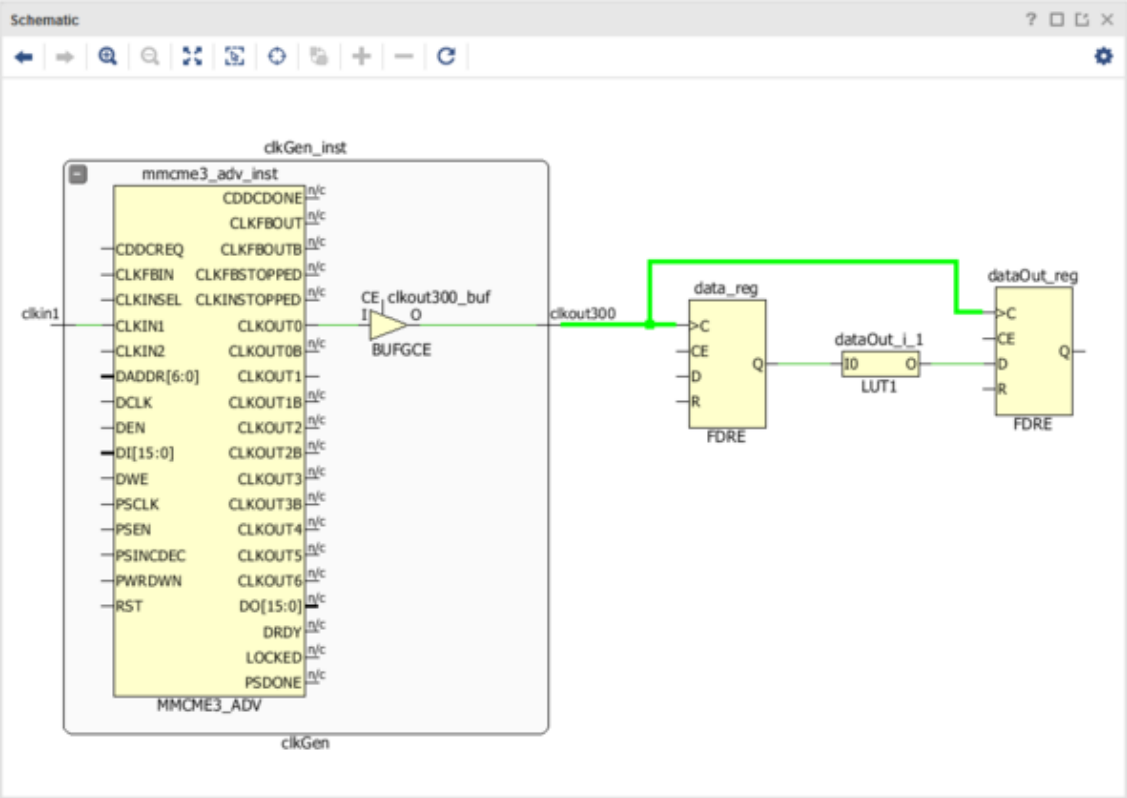
Related Information

Defining Clock Groups and CDC Constraints

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: 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.

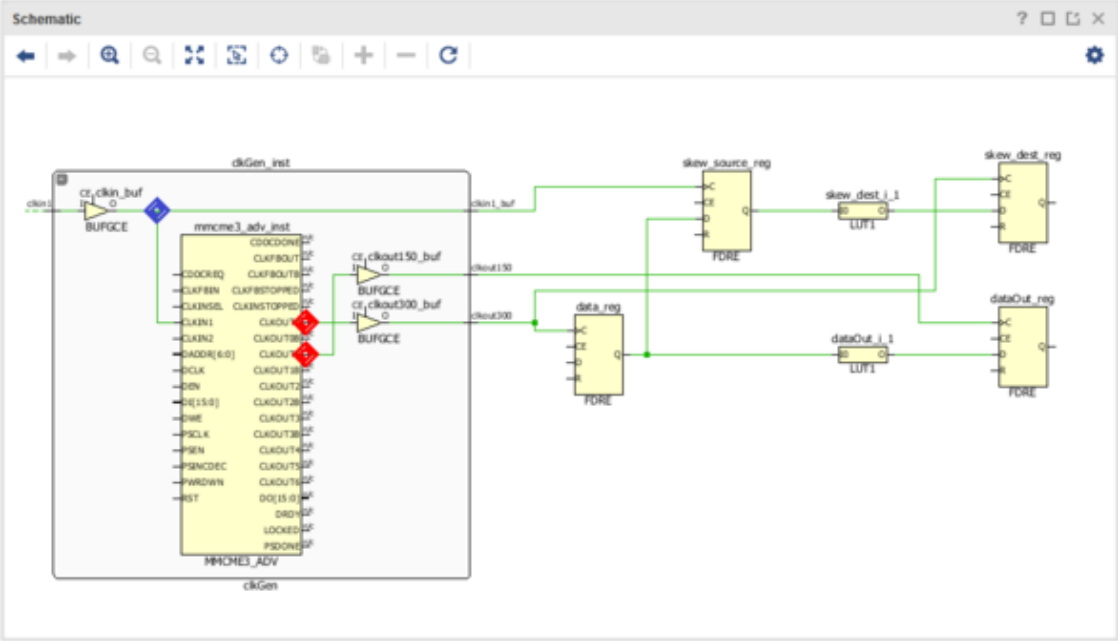
Limiting Synchronous Clock Domain Crossing Paths

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: 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.

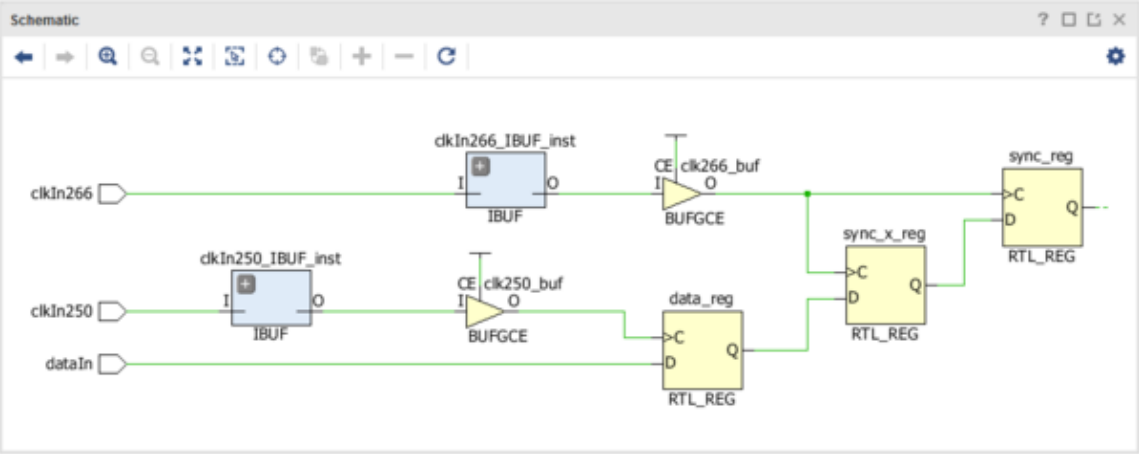
Adding Timing Exceptions between Asynchronous Clocks

Timing paths in which the source and destination clocks originate from different primary clocks or have no common node, no common phase, or no common period 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.

Figure: Asynchronous CDC Paths with Proper CDC Circuitry and No Common Node



Related Information

Defining Clock Groups and CDC Constraints

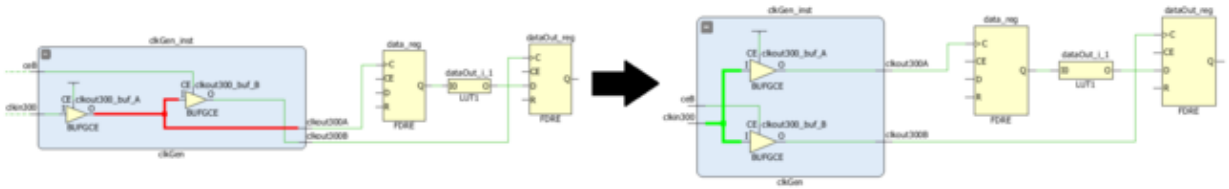
Applying Common Techniques for Reducing Clock Skew

★ **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:

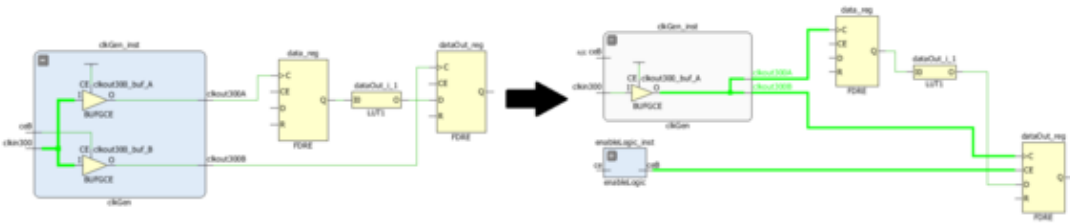
- Avoid timing paths between cascaded clock buffers by eliminating unnecessary buffers or connecting them in parallel as shown in the following figure.

Figure: Synchronous Clocking Topology with Cascaded BUFG Reconnected in Parallel



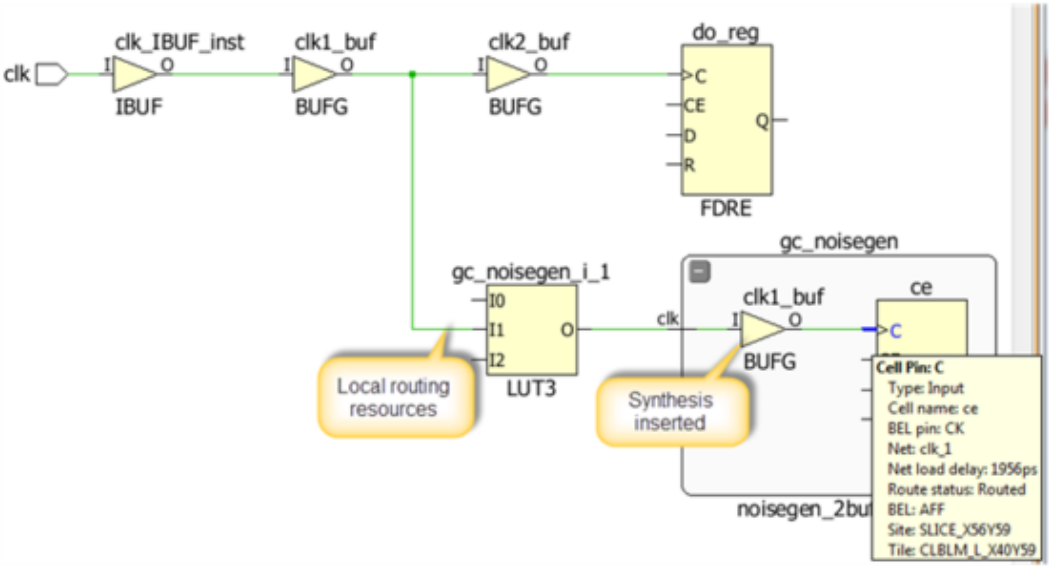
- Combine parallel clock buffers into a single clock buffer and connect any clock buffer clock enable logic to the corresponding sequential cell enable pins, as shown on figure below. If some of the clocks are divided by the buffer's built-in divider, implement the equivalent division with clock enable 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: 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: Skew Due to Local Routing on Clock Network



!! Important: The 7 series and UltraScale device clocking architectures differ. You must follow the clocking guidelines for your targeted architecture and verify that your design complies.

Related Information

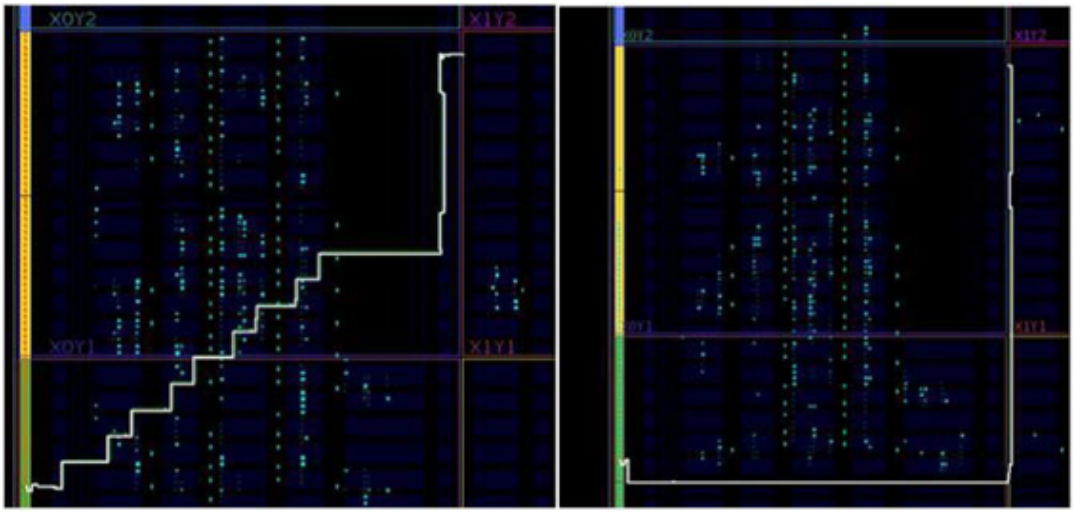
Clocking Guidelines

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: Comparison of Fabric Clock Route versus Dedicated Clock Route

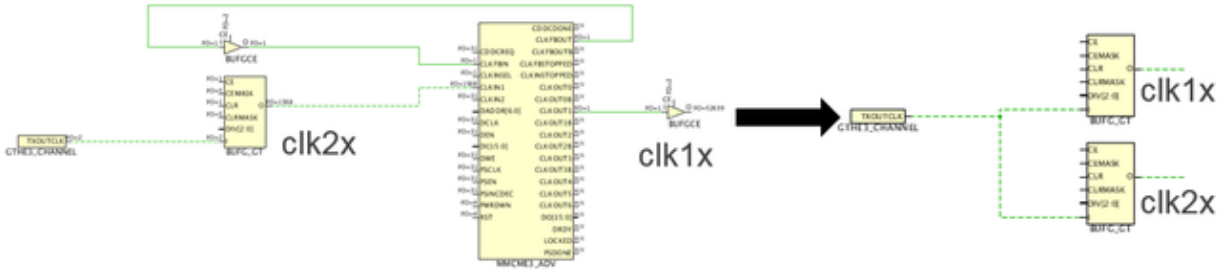


- Do not allow regional clock buffers (BUFR/BUFIO/BUFH) to drive logic in several clock regions as the skew between the clock tree branches in each region will be very high. Remove inappropriate LOC or Pblock constraints to resolve this situation.

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: 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.

  ✎ **Note:** This optimization technique is automatically applied by the `report_qor_suggestions` Tcl command.

- 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, constraining the location of 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. 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.

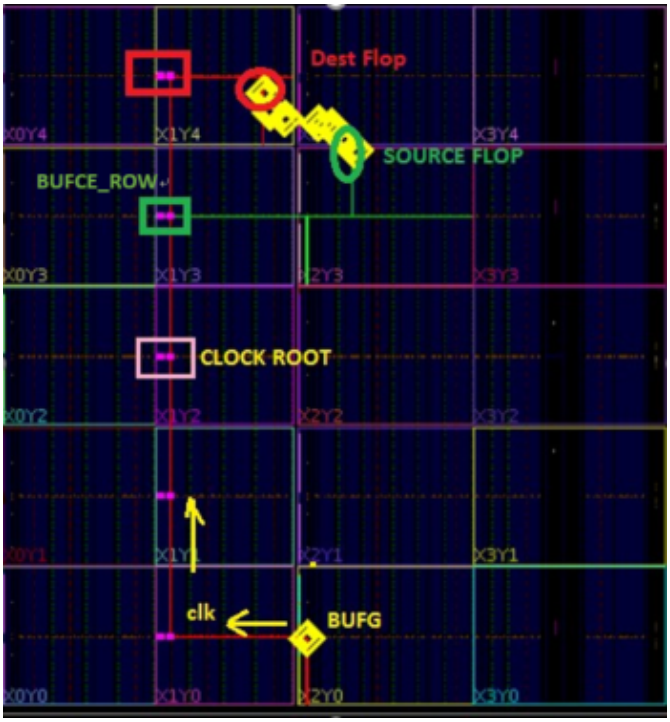
Related Information

Synchronous CDC
Clock Constraints

Reducing Clock Delay in UltraScale and UltraScale+ Devices

In UltraScale and UltraScale+™ global clock routing, the clock net is first routed from a global clock buffer via the horizontal and vertical routing track to a central location called the clock root. From the clock root, the clock net spans out to drive clock rows in each clock region via the vertical distribution track. On each row, there are programmable delays in the clock network on BUFCE_ROW route-through sites that perform a coarse-grained deskew as the clock spans farther from the clock root.
The following figure shows a clock path from the global clock buffer (BUFG) to the clock root. The clock routing switches from routing to the vertical distribution track, through the BUFCE_ROW in each clock region row that drives the horizontal distribution tracks, and then to the leaf level. The source is shown in green and the destination in red.

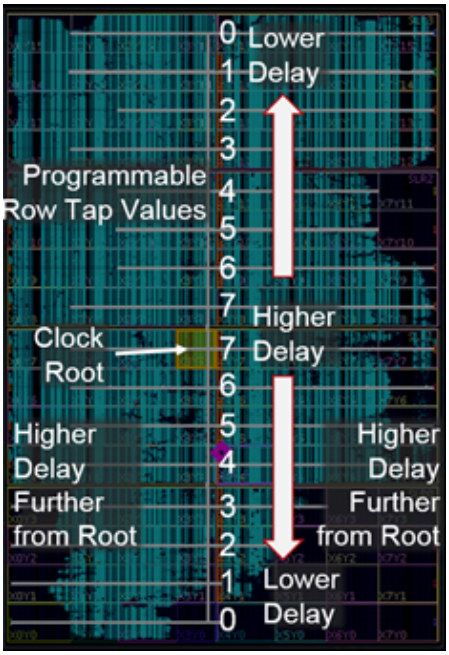
Figure: Clock Path from BUFG to the Leaf Level via BUFCE_ROW



The row programmable tap delay is the largest near the clock root. This delay decreases by one tap for one clock region as the clock reaches farther away from the root in the vertical direction, eventually decreasing to zero.
The following figure shows the topology of the programmable row tap values decreasing from the root. Higher tap values mean higher delays and higher crossing SLR clock skew, because the higher tap values add additional uncertainty for timing due to the minimum/maximum delay variation introduced by the manufacturing process variation.

This makes it more difficult to meet timing near the root where programmable tap delay values are higher. Farther from the root in the vertical direction, there is less uncertainty, and it is generally easier to fix hold violations on crossing SLR buses. For SLR crossing buses that are farther from the root in the horizontal direction, the clock row delays increase. This additional delay introduces more minimum/maximum delay variation and reduces the performance of SLR crossings.

Figure: Row Programmable Tap Delay Settings Across an UltraScale+ SSI Technology Device



For UltraScale+ SSI technology devices, you can improve SLR crossing speed using either of the following methods:

- Move the clock root close to the SLR crossings in the horizontal direction
- Limit the maximum row programmable tap delay value to reduce the uncertainty

---

✎ **Note:** Timing paths farther from the root in the vertical direction might become slightly slower due to increased delay from hold fixing route detours. However, using these methods results in an overall performance gain.

You can review the row programmable tap delay settings that the Vivado tool chose for each global clock in your design in the Device Cell Placement Summary for Global Clock sections in the Clock Utilization Report. Following is an example that shows the row programmable tap delay settings for the g13 global clock in the HORIZONTAL PROG DELAY column, which is highlighted in yellow.

Figure: Global Clock Row Programmable Tap Delay Settings in the Clock Utilization Report

22. Device Cell Placement Summary for Global Clock g13

Global Id	Driver Type/Pin	Driver Region (D)	Clock	Period (ns)	Waveform (ns)	Root (R)	Slice Loads
g13	BUFGCE/O	X4Y10	Multiple	4.926	{0.000 2.463}	X3Y8	12511

* Slice Loads column represents load cell count of all cell types other than IO, GT and clock resources
** IO Loads column represents load cell count of IO types
*** Clocking Loads column represents load cell count that are clock resources (global clock buffer, MMCM, PLL, etc)
**** GT Loads column represents load cell count of GT types

	X0	X1	X2	X3	X4	X5	X6	X7	HORIZONTAL PROG DELAY
Y15	2820	4086	308	0	0	0	0	0	0
Y14	713	392	0	0	3	0	0	0	0
Y13	0	0	0	0	0	0	0	0	0
Y12	0	0	0	0	0	0	0	0	0
Y11	0	0	0	0	0	62	94	3	3
Y10	0	0	801	3	(D) 45	224	216	0	4
Y9	0	0	252	91	361	185	10	0	5
Y8	0	0	66	(R) 261	241	2	0	0	5
Y7	0	17	365	117	16	0	0	0	4
Y6	0	165	275	39	2	0	0	0	3
Y5	0	120	66	0	0	0	0	0	2
Y4	0	27	7	0	0	0	0	0	1
Y3	21	21	0	3	0	0	0	0	0
Y2	11	1	0	0	0	0	0	0	0
Y1	0	0	0	0	0	0	0	0	0
Y0	0	0	0	0	0	0	0	0	0

For UltraScale+ SSI technology devices, the placer limits the maximum row programmable tap delay value to reduce minimum/maximum delay variation and reduce SLR crossing clock skew near the clock root, while also ensuring that clock regions on either side of SLR crossings have an increasing or decreasing tap delay value to balance the clock skew on SLR crossing paths farther from the root. The MAX_PROG_DELAY property value of the clock net can be queried to find the maximum row programmable tap delay value used by the placer.
You can also limit the row programmable tap value using the USER_MAX_PROG_DELAY property. Following is an example. To set the USER_MAX_PROG_DELAY property, the value must be applied to the net segment directly driven by the global clock buffer. If the USER_MAX_PROG_DELAY property is not set, the placer can use the maximum possible tap setting of 7.

```
set_property USER_MAX_PROG_DELAY <0-7> [get_nets -of [get_pins
BUFG/O]]
```

Following are tips when using the USER_MAX_PROG_DELAY property:

- The recommended USER_MAX_PROG_DELAY tap value is 3 or 4 for clocks that span the majority of UltraScale+ SSI technology devices. When clock roots are near GT, PCIe® , or CMAC blocks that are off-center in the device, SLR crossing performance on the opposite device side is heavily impacted, because the common node for the launch and capture clock is farther away from the SLR crossing.
- For clock groups using the CLOCK_DELAY_GROUP for clock network matching, ensure that all clocks within the clock group use the same USER_MAX_PROG_DELAY value.

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

✎ **Note:** The `report_qor_suggestions` Tcl command flags this issue.

When configuring an MMCM for frequency synthesis, Xilinx recommends configuring the MMCM to achieve the lowest output jitter on the clocks. Optimize the MMCM settings to run at the highest possible voltage-controlled oscillator (VCO) frequency that meets the allowed operating range for the device. The following equations show the relationship between VCO frequency, M (multiplier), D (divider), and O (output divider) settings to both the input and output clock frequencies:

$$F_{VCO} = F_{CLKIN} \times \frac{M}{D}$$

$$F_{OUT} = F_{CLKIN} \times \frac{M}{D \times O}$$

★ **Tip:** You can increase the VCO frequency by increasing M, lowering D, or both and compensating for the change in frequency by increasing O. Increases in VCO frequency negatively affects the power dissipation from the MMCM or PLL. You can also make small increases in the VCO frequency when you switch from multiple MMCM clock

outputs using BUFGs to one MMCM clock output using BUFGCE_DIVs, which allows more clocks to use the fractional divider. When selecting between MMCM and PLL, MMCMs are preferred because they are able to operate at a higher VCO frequency, have improved granularity for selecting M and D values, and have fractional dividers (CLKOUT0).

Different architectures have different VCO frequency maximums. Therefore, Xilinx recommends regenerating clocking components to be optimal for your target architecture. Xilinx recommends using the Clocking Wizard to automatically calculate M and D values along with the VCO frequency to properly configure an MMCM for the target device.

★ **Tip:** When using the Clocking Wizard from the IP catalog, make sure that Jitter Optimization Setting is set to the Minimize Output Jitter, which provides the higher VCO frequency. In addition, performing marginal changes to the desired output clock frequency can allow for an even higher VCO frequency to further reduce clock uncertainty.

The following MMCM frequency synthesis example 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: MMCM Frequency Synthesis Example

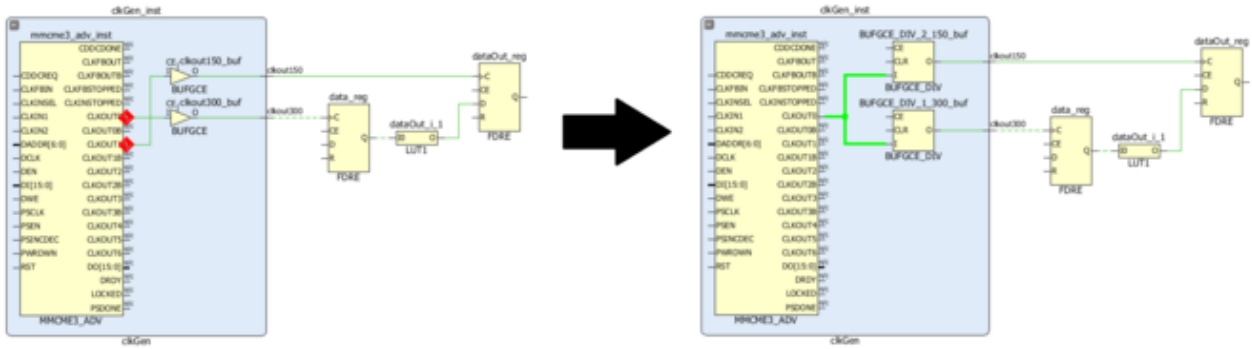
	MMCM_1	MMCM_2
Input clock	62.5 MHz	62.5 MHz
Output clock	40.0 MHz	39.991 MHz
CLKFBOUT_MULT_F(M)	16	22.875
DIVCLK_DIVIDE(D)	1	1
VCO Frequency	1000.000 MHz	1429.688
CLKOUT0_DIVIDE_F(O)	25	35.750
Jitter (ps)	167.542	128.632
Phase Error (ps)	384.432	123.641

Using BUFGCE_DIV to Reduce Clock Uncertainty

★ **Tip:** The `report_qor_suggestions` Tcl command flags this issue.

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 BUFGCE_DIVIDE set to 1 to match the 150 MHz clock delay accurately, as shown in the following figure.

Figure: 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.

Note: The `report_qor_suggestions` Tcl command provides these constraints.

The following tables compare the clock uncertainty for setup and hold analysis of an UltraScale synchronous CDC timing path.

Table: Comparison of Clock Uncertainty for Setup Analysis of an UltraScale Synchronous CDC Timing Path

Setup Analysis	MMCM Generated 150 MHz Clock	BUFGCE_DIV 150 MHz Clock

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

Table: Comparison of Clock Uncertainty for Hold Analysis of an UltraScale Synchronous CDC Timing Path

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

Related Information

Synchronous CDC

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).

Improving the Netlist with Block-Level Synthesis Strategies

Although 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.
For example, one module might require the use of MUXF* resources to implement a timing critical function, but the rest of the design might benefit from implementation of logic in LUTs rather than MUXF* to reduce congestion. In this case, set the

PERFORMANCE_OPTIMIZED strategy for the timing-critical module, and synthesize the rest of the design using the Flow_AlternateRoutability strategy to reduce congestion.

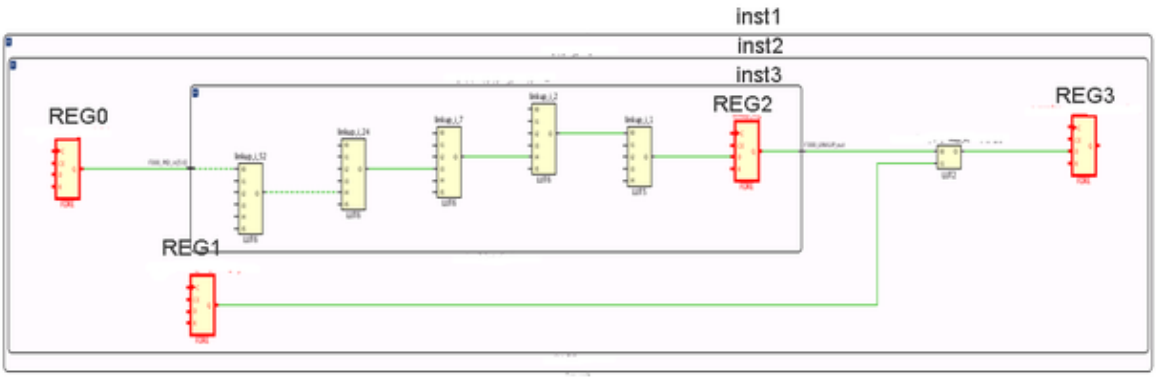
Related Information

Block-Level Synthesis Strategy

Improving Logic Levels

Throughout the design cycle, you must verify that the logic level distribution fits the clock frequency goals for the target Xilinx device 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 five 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: Schematic Showing Critical Path with Five 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).
The following figure shows 58 paths with five logic levels within the inst1/inst2 hierarchy constrained with the 600 MHz clock and 32 paths with only one logic level.

Figure: Logic Level Distribution with Default Synthesis Optimization

```
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 |
+------------------+-------------+---+----+----+---+---+----+---+---+---+---+----+-------+-------+-------+-------+-----+
```

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 three logic levels.

Figure: Logic Level Distribution with Retiming Enabled for Synthesis Optimization

```
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 |
+------------------+-------------+---+---+---+-----+---+---+---+---+---+---+----+-------+-------+-------+-------+-----+
```

Reducing Control Sets

✎ **Note:** 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 eight registers, which all share one clock, one set/reset, and one clock enable. Only one control set can be used per group of eight registers.
- An UltraScale device half-CLB comprises two groups of four registers, which share one clock and one set/reset. In addition, each group of four registers has one 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 two control sets can be used per group of eight 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.

Follow Control Set Guidelines

The following table provides a guideline for the recommended number of control sets, depending on the target device size, for both 7 series and UltraScale devices.

Table: Control Set Guidelines

Guideline	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

Guideline	Percentage of Control Sets
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.

Reduce the Number of Control Sets

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 relying on `place_design` to perform coarse replication and using `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 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. Note that increasing `contol_set_opt` can have negative impacts on power by eliminating clock enables that can actively reduce power. 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.
- Use the CONTROL_SET_REMAP property to map low-fanout control signals driving the synchronous set/reset and/or CE pin of a register to the D-input. For more information, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904).
- 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.

Related Information

Control Signals and Control Sets

Optimizing High Fanout Nets

High fanout nets often lead to implementation issues. Because die sizes increase with each device 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.

Allow Register Replication

Most tools can replicate registers to reduce high fanout nets on critical paths. Alternatively, you can apply attributes on specific registers or levels of hierarchy to specify which registers can or cannot be replicated. For example, the presence of a LUT1 on a replicated net indicates that an attribute or constraint is partly preventing the optimization. During synthesis, a KEEP_HIERARCHY attribute on a hierarchical cell traversed by the optimized net or a KEEP attribute on net segment in a different hierarchy can alter the replication optimizations. During synthesis and implementation, a DONT_TOUCH constraint also prevents beneficial replications.
Sometimes, designers address the high fanout nets in RTL or synthesis by using 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 or is set on a net connected to several major hierarchies. 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 and increase design power by unnecessarily adding additional registers that may not be necessary for timing closure
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.
To restructure and reduce the number of control set trees and high fanout nets, you can use the `opt_design` Tcl command with one of the following options:

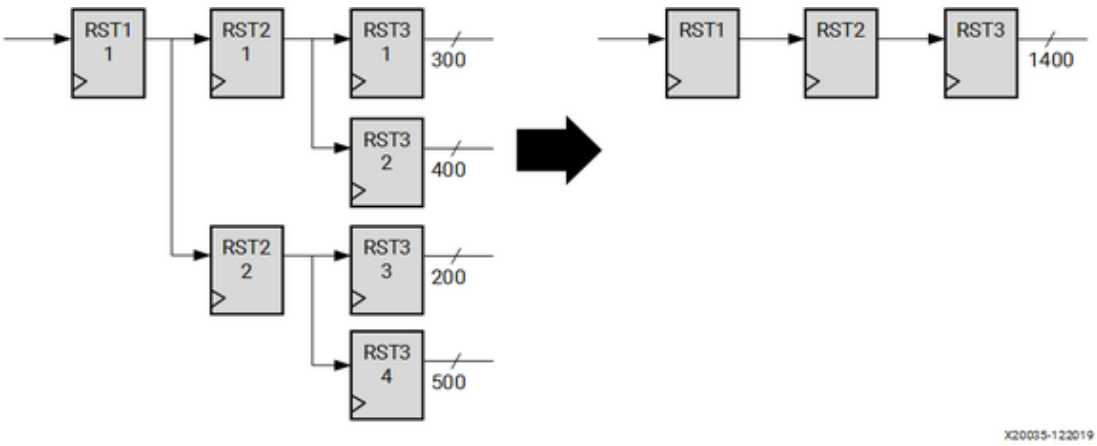
`-control_set_merge`
: This option aggressively combines the drivers of logically-equivalent control signals to a single driver.

`-merge_equivalent_drivers`
: This option merges logically-equivalent signals, including control signals, to a single driver.

✎ **Note:** Try this option first, because the tools are aware of major hierarchies and Pblock constraints when you run this option.

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: Control Set Merging Using opt_design -control_set_merge



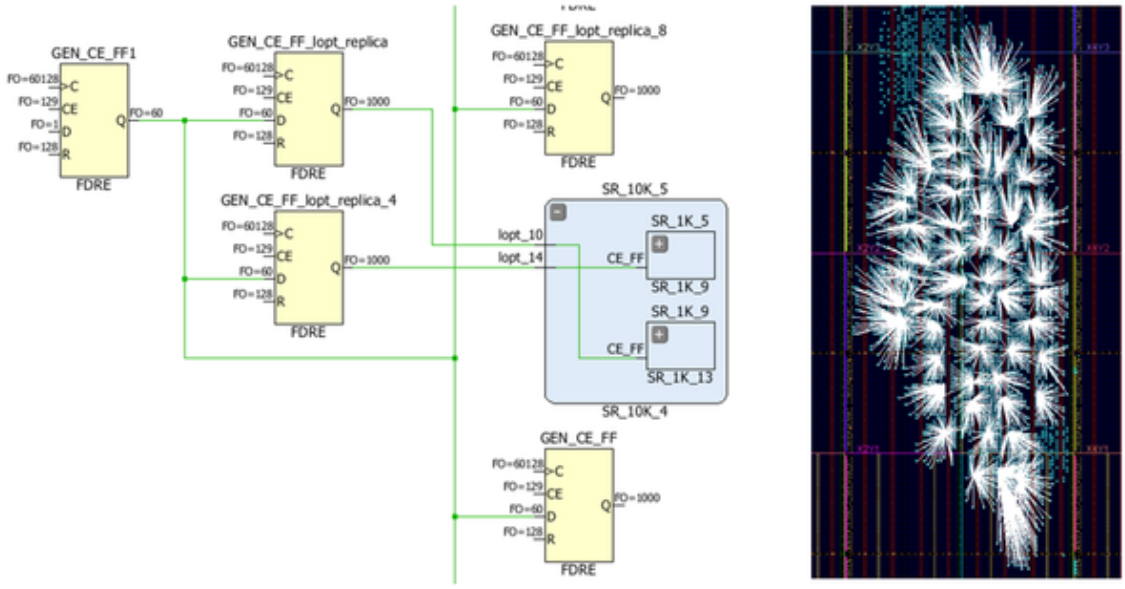
After reducing the number of replicated objects, you can use the `opt_design` Tcl command to perform limited replication based on the hierarchy characteristics, with the following option:

`-hier_fanout_limit <arg>`

This option replicates registers according to the 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 60000 using `opt_design -hier_fanout_limit 1000`. Because each module SR_1K contains 1000 loads, the driver is replicated 59 times.

Figure: Module-Based Replication on a High-Fanout Clock Enable Net



Fanout optimization is enabled by default in `place_design`. 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, and UltraRAMs are considered for replication and are co-located with the loads if replication occurs. You can force the replication of a register or a LUT driving a net by adding the FORCE_MAX_FANOUT property to the net. The value of the FORCE_MAX_FANOUT specifies the maximum physical fanout the nets should have after the replication optimization.

You can force replication based on physical device attributes with the MAX_FANOUT_MODE property. Supported MAX_FANOUT_MODE properties are CLOCK_REGION, SLR, MACRO. For example, the MAX_FANOUT_MODE property with a value of CLOCK_REGION replicates the driver based on the physical clock region, the loads placed into same clock region will be clustered together. For more information, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904).

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.

Related Information

Replicate High Fanout Net Drivers

Promote High Fanout Nets to Global Routing

✎ **Note:** 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 and the clock period of the logic driven by the net is above the limit specific to the targeted device and speed grade.
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 ensures 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.
The placer also automatically routes high fanout nets (fanout > 10000) 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.

Related Information

Control Signals and Control Sets

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 Command

You can use the `group_path` command with the `-weight` option to give higher priority to the path endpoints defined in a clock 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 that belong to clock group `clock` 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. Reducing hold violations prior to routing can be beneficial if hold fixing has been identified as a source of routing congestion. The `phys_opt_design` hold fixing options each use different resources and have specific targets. It is important to use the proper option depending upon the device utilization and desired impact. Prior to running `phys_opt_design` for hold fixing, it is important to validate that the design has properly constrained clocktrees for minimal skew.
The insertion of negative-edge triggered registers between sequential elements can split a timing path into two half period paths and significantly reduce hold violations. 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: Fixing Hold Violation with Negative Edge Register Insertion

You can also insert LUT1 delays onto datapaths to reduce hold violations. To insert LUT1 delays, use one of the following options during the `phys_opt_design` implementation step:

`-hold_fix`

Performs LUT1 insertion and only considers paths that are the largest WHS violators with sufficient positive setup slack.

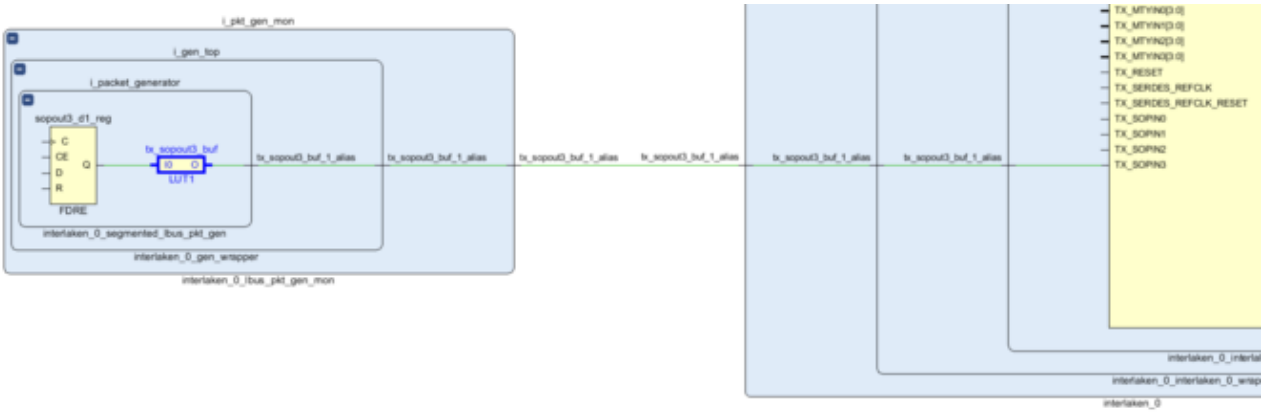
`-aggressive_hold_fix`

Performs LUT1 insertion in a more aggressive manner than the standard `-hold_fix` option. The `-aggressive_hold_fix` optimization considers many hold violating paths for LUT1 insertion and can be used to significantly reduce design THS at the expense of LUT utilization.

✎ **Note:** The `phys_opt_design -directive ExploreWithAggressiveHoldFix` directive runs the `Explore` directive along with the `-aggressive_hold_fix` as a single optimization.

The following figure shows a LUT1 delay inserted after a flip-flop driving an ILKN block. Before the optimization, the path from the flip-flop to the ILKN is the WHS path in the design with -0.277 ns hold slack. After the insertion of the LUT1 delay (highlighted in blue), the hold slack is positive and the setup slack remains positive.

Figure: Fixing Hold Violation with LUT1 Delay 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. 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. The `report_qor_suggestions` can automate the resolution of many of the items that cause congestion.

★ **Tip:** Before you try to address congestion with the techniques discussed in the following sections, make sure that you have clean constraints and you followed the clocking guidelines recommended by Xilinx. Excessive hold time failures (or negative hold slack) and clock uncertainties require the router to detour, which can lead to congestion. Avoid overlapping Pblocks, which can also cause 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` to get more accurate numbers, once unused logic has been trimmed instead of after synthesis.

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.
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).

Related Information

Congestion Level Ranges

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.

Reduce MUXF Mapping

★ **Tip:** This optimization technique is automatically applied by the `report_qor_suggestions` Tcl command.

Using MUXF* primitives helps critical paths with many logic levels or a tight clock requirement while also reducing power. MUXF* includes MUXF7, MUXF8, and MUXF9, which are dedicated multiplexer resources located within the CLB. These resources are grouped with up to eight LUTs during placement. This grouping forces high CLB input utilization with higher routing demand and limits placement flexibility when the netlist

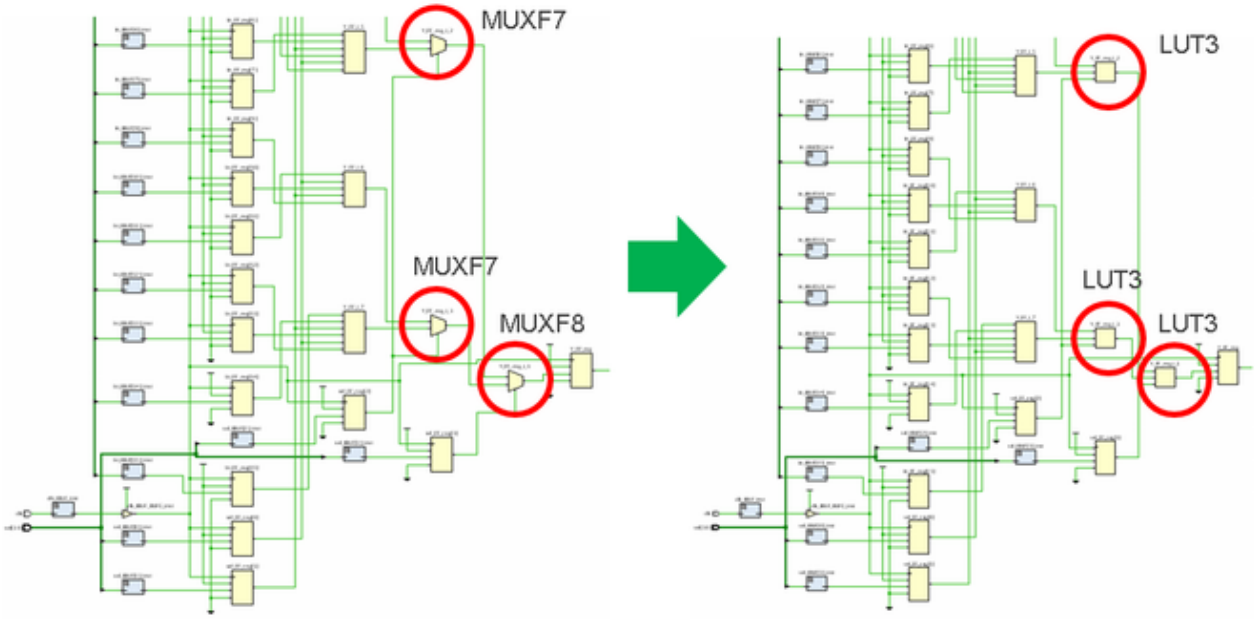
connectivity is complex, leading to potential higher routing congestion and timing degradation.

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.

✎ **Note:** Disabling these resources can result in increased power. Use this method only when needed to achieve timing closure.

The following figure shows a 16-1 MUX before and after the MUXF* optimization.

Figure: 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.

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.

Figure: South Short Congestion in the report_design_analysis Congestion Table

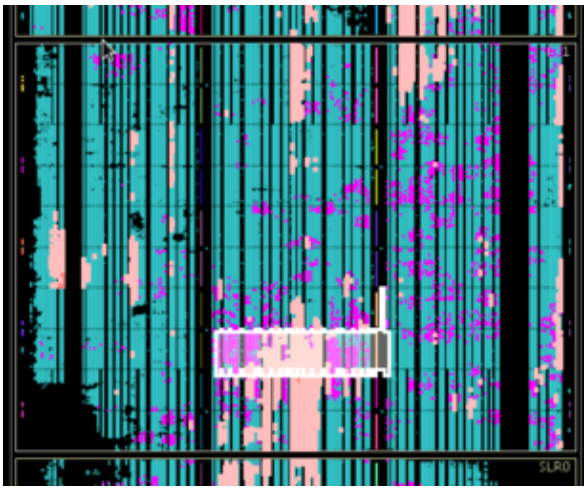
Direction	Type	Congestion Level	Percentage Tiles	Congestion Window	Cell Names	Combined LUTs	LUT6	...	MUXF
...									
North	Short	4	6.983%	(CLEL_L_X48Y73,CLEL_R_X63Y88)	inst_name1(92%)	0%	22%	...	4%
South	Short	5	7.136%	(CLEL_L_X32Y297,CLEL_R_X63Y328)	inst_name2(99%)	2%	49%	...	26%
East	Short	4	6.005%	(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%

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: MUXF Congestion Highlighted in the Vivado IDE Device Window



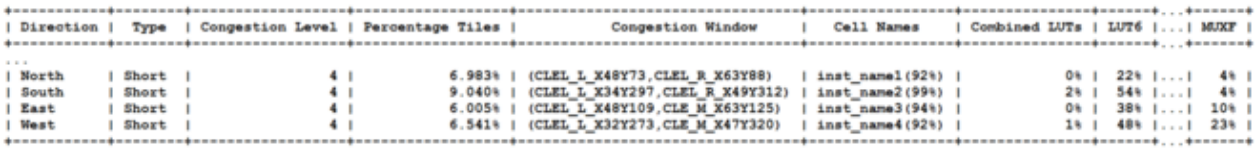
When high MUXF* utilization overlaps with areas of higher congestion, Xilinx recommends reducing the number of MUXF* by mapping their corresponding functionality to LUTs, which have higher placement and routing flexibility. You can use the following command in the XDC synthesis constraints to modify 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.

Figure: Initial Router Congestion Table after Reducing MUXF Usage on a Module



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%

Disable LUT Combining

✎ **Note:** This optimization technique is automatically applied by the `report_qor_suggestions` Tcl command.

LUT combining reduces logic utilization by combining LUT pairs with shared inputs into single dual-output LUTs that use both O5 and O6 outputs. However, LUT combining can potentially increase congestion because it tends to increase the input/output connectivity for the slices. If LUT combining is high in the congested area (> 40%), you can try using a synthesis strategy that eliminates LUT combining to help alleviate congestion. The `Flow_AlternateRoutability` synthesis strategy and directive instructs the synthesis tool to not generate any additional LUT combining.

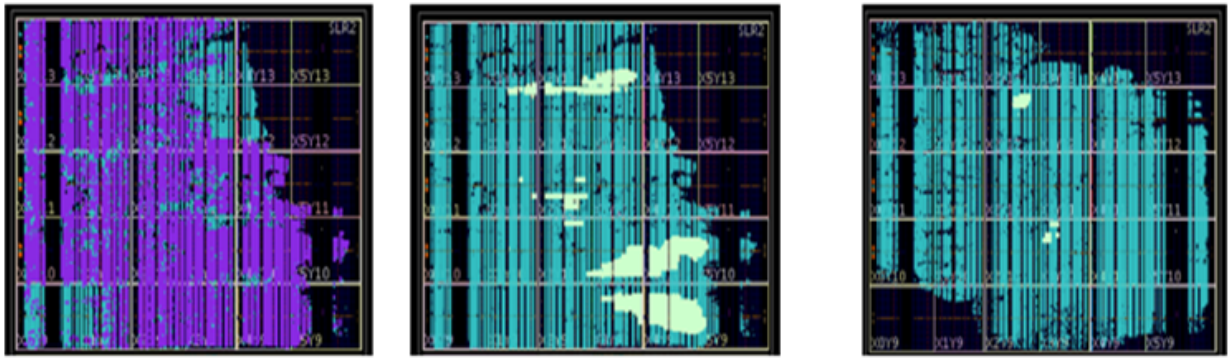
✎ **Note:** If you are using Synplify Pro for synthesis, you can use the Enable Advanced LUT Combining option in the Implementation Options under the Device tab. This option is on by default. If you are modifying the Synplify Pro project file (`*prj`), the following is specified: `set_option -enable_prepacking 1`.

You can use the following command to select cells with LUT combining enabled in your design:

```
select_objects [get_cells -hier -filter {SOFT_HLUTNM != "" ||
HLUTNM != ""}]
```

The following figure shows the horizontal congestion of a design with and without LUT combining. The cells with LUT combining are highlighted in purple.

Figure: Effect of LUT Combining on Horizontal Congestion



To disable LUT combining on a module that overlaps with areas of higher congestion, use the following Tcl command:

```
reset_property SOFT_HLUTNM [get_cells -hierarchical -filter {NAME
=~ <module name> && SOFT_HLUTNM != ""}]
```

Limit High-Fanout Nets in Congested Areas

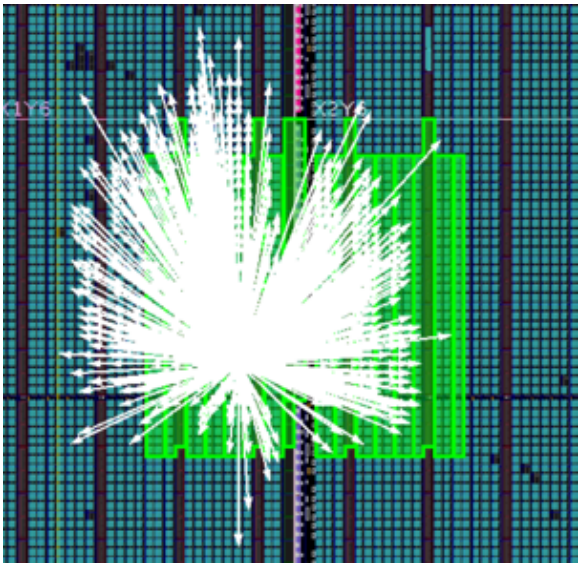
✎ **Note:** This optimization technique is automatically applied by the `report_qor_suggestions` Tcl command.

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: 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 newly 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.

Use Cell Bloating

You can use cell bloating to insert whitespace (increased cell spacing) during the `place_design` step. This leads to a lower density of cells in a given area of the die, which can reduce congestion by increasing available routing. This technique is particularly effective in small, congested areas of relatively high-performance logic.
To use cell bloating, apply the CELL_BLOAT_FACTOR property to hierarchical cells and set the value to LOW, MEDIUM, or HIGH. When working with smaller modules of several hundred cells, HIGH is the recommended setting.

⚠ **CAUTION!** If the device already uses too many routing resources, cell bloating is not recommended. In addition, using cell bloating on larger cells might force placed cells to be too far apart.

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.

Using Strategies and Directives

You can use strategies and directives to find the optimal solution for your design. Strategies are applied globally to a project implementation run. Directives can be set individually for each step of the implementation flow in both Project and Non-Project Modes.

ML Strategies

Machine learning (ML) strategies allow you to quickly obtain an optimized strategy for your design. If you are running multiple implementation strategies to generate implementation results, you can use ML strategies instead to help you predict which results are most likely to generate a good result.

✎ **Recommended:** Xilinx recommends performing three implementation runs with different strategy suggestions to identify and address errors in the prediction.

You can generate strategy suggestion objects on a routed design by running the `report_qor_suggestions` command. Prior to running this command, you must run the implementation flow as follows:

- In Project Mode, use the Default or PerformanceExplore strategy.
- In Non-Project Mode, use the following Tcl commands:
  - `opt_design`: Set the `-directive` option to `Default` or `Explore`.
  - `place_design`, `phys_opt_design`, and `route_design`: Set the `-directive` option to `Default` or `Explore`. The option must match across all three Tcl commands.

After generating ML strategy suggestions, you must write the suggestions using `write_qor_suggestions -strategy_dir <directory>`. This writes one RQS file per strategy. To activate strategy objects, an RQS file with the strategy suggestion must be read using `read_qor_suggestions` prior to running `opt_design`, and the directives for all commands must be set to RQS (for example, `opt_design -directive RQS`).
Xilinx recommends the following when using ML strategies:

- For best results, resolve all methodology checks, and make sure the design has a QoR assessment score of three or higher. To verify, run `report_qor_assessment` after `synth_design` or `opt_design`.
- To further enhance performance, combine ML strategy suggestions with other QoR suggestions in the same RQS file.

> ✎ **Note:** ML strategy suggestions are combined automatically when QoR suggestions are written. To disable this feature, use `write_qor_suggestions -of_objects [get_qor_suggestions ...]`, and filter only the desired suggestions.

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

Predefined Strategies

Xilinx provides a set of predefined strategies that are tuned to be effective solutions for the majority of designs.

> ✎ **Note:** Xilinx does not recommend running the SSI technology strategies for a non-SSI technology device.

Custom Strategies

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 of their functions, enter the implementation command followed by the `-help` option (for example, `place_design -help`). For information on strategies, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904).

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.

Using 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 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.

Overconstraining 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 running 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 run time.

★ **Tip:** An alternative to overconstraining the design is to change the relative priority of each path group. By default, each clock and user-defined path group is analyzed independently with the same priority during implementation. You can set a higher priority for any clock-based path group using the `group_path -weight 2 -name <ClockName>` options. The priority of user-defined path groups cannot be changed.

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, 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.

For SSI technology devices, you can also consider using SLR Pblocks or soft floorplanning constraints (USER_SLR_ASSIGNMENT).

Related Information

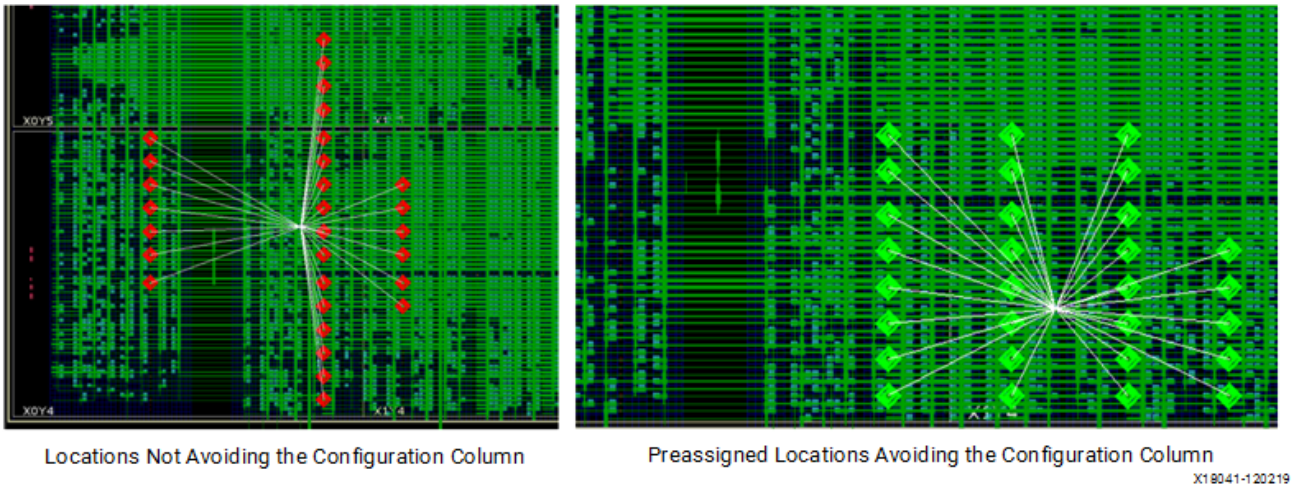
SSI Technology Considerations

Grouping 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: Locations Avoiding the Configuration Column



Reusing Placement Results

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. Following is a simple Tcl script that saves block RAM placement into an XDC file for UltraScale and UltraScale+ device designs.

```
set_property IS_LOC_FIXED 1 \
  [get_cells -hier -filter {PRIMITIVE_TYPE =~ BLOCKRAM.*.*}]
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. Use the Incremental

Compile flow if you make small changes to the design and want to re-use prior placement to achieve more predictable results and faster compile time.

Using Incremental Implementation

You can use incremental implementation to reduce implementation compile time and produce more predictable results. Xilinx recommends making incremental implementation part of your standard timing closure strategies. For more information, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904) and this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906).
This section covers recommendations for automatic incremental implementation, including both high and low reuse modes.

Choose a High Quality Reference Checkpoint

Because the incremental implementation flow depends on reuse, the most important input to the flow is the reference checkpoint. When you use automatic incremental implementation in Project Mode, the Vivado tools manage the updating of the reference checkpoint. This ensures that reuse is high and timing is almost closed.
In all other use cases of the incremental implementation flow, you have control over the selection of the reference checkpoint. Following are guidelines to help improve your selection of the reference checkpoint:

- Use a reference checkpoint that meets timing or is close to meeting timing. If the reference checkpoint is close to meeting timing, it might be beneficial to improve timing as follows before running the incremental implementation flow.

  ✎ **Note:** For automatic incremental implementation, the checkpoint is rejected unless WNS is less than -0.250 ns.

  - Run `route_design -tns_cleanup` to optimize paths that are not the worst case path.
  - Run the post-route `phys_opt_design` command to improve timing failures. Although this command might increase run time, these optimizations are replayed quickly in the incremental implementation run.
  - Use the `report_qor_suggestions` command to generate suggestions to improve the design. New suggestions applied in the incremental implementation flow must be incremental implementation-friendly. Suggestions already applied in the reference checkpoint do not need to be incremental implementation-friendly. For suggestions that are not incremental implementation friendly, consider applying the suggestions and updating the checkpoint using the default flow.
- Select the checkpoint with the lowest congestion, which more readily accommodates changes than congested checkpoints.
- Maximize matching between reference and incremental checkpoints.

  ✎ **Note:** For automatic incremental implementation, the checkpoint is rejected unless cell matching is at least 94% and net matching is at least 90%.

  - Use incremental synthesis to reduce changes introduced into the netlist due to RTL changes. Enable incremental synthesis early in the design closure cycle rather than waiting until you are ready to use incremental implementation.
  - Ensure that `synth_design` and `opt_design` options match for the reference checkpoint and the incremental implementation runs.
  - Match tool versions. Although this is not a requirement, thresholds change and new optimizations are added, which can lead to reduced matching.
  - Avoid using `opt_design AddRemap` and `ExploreWithRemap` directives unless these are the only directives that close timing. These directives have reduced naming consistency when changes are introduced to the codebase.
- Use `report_qor_assessment` to determine whether the design is ready for the incremental implementation flow to be run and whether it is preferable to switch from the default flow.

★ **Tip:** To adjust the incremental implementation thresholds, run `config_implementation -help` for information. To identify differences between the reference and the incremental checkpoints, run `report_incremental_reuse`.

Select Incremental Implementation Directives for High Reuse Mode

You can adjust the incremental implementation flow behavior using directives. The tools follow these directives when the incremental implementation algorithms are used on the implementation run. When flow reverts to the default algorithms, the tools follow the directives specified with the `place_design`, `phys_opt_design`, and `route_design` commands.
Following are the directives available for use with the incremental implementation flow:

RuntimeOptimized
: Targets the WNS from the reference checkpoint. This helps maintain consistency with the reference checkpoint and improves placer and router run time by at least 2x. If the reference checkpoint does not close timing, this directive does not attempt to close timing. This directive is the default.

TimingClosure
: Targets WNS = 0.000 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 run time with more effort to try to meet timing. This mode can improve WNS by up to 250 ps on difficult designs. Use this directive with QoR Suggestions for the best chance at closing timing. There is usually a run time hit with this directive.

Quick
: This option is intended for designs that easily meet timing with greater than 99% reuse. Typically, this option is used for ASIC emulation and prototype designs with minor changes that do not impact timing.

Following is an example command for Project Mode:

```
set_property -name INCREMENTAL_CHECKPOINT.MORE_OPTIONS -value {-
directive TimingClosure} -object [get_runs <runName>]
```

Following is an example command for Non-Project Mode:

```
read_checkpoint -incremental -directive TimingClosure
<reference>.dcp
```

✎ **Note:** The `RuntimeOptimized` directive replaces the `Default` mapping directive, and the `TimingClosure` directive replaces the `Explore` mapping directive from previous Vivado Design Suite releases.

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 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, including checking failing checkpoints to identify the difference between good and bad runs.
  - Identify runs that have a good WNS and low congestion levels.
  - Use floorplanning to define SLR placement.
- After determining the area to target, compare a set of runs using low reuse mode against a baseline set of runs using the default flow to evaluate effectiveness.
  - Use different `place_design` directives to generate multiple results for comparison.

✎ **Note:** In low reuse mode, incremental implementation directives are ignored, and target WNS is always 0.000 ns.

To reuse only block memory placement, use the following Tcl script:

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

To reuse only DSP placement, use the following Tcl script:

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

To reuse both Block Memory and DSP placement, use the following Tcl script:

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

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_objects [get_cells
<cell_name>]
```

✎ **Recommended:** When reusing a hierarchical module, Xilinx recommends using out-of-context synthesis or incremental synthesis with a PRESERVE_BOUNDARY constraint to ensure cell matching is 100%.

Avoid Floorplanning and Overconstraining

When using the incremental implementation flow, avoid the following:

- Do not floorplan incremental implementation runs.
  Pblock placement is overridden by reference checkpoint placement.
- Do not overconstrain the placer.
  Overconstraining the design in the incremental implementation run can severely impact reuse, because the tools try to meet a target WNS that is artificially altered.

Related Information

Overconstraining the Design

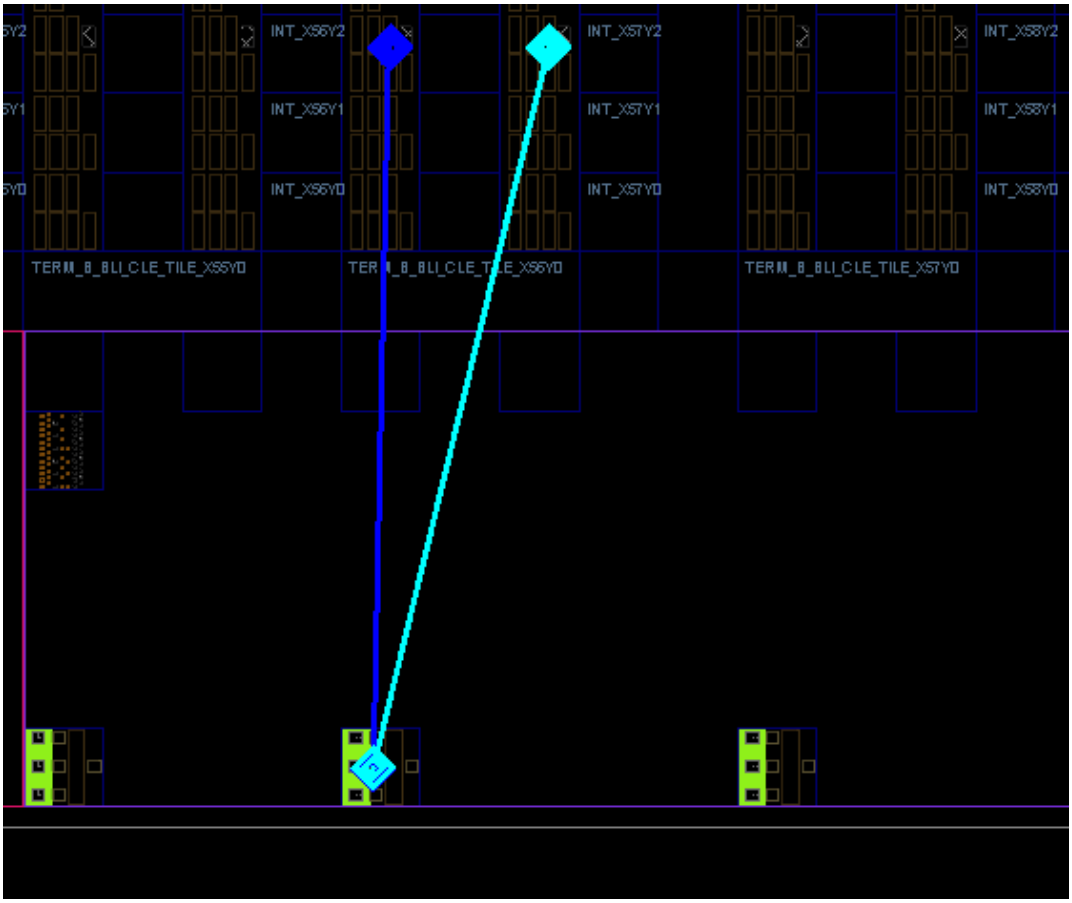
XPIO-PL Interface Techniques for Timing

Boundary logic interface flip-flops exist in hardware between the XPIO-programmable logic (PL) interface, which you can use to improve timing. Dedicated blocks in the XPIO such as the XPHY Logic, I/O Logic, and clock-modifying blocks have boundary logic interface flip-flops. You can apply boundary logic interface (BLI) constraints to flip-flops in your design to automatically take advantage of this hardware feature during design placement. In this example, the data paths to and from the I/O Logic cells ODDRE1 and IDDRE1 in the XPIO are taking advantage of the BLI FFs.

```
set_property BLI TRUE [get_cells {oddr_D1_BLI_reg
oddr_D2_BLI_reg}]
set_property BLI TRUE [get_cells {iddr_Q1_BLI_reg
iddr_Q2_BLI_reg}]]
```

The following figure shows the resulting placement and connectivity from setting the BLI property to TRUE.

Figure: Placement of XPIO-PL Interface BLI Flip-Flops for ODDRE1 and IDDRE1



SSI Technology Considerations

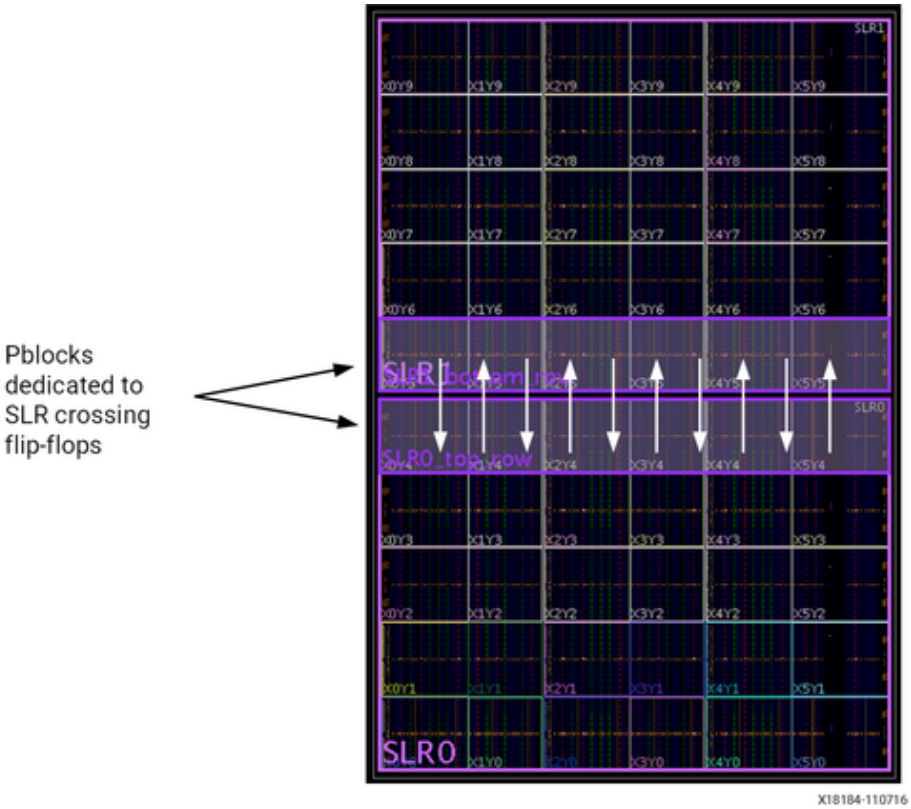
Stacked silicon interconnect (SSI) technology devices consist of multiple super logic regions (SLRs), joined by an interposer. The interposer connections are called super long lines (SLLs). There is some delay penalty when crossing from one SLR to another. To minimize the impact of the SLL delay on your design, floorplan the design so that SLR crossings are not part of the critical path. Minimizing SLR crossings through floorplanning by keeping a challenging module within one SLR only can also improve timing and routability of the design targeting SSI technology devices.

Using Hard SLR Floorplan Constraints

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

Figure: SLR-Crossing Pblock Example



!! Important: Xilinx recommends using CLOCKREGION ranges instead of LAGUNA ranges for SLR-crossing Pblocks.

★ Tip: You can define SLR Pblocks by specifying a complete SLR. For example, `resize_pblock pblock_SLR0 -add SLR0`.

For more information, see this link in *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906).

✎ Video: For information on using floorplanning techniques to address design performance issues, see the Vivado Design Suite QuickTake Video: Design Analysis and Floorplanning.

Using Soft SLR Floorplan Constraints

For large designs, logic for most of the major blocks fits in one SLR as expected and closes timing after a few design iterations. However, small portions of the logic, especially the connectivity across major blocks and across SLRs, is subject to QoR variation depending on the overall design placement. In such cases, the placer and

physical optimization algorithms need additional flexibility to replicate or move some of the logic to a different SLR to address placement challenges and close timing.
You can use the USER_SLR_ASSIGNMENT property to floorplan the design by assigning large design blocks to SLRs. Set this property to a string value, which is applied to hierarchical cells and ignored on leaf cells. The value you set for this property influences the logic partitioning as follows:

SLR name
When a hierarchical cell is assigned the name of an SLR (SLR0, SLR1, SLR2, etc.), the placer attempts to place the entire cell within the specified SLR.

String value
When a hierarchical cell is assigned an arbitrary string value, the placer chooses the SLR. This prevents cells from being partitioned into multiple SLRs.

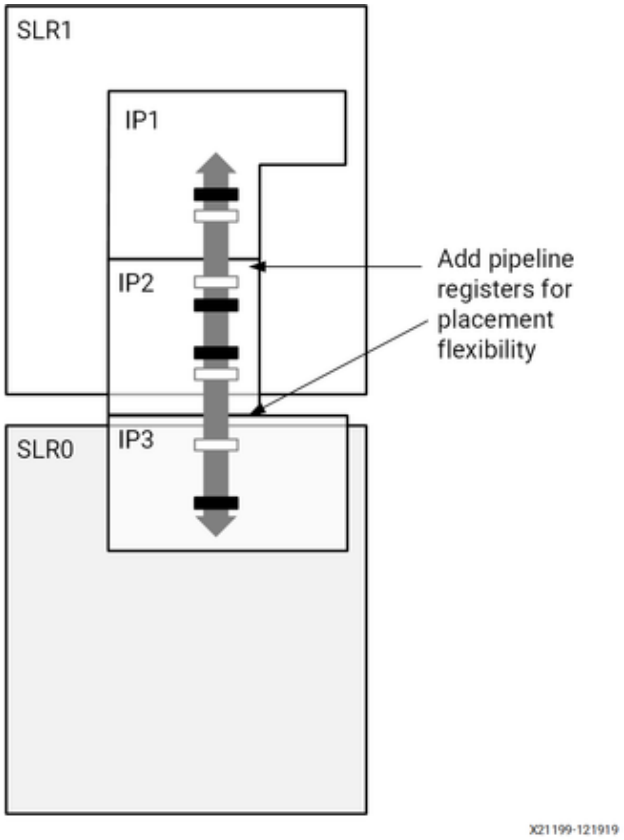
✎ **Note:** If multiple cells have the same USER_SLR_ASSIGNMENT value, the placer attempts to group the cells in the same SLR.

The USER_SLR_ASSIGNMENT property is a soft constraint during SLR partitioning while the Pblock is always a hard constraint during SLR partitioning and global placement. Unlike Pblocks, the USER_SLR_ASSIGNMENT can be ignored by the placer to find a valid SLR partitioning of the design. Both USER_SLR_ASSIGNMENT and Pblocks allow the detailed placer and physical optimization to make fine-tuned adjustments to leaf cell placement near the SLR boundaries to improve timing. These adjustments include moving pipeline registers across SLR boundaries if the moves improve timing. These register moves are not permitted across Pblock boundaries.
In the following example, a design contains three timing-critical hierarchical blocks with cell names IP1, IP2, and IP3 and targets a two-SLR device. To split the three blocks so that IP1 and IP2 are kept together in SLR1 while IP3 is placed in SLR0, the following XDC constraints are applied:

```
set_property USER_SLR_ASSIGNMENT SLR1 [get_cells {IP1 IP2}]
set_property USER_SLR_ASSIGNMENT SLR0 [get_cells IP3]
```

The following figure shows the resulting placement. To improve performance, you can incorporate extra pipeline stages to traverse distances within the device. This is particularly helpful along expected SLR crossings, between IP2 and IP3 in this example. During detail placement and `phys_opt_design`, the pipeline registers from IP2 and IP3 can automatically move across SLR boundaries if this improves timing.

Figure: Placement Example for the USER_SLR_ASSIGNMENT Property



For cases in which you cannot set USER_SLR_ASSIGNMENT or the placer splits challenging paths across SLRs, you can use the USER_CROSSING_SLR property to direct where SLR crossings should or should not occur. Typically, you apply this property to nets or leaf pins where you want pins to be placed in the same SLR as the net driver, or where you want the SLR crossing for the case of a register chain. Set this property to a Boolean value, which is applied to nets and pins to constrain individual SLR crossings:

TRUE

Indicates that the target net object should cross an SLR or the target pin object should be connected across an SLR. You can only apply the TRUE value to register-to-register connections with a single fanout in between.

✎ **Note:** You cannot use the TRUE value for random logic. This option is useful for ensuring a chain of registers always crosses a SLR boundary on a specific register when trying multiple implementation strategies.

FALSE

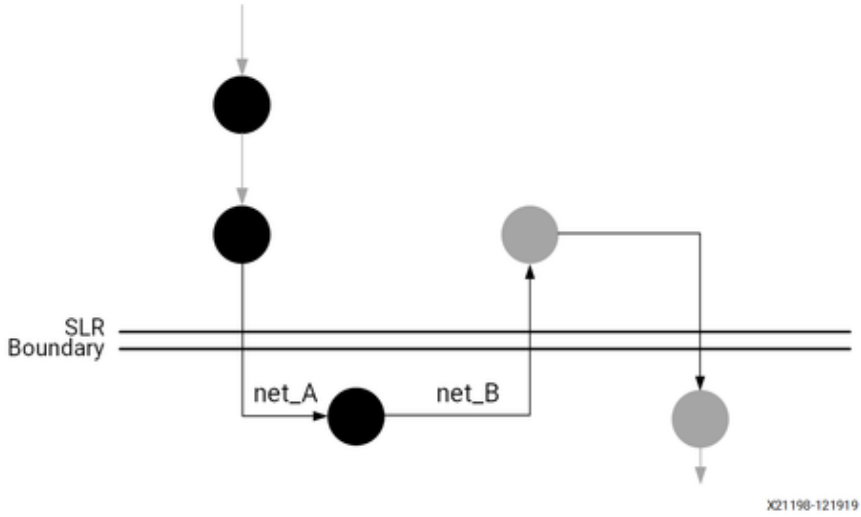
Indicates that the target net object should not cross an SLR or the target pin object should not be connected across an SLR. You can apply the FALSE value to any net or pin.

✎ **Note:** Pins must not be inside macro primitives, because these pins are internal and cannot be constrained.

In the following example, a pipeline register chain crosses an SLR twice, resulting in an unintentional, inefficient zigzag path.

Note: In the next two figures, each dot represents a register stage.

Figure: Suboptimal SLR Crossings Before Setting the USER_CROSSING_SLR Property

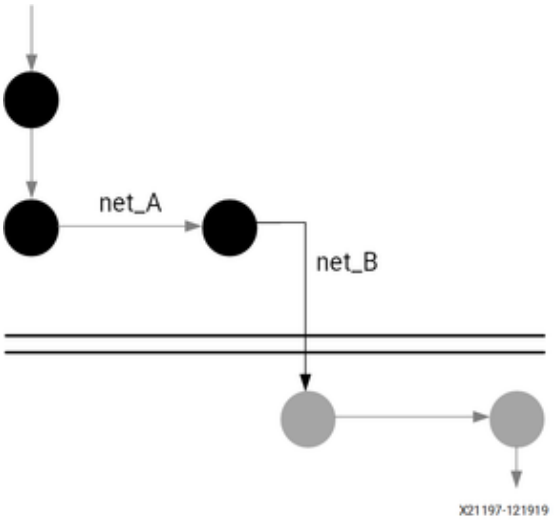


To achieve the optimal placement in which only net_B crosses the SLR, the following XDC constraints are applied:

```
set_property USER_CROSSING_SLR FALSE [get_pins -leaf -of
[get_nets net_A]]
set_property USER_CROSSING_SLR TRUE [get_pins -leaf -of [get_nets
net_B]]
```

The resulting placement contains just a single SLR crossing on net_B as shown in the following figure.

Figure: Optimal SLR Crossings After Setting the USER_CROSSING_SLR Property



Using SLR Crossing Registers

When targeting UltraScale+ SSI technology devices, you can map a register-to-register SLR crossing to a Laguna TX_REG driving a Laguna RX_REG directly. This type of connection is only possible in the UltraScale+ device family, where the Vivado router can fix hold time violations by setting local programmable clock delays. Using the TX_REG to RX_REG SLR crossing topology for pipeline register crossings offers the following performance advantages:

- The placement of SLR crossings spreads vertically, reducing routing congestion near SLR boundaries.
- Locating registers in Laguna sites improves delay estimation accuracy, resulting in higher timing QoR.
- SLR-crossing performance becomes faster and more consistent.

✎ **Note:** When targeting UltraScale SSI technology devices, you can only use a Laguna TX_REG or RX_REG on a SLR crossing net, and you cannot use both at the same time. Performance advantages are similar to the ones listed above.

You can set the USER_SLL_REG property on registers that you expect to be placed at an SLR crossing boundary on a Laguna register site. The USER_SLL_REG constraint is ignored by `place_design` if the register D and Q pins are connected to a net that either does not cross an SLR boundary or drives loads placed in multiple SLRs. For example:

```
set_property USER_SLL_REG TRUE [get_cells {reg_A reg_B}]
```

A reliable method of mapping registered crossings to Laguna is to apply both BEL and LOC constraints to the registers to lock them in place. The LOC value assigns the Laguna site, and the BEL value chooses a particular Laguna register inside the site, one of six TX_REG registers or one of six RX_REG registers. Laguna crossing registers are

a fixed distance apart, which means that each TX_REG register is paired with an RX_REG register for a direct connection.
In the following example, a register-to-register connection is manually placed onto a TX_REG to RX_REG connection. Pipeline register reg_A drives a single fanout with the single load of register reg_B. For a VU9P target device, the following XDC constraints are applied so that reg_A in SLR2 drives reg_B in SLR1 using a direct TX_REG to RX_REG connection:

```
set_property BEL TX_REG3 [get_cells reg_A]
set_property BEL RX_REG3 [get_cells reg_B]
set_property LOC LAGUNA_X2Y480 [get_cells reg_A]
set_property LOC LAGUNA_X2Y360 [get_cells reg_B]
```

The BEL assignments are applied first, and the register position (0, 1, ... 5) must match between TX_REG and RX_REG, which is 3 for this example. Finally, the distance between paired Laguna sites is 120 rows. The register reg_A drives from the bottom row of the SLR2 Laguna column across to the bottom row of the SLR1 Laguna column.
When creating LAGUNA BEL and LOC constraints, try grouping registers with same clock, clock enable and reset signals to avoid control set compatibility issues.

Using Auto-Pipelining for SLR Crossings

Whether you use soft SLR floorplan constraints, hard SLR floorplan constraints, or no floorplan constraints, the number of pipeline stages required to meet timing between major portions of the design located in different SLRs varies based on the following:

- Target frequency
- Device floorplan
- Device speed grade

You can leverage the auto-pipelining feature to allow the placer algorithms to decide on the number of required stages and their optimal location, which helps timing closure across SLR boundaries. When using this feature, the Vivado placer automatically uses Laguna registers without additional intervention.
You can enable auto-pipelining by setting AUTOPIPELINING_* attributes on buses and handshake logic in your RTL, but make sure that the additional latency does not adversely affect the design functionality. Alternatively, you can use the Xilinx AXI Register Slice Memory Mapped or Streaming IP, configured in the SLR crossing.

Related Information

Auto-Pipelining Considerations

Using Intelligent Design Runs

To automatically address most timing closure challenges during implementation, you can use an Intelligent Design Run (IDR). An IDR is a special type of implementation run that leverages `report_qor_suggestions`, ML-based strategy predictions, and incremental compile. An IDR can run up to 6 iterations of place and route, which leads to a typical compile time of 4.5 times that of a standard run. However, using IDR can provide significant benefits by reducing the knowledge required to close timing and by saving hours of user analysis.

✎ **Recommended:** Because an IDR takes longer than a standard implementation run, Xilinx recommends using IDRs less frequently than standard runs. For example, use an IDR after you resolve all methodology warnings and after trying a few common implementation strategies, such as default and explore.

★ **Tip:** To iterate more quickly, you can extract the QoR suggestions and ML strategies from the IDR for use in a standard implementation run. If a significant design change is made, rerun the IDR to update the associated files.

An IDR comprises the following stages:

1. Uses `report_qor_suggestions` to apply optimization properties to elements in the design in a predetermined order.
2. Uses machine learning (ML) strategies to generate tool options for `opt_design`, `place_design`, `phys_opt_design`, and `route_design` that are optimized for the design.
3. Uses a Last Mile Timing Closure feature to apply extensive effort on paths that are difficult to resolve to get the final result.

To ensure success when using IDR, follow these requirements:

- The implementation must be project based. For non-project users, the easiest method is to create a post-synthesis netlist-based project using a pre-`opt_design` checkpoint.
- The device must be from either an UltraScale or UltraScale+™ - device-based family.
- The design must have a baseline with accurate and achievable constraints.
- All designs must comply with the recommended methodology, as reported by the `report_methodology` Tcl command.
- An SLR-based floorplan might be required for SSI technology-based devices.
- Apply only automatic implementation suggestions. Text-based suggestions or suggestions with `APPLICABLE_FOR = synth_design` must be applied before starting an IDR.

For more information see this link in the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906).

Power Closure

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, see the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907).

✎ **Recommended:** When targeting UltraScale and UltraScale+™ devices and using the Explore directives or Explore-based strategies, you must manually enable block RAM power optimization by running `power_opt_design` or using `opt_design -bram_power_opt` after `opt_design` runs. Xilinx recommends targeting block RAMs to achieve power reduction.

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, that the board voltage regulators remain within their current operating limits and the design stays within any system power limits. You can then take prompt remedial actions if the power approaches your budget too closely.
Specify a power budget to report the power margin using the XDC constraint:

```
set_operating_conditions -design_power_budget <value in watts>
```

This value is used by the `report_power` command. The difference between the calculated on-chip power and the power budget is the power margin, which is displayed in red in the Vivado IDE if the power budget is exceeded. This makes it easier to monitor power consumption throughout the flow.

★ **Tip:** For UltraScale+ devices, you can export an XDC file from XPE that contains the environment settings, including the XPE estimate that can be used as a power budget constraint. You can override the power budget using either XPE or the XDC. Add the XDC constraints for power margin reporting.

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 and is sorted starting with highest fanout. Review this list for Low confidence levels, which indicate resets with high switching activity and enables with very low or zero switching activity. Both factors contribute to erroneously optimistic power results. For more information, see Power Constraints Advisor in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907).

Recommended Power Constraints

Applying the correct power constraints to a design is critical to design closure. The Vivado tools `report power` command reports the power margin based on the budget applied as well as additional constraints. Following is a list of the minimum recommended constraints. For more information, see the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907).

Minimum Recommended Constraints

The following constraints ensure that the power estimation checks the power budget and uses the worst-case maximum process for static power analysis:

```
set_operating_conditions -design_power_budget <Power in Watts>
set_operating_conditions -process maximum
```

Additional Recommended Constraints

The following constraints define the thermal solution and allow the `report_power` command to estimate the junction temperature and therefore, the static power more accurately:

```
set_operating_conditions -ambient_temp <max Ambient requested for
application is Celsius>
set_operating_conditions -thetaja <the rise in junction
temperature for every watt dissipated, obtained from thermal
simulation, C/W>
```

The Vivado tools `report_power` command also allows you to report power on a on a per regulator or voltage regulator module (VRM) basis using the following constraints.

Creating a New Power Rail

```
create_power_rail <power rail name> -power_sources {supply1,
supply2 ,..}
```

Adding Power Sources to an Existing Power Rail

```
add_to_power_rail <power rail name> -power_sources {supply1,
supply2, ..}
```

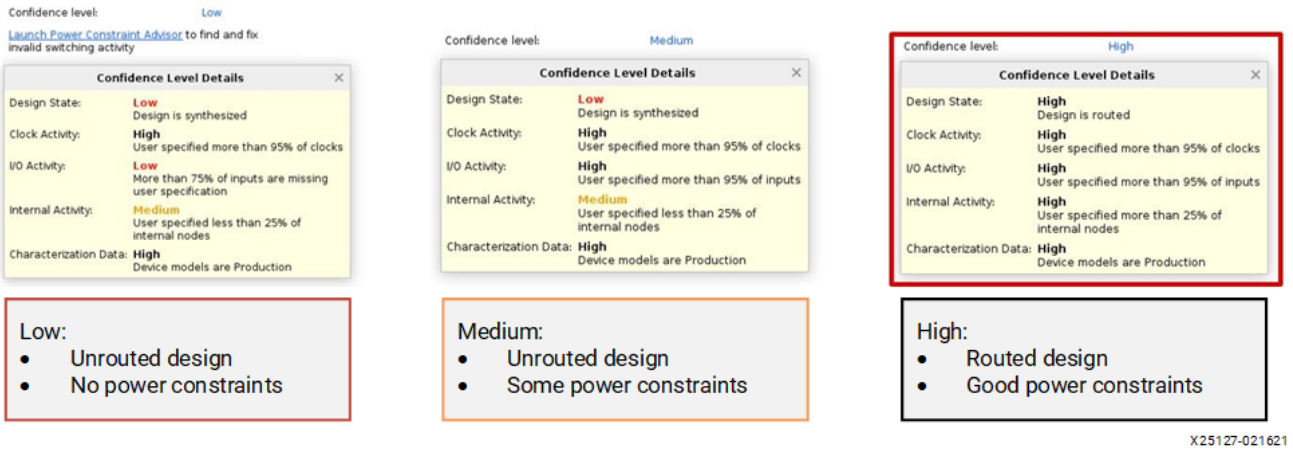
Defining Current Budget

```
set_operating_conditions -supply_current_budget {<supply rail
name> <current budget in Amp>} -voltage {<supply rail name>
<voltage>}
```

Best Practices for Accurate Power Analysis

For accurate power analysis, make sure you have accurate timing constraints, I/O constraints, and switching activity. The `report_power` command indicates a confidence level, as shown in the following figure. Target a High confidence level to ensure accurate power analysis. For more information, see this link in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907).

Figure: Power Analysis Confidence Level



Reviewing the Design Power Distribution After Running 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).

✎ **Power Tip:** Review and validate the decoupling requirement of the completed Vivado design against the current schematic/PCB. You can generate a .xpe file from Vivado tools `report_power` using the following Tcl commands:

```
set_operating_conditions -process maximum

set_operating_conditions -ambient_temp <max Ambient requested for
application is Celsius>

set_operating_conditions -thetaja <the rise in junction
temperature for
every watt dissipated, obtained from thermal simulation, C/W>

report_power -xpe {C:/Design_Runs/Vivado_export.xpe} -name
{Any_Name}
```

You can then import the .xpe file into XPE. The XPE Power Delivery sheet shows the decoupling requirement based on the power estimation and power delivery option.

Further Refining Control Signal Activity After Running 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 Further Refining Control Signal Activity in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907).

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.

  ✎ **Note:** If the design has a timing margin, conduct multiple runs to evaluate if any of the runs have a better total power. For example, a design that has 2 ps of margin can perform similarly to a design with 15 ps, but the 2 ps design might have lower power.

Running Power Optimization

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 focuses 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.

Related Information

Coding Styles to Improve Power

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 have the `IS_CLOCK_GATED` property with value `TRUE`, indicating that the paths are the result of a power optimization. To improve timing at the expense of increased power in a subsequent implementation, use the `set_power_opt` XDC constraint to disable power optimization on the power-optimized cells in the critical path. Then rerun implementation with the `set_power_opt` XDC constraints or Tcl commands.
The following Tcl example disables power optimization on cells in the top 100 failing paths:

```
set pwr_critical_cells [get_cells -of [get_timing_paths -
slack_lesser_than 0 -max_paths 100] -filter {IS_CLOCK_GATED}]
set_power_opt -exclude_cells $pwr_critical_cells
```

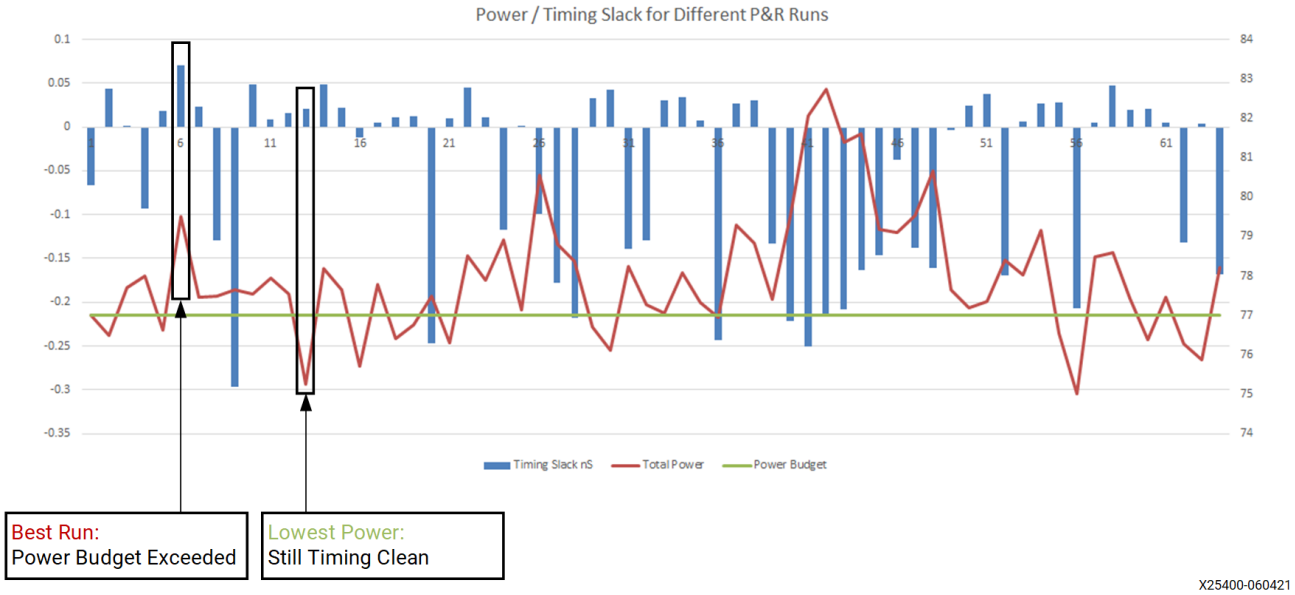
Power Timing Slack

When closing a design for timing, it is more efficient and effective to simultaneously close the design from a power perspective. This approach allows for the best run selection that satisfies both criteria. To close both timing and power, add the report_power constraint to the script you are running. For more information and an example script, see Xilinx Answer Record 76056.
The following figure shows an example of this approach. For all 64 timing closure runs, report power was also run, and all runs are plotted together. From the graph, 36 runs

were timing clean, and from a power perspective, the total power budget is 77W. The 64 runs were in the range of 75W to 83W, an 8W or ~10% range.
Looking at the best run from a timing perspective, run #6 had a power estimate of 79.5W, which exceeds the total power budget. However, from the timing clean runs, run #13 yielded the lowest power at 75W and was still timing clean. Understanding the design from both a timing and power perspective allows you to select the best run for both, without impacting the timing result. In this example, this approach enabled a 4W power saving.

✎ **Power Tip:** You can also improve design power by removing the DONT_TOUCH constraint to allow upfront logic trimming, including clocking primitives.

Figure: Power and Timing Slack for Different Place and Route Runs



Configuration and Debug

After successfully completing the design implementation, the next step is to load the design into the device and run it on hardware. Configuration is the process of loading application-specific data into the internal memory of the 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)
- *Vivado Design Suite Tcl Command Reference Guide* (UG835)
- *7 Series FPGAs Configuration User Guide* (UG470)
- *UltraScale Architecture Configuration User Guide* (UG570)
- Vivado Design Suite QuickTake Video: How To Use the "write_bitstream" Command in Vivado

Configuration

You must first successfully synthesize and implement your design 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 bitstream (.bit or .rbt).
- Select Tools > Edit Device 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 for your device.

★ Tip: If configuration is not successful, you can use a JTAG readback/verify operation on the 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 IP that allows you to observe and control the design. After debugging, you can remove the instrumentation or special IP to increase performance and logic reduction.
Debugging a design is a multistep, iterative process. Like most complex problems, it is best to break the 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 devices and in-system debugging of the design from the Program and Debug section of the Flow Navigator 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 the *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

Debugging the PL

Debugging the programmable logic (PL) can be necessary if you encounter situations that are difficult to replicate in PL logic simulation. This section covers the debugging tools that allow visibility into the PL domain.

Using ILA Cores

The Integrated Logic Analyzer (ILA) core allows you to perform in-system debugging of post-implementation designs on a 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.

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: Debugging Flows

Debugging Flow Name	Flow Steps	Pros/Cons
HDL instantiation probing flow	Explicitly attach signals in the HDL source or IP-Integrator canvas 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. • This method provides the option to probe at the HDL design level. • Allows for probing certain protocols such as AXI or AXI4-Stream at the interface level • It is easy to make mistakes when generating, instantiating, and connecting debug cores.

Debugging Flow Name	Flow Steps	Pros/Cons
Netlist insertion probing flow	Use one of the following two methods to identify the signal for debug: • Use the MARK_DEBUG attribute to mark signals for debug in the source RTL code. • Use the MARK_DEBUG right-click menu option to select nets for debugging in the synthesized design netlist. 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. • This method allows probing at different design levels (HDL, synthesized design, system design). • 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.	• This method provides fully automatic netlist insertion. • You can turn debugging on or off by modifying the Tcl commands. • This method does not require HDL source modification.

Related Information

Modifying the Implemented Netlist to Replace Existing Debug Probes

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 timing QoR results.
- Probe nets that are synchronous 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 window), 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.

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:
    1. Open the synthesized design.
    2. Find the `dbg_hub` cell in the netlist.
    3. Go to the Properties window of the `dbg_hub`.
    4. Find property C_CLK_INPUT_FREQ_HZ.
    5. Set it to frequency (in Hz) of the clock that is connected to the `dbg_hub`.
    6. Find property C_ENABLE_CLK_DIVIDER and enable it.
    7. 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 probed waveforms.

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: Impact of ILA Features on Design Timing and Resources

ILA Feature	When to Use	Timing	Area

ILA Feature	When to Use	Timing	Area
Capture Control/ Storage Qualification	To capture relevant data To make efficient use of data capture storage (block RAM)	Medium to High Impact	• No additional block RAMs • Slight increase in LUT/FF count
Advanced Trigger	When BASIC trigger conditions are insufficient To use complex triggering to focus in on problem area	High Impact	• No additional block RAMs • Moderate increase in LUT/FF count
Number of Comparators per Probe Port ✎ **Note:** Maximum is 4.	To use probe in multiple conditionals: • 1-2 for Basic • 1-4 for Advanced • +1 for Capture Control	Medium to High Impact	• No additional block RAMs • Slight to moderate increase in LUT/FF count
Data Depth	To capture more data samples	High Impact	• Additional block RAMs per ILA core • Slight increase in LUT/FF count

ILA Feature	When to Use	Timing	Area
ILA Probe Port Width	To debug a large bus versus a scalar	Medium Impact	• Additional block RAMs per ILA core • Slight increase in LUT/FF count
Number of Probes Ports	To probe many nets	Low Impact	• Additional block RAMs per ILA core • Slight increase in LUT/FF count

★ **Tip:** In the early stages of the design, there are usually many spare resources in the device that can be used for debugging.

ILA Core Designs with High-Speed Clocks

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.

✎ **Note:** 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.

Using VIO Cores

The Virtual Input/Output (VIO) core allows you to monitor and drive internal device signals in real time. Use this core when it is necessary to drive or monitor low speed signals, such as resets or status signals. The VIO debug core must be instantiated in the design and can be used in both Vivado IP integrator block design and RTL. The VIO core is available in the IP catalog for RTL-based designs and in IP integrator.

For information on customizing the VIO core, see the *Virtual Input/Output LogiCORE IP Product Guide* (PG159). For information on taking measurements with a VIO core, see this link in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

VIO Core Considerations

When using VIO cores, consider the following:

- Signals connected to VIO input probes must be synchronous to the clock connected to the VIO `clk` port on the VIO core. Connecting signals that are not synchronous to the `clk` port results in a clock domain crossing at the VIO input probe port.
- Signals driven from VIO output probes are asserted and deasserted synchronous to the clock connected to the VIO `clk` port on the VIO core.
- The VIO core has a relatively low refresh rate because it is intended to replace low speed board I/O, such as push-buttons or light-emitting diodes (LEDs). To capture high-speed signals, consider using the ILA core.

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, Zynq®-7000 SoC, or Zynq UltraScale+ MPSoC.
  - 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.

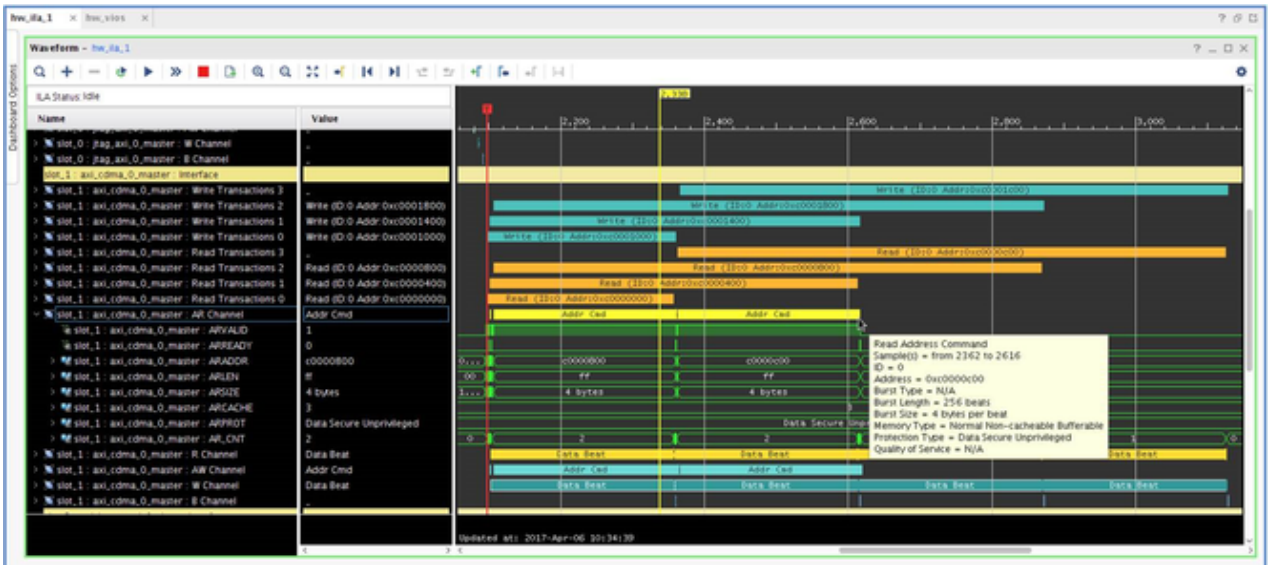
For more information on using System ILA in your IP integrator design, see the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994).

Debugging AXI Interfaces in Vivado Hardware Manager

The ILA 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 changed the ILA mode to Interface, 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 interface slots on the ILA.

Figure: 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).

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).

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.

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).

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).

Connecting a Net to a Free External Pin Using Post-Route ECO

In some cases you might want to bring a net out to a free device pin for debug using external test equipment. This approach can be useful if you are debugging issues that

require minimal changes to the design QoR or require measurements not possible to obtain through other means. You can do this by using the Engineering Change Order (ECO) flow as long as the device has an unused I/O that can be used for this purpose. For more information on using the ECO flow to modify a routed design, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904).

Using Remote Debugging

Xilinx provides the following ways to debug or upgrade your design remotely:

- Use the Xilinx Hardware Server product to connect to a remote computer in the lab.