

Versal ACAP System Integration and Validation Methodology Guide

UG1388 (v2021.1) July 26, 2021

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
07/26/2021 Version 2021.1	
Logic Simulation Using SystemC Models	Added CIPS VIP to table.
NoC Emulation	Added PG313 link and updated title.
Chapter 3: Design Closure	Updated design closure description.
Timing Closure	Added timing result note.
Checking for Valid Constraints	Added baselining design to note.
Checking for Positive Timing Slacks	Updated to timing score description.
Checking That Your Design is Properly Constrained	Added timing constraint note.
Fixing Issues Flagged by report_methodology	Added methodology violation note and link to methodology blog.
Methodology DRCs with Impact on Timing Closure	Added UG906 link.
Assessing the Maximum Frequency of the Design	Updated WNS description.
Analyzing and Resolving Timing Violations	Updated Analyzing and Resolving Timing Violations figure.
Clock Skew and Uncertainty	Added clock uncertainty description and related links.
Timing Closure	Added report_qor_suggestions note throughout subsections.
Reducing Clock Delay in Versal Devices	Added section.
Power Closure	Added power optimization capabilities description.
Power Timing Slack	Added section.
Analyzing System Performance for Platform-Based Designs	Updated title and added traditional design note.
Analyzing AI Engine Performance in Simulation	Added AI Engine bottleneck description.
Measuring Performance with AI Engine Run Time Event APIs	Updated event API code block.
JTAG Status and Error Status	Added POR description.
Rails Voltage Status	Updated Register PWR_SUPPLY_STATUS Bit-Field Details table.
Debugging the NoC	Added section.
Debugging with SmartLynq+	Added SmartLynq+ link.
Using VIO Cores	Added link to PG364.
Using IBERT GTY for Transceiver Link Characterization	Added link to PG331.
Using the ChipScoPy Python Client For Debugging	Added section.
Debugging the Software	Added section.
Performance Validation	Added link to UG1076.

Table of Contents

Revision History.....	2
Chapter 1: Introduction.....	5
About the Versal ACAP Design Methodology.....	5
Navigating Content by Design Process.....	5
About This Guide.....	6
Chapter 2: Simulation Flows.....	7
Logic Simulation.....	7
HLS Simulation.....	8
AI Engine Simulation.....	9
Embedded Software Simulation.....	9
Hardware Emulation.....	10
Chapter 3: Design Closure.....	12
Timing Closure.....	13
Power Closure.....	78
Chapter 4: System Performance Closure.....	83
Analyzing System Performance for Platform-Based Designs.....	83
Improving Performance in the PS.....	89
Improving Performance in the PL.....	91
Improving Performance Through the NoC.....	97
Improving Performance in the AI Engine.....	99
Improving Performance Through the CPM and PL PCIe.....	99
Chapter 5: Configuration and Debug.....	105
Configuration.....	105
Debugging.....	106
Chapter 6: Validation.....	127
Block and IP Validation	127
AI Engine Design Validation.....	129

System Validation.....	131
Design Debug.....	131

Appendix A: Additional Resources and Legal Notices..... 132

Xilinx Resources.....	132
Solution Centers.....	132
Documentation Navigator and Design Hubs.....	132
References.....	133
Training Resources.....	136
Please Read: Important Legal Notices.....	136

Introduction

About the Versal ACAP Design Methodology

The Xilinx[®] Versal[™] adaptive compute acceleration platform (ACAP) design methodology is a set of best practices intended to help streamline the design process for Versal devices. The size and complexity of these designs require specific steps and design tasks to ensure success at each stage of the design. Following these steps and adhering to the best practices will help you achieve your desired design goals as quickly and efficiently as possible.

Navigating Content by Design Process

Xilinx[®] documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal[™] ACAP design process [Design Hubs](#) can be found on the Xilinx.com website. This document covers the following design processes:

- **System Integration and Validation:** Integrating and validating the system functional performance, including timing, resource use, and power closure.

For additional methodology information, see the following documents:

- **System and Solution Planning:** Identifying the components, performance, I/O, and data transfer requirements at a system level. Includes application mapping for the solution to PS, PL, and AI Engine. See the *Versal ACAP Design Guide* ([UG1273](#)) and *Versal ACAP System and Solution Planning Methodology Guide* ([UG1504](#)).
- **Embedded Software Development:** Creating the software platform from the hardware platform and developing the application code using the embedded CPU. Also covers XRT and Graph APIs. See the [Programming the PS Host Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* ([UG1416](#)).
- **AI Engine Development:** Creating the AI Engine graph and kernels, library use, simulation debugging and profiling, and algorithm development. Also includes the integration of the PL and AI Engine kernels. See the *Versal ACAP AI Engine Programming Environment User Guide* ([UG1076](#)) and *Versal ACAP AI Engine Kernel Coding Best Practices Guide* ([UG1079](#)).

- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the Vivado[®] timing, resource use, and power closure. Also involves developing the hardware platform for system integration. See the *Versal ACAP Hardware, IP, and Platform Development Methodology Guide* ([UG1387](#)).
- **Board System Design:** Designing a PCB through schematics and board layout. Also involves power, thermal, and signal integrity considerations. See the *Versal ACAP Board System Design Methodology Guide* ([UG1506](#)).

About This Guide

This guide includes high-level information, design guidelines, and design decision trade-offs for the following topics:

- **Chapter 2: Simulation Flows:** Provides high-level information on simulation flows used for Versal ACAP designs.
- **Chapter 3: Design Closure:** Provides recommendations for meeting timing and power requirements.
- **Chapter 4: System Performance Closure:** Provides recommendations for analyzing and improving the performance of the design.
- **Chapter 5: Configuration and Debug:** Provides an overview of configuration and various methods for debugging the design.
- **Chapter 6: Validation:** Provides methods for validating individual Versal ACAP resources as well as the entire system.

Simulation Flows

To address the different needs in simulation scope, abstraction, and purpose, Xilinx provides dedicated flows for the various components of a Versal™ ACAP design, including the AI Engine, PS, and PL. In addition, Xilinx also provides the ability to co-simulate a complete system comprised of PL, PS and optionally AI Engine components. The following sections provide details on the scope and purpose of each of the simulation flows.

Note: The majority of these simulation flows are available in both the traditional design flow and the platform-based design flow. However, co-simulation of a complete system is only possible in the platform-based design flow.

Logic Simulation

Logic simulation tests a hardware design targeting the PL fabric and is the traditional FPGA simulation flow. The scope of this simulation is scalable, ranging from individual hardware blocks to the complete hardware platform. The simulated models are generally RTL, making the abstraction cycle-accurate. Simulation speed is proportional to the size of the test design, and larger designs take comparatively longer to simulate. To improve simulation performance, you can replace some Versal ACAP IP blocks with SystemC transaction-level models, which simulate faster but are no longer cycle-accurate. The purpose of this simulation is to verify and debug detailed hardware functionality before implementing the design on the device.

Logic simulation is available through the Vivado Design Suite. For more information, see the *Vivado Design Suite User Guide: Logic Simulation* ([UG900](#)).

Note: Logic simulation is possible in both the traditional design flow and the platform-based design flow.

Logic Simulation Using SystemC Models

SystemC is a C++ library that enables hardware modeling. This library provides structural elements, such as modules, ports, and interfaces, as well as data types. In addition to cycle-accurate simulation models, Xilinx provides fast, transaction-accurate, SystemC simulation models for some Versal ACAP infrastructure blocks for use in Vitis hardware emulation flows. SystemC models allow faster simulation compared to RTL models, which helps to reduce overall simulation time.

In general, SystemC models are used for performance analysis, architecture exploration, DMA synchronization, and address trace generation and performance modeling. However, Xilinx recommends using RTL models when accuracy and debugging are more important, such as with DMA transaction or timing dependent issues.

Table 1: Supported Simulation Models for Versal ACAP Blocks

Block	Cycle Accurate	Performance
PS	QEMU (functional only)	QEMU (functional only) CIPS Verification IP (VIP)
NoC	Behavioral SystemVerilog (cycle approximate)	SystemC
DDRMC	Behavioral SystemVerilog	SystemC
PL-based soft memory controller	Behavioral SystemVerilog	Behavioral SystemVerilog
CPM	Behavioral SecureIP	Behavioral SecureIP
GT	Behavioral SecureIP	File I/O (for Vitis software platform users only)
GT-based IP	Behavioral SecureIP	AXI verification IP File I/O (for Vitis software platform users only)
HLS-based IP	RTL	RTL
Other IP	Varies by IP	Varies by IP
PL	Behavioral Verilog VHDL SystemVerilog	Behavioral Verilog VHDL SystemVerilog
AI Engine	SystemC	SystemC

HLS Simulation

HLS simulation exclusively tests HLS code and is an integral part of the HLS development process. The scope of this simulation is a single HLS kernel. Two abstractions are supported, untimed and RTL (cycle-accurate). These two abstractions are referred to as Csim and Cosim respectively. In the Cosim flow, the output of RTL code generated by the HLS compiler is automatically compared against the output of the original C code. The purpose of this flow is to verify the functional correctness of the RTL and to validate performance in a standalone context, independently of interactions with other functions.

HLS simulation is available through the Vitis unified software platform. For more information, see the [Vitis HLS Documentation](#) in the Application Acceleration Development flow of the *Vitis Unified Software Platform Documentation* (UG1416).

Note: HLS simulation is possible in both the traditional design flow and the platform-based design flow.

AI Engine Simulation

You can simulate AI Engine graphs and designs using the following simulator flows. These flows provide a trade-off between simulation speed and accuracy in your AI Engine kernel and graph development phase:

- **Untimed (x86simulator):** Verifies the functional correctness of the code and validates performance in a standalone context, independently of interactions with other functions. This simulation flow is helpful for verifying the functional accuracy of your AI Engine kernels and graphs and offers the quickest simulation run time.
- **Cycle-approximate (aiesimulator):** Calculates core vector load and memory access more precisely, which you can use to estimate AI Engine graph performance and improve the accuracy of the power estimation. In addition, this simulation flow models the GMIO and PLIO interfaces to the AI Engine graph and models the NoC, PS, and PL, providing cycle-approximate results.

Note: This simulation is only possible in the platform-based design flow.

For more information on these simulation flows, see the following resources:

- [Simulating an AI Engine Graph Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416)
- [Vitis Tutorials: AI Engine Development Debug Walkthrough](#)
- [Vitis Tutorials: AI Engine Versal ACAP Integration](#)

Embedded Software Simulation

Embedded software simulation tests a software design that targets only the PS. It is based on the Quick Emulator (QEMU), which emulates the behavior of the dual-core Arm® Cortex®-A72 integrated in the Versal ACAP. This simulation enables a fast, compact functional validation of the platform OS. This flow includes a SystemC transaction-level model of the system, which allows for early system exploration and verification.

Embedded software simulation is available through the Vitis unified software platform. For more information, see this [link](#) in the *Versal ACAP System Software Developers Guide* (UG1304) and the [Xilinx Wiki: QEMU User Documentation](#).

Note: Embedded software simulation is possible in both the traditional design flow and the platform-based design flow.

Hardware Emulation

Hardware emulation simulates a complete Versal ACAP system composed of the AI Engine, PS, and PL. Using the Vitis software platform, you can integrate blocks and functions targeting all three compute domains. The Vitis linker automatically generates a complete co-simulation setup involving RTL, SystemC, and QEMU models:

- Embedded software code running on the PS is emulated using QEMU.
- Code running on the AI Engines is emulated using the SystemC AI Engine simulator.
- User PL kernels are simulated as RTL code.
- IP blocks in the hardware platform are simulated either as RTL or SystemC TLM, based on the types of models available or selected.

As a result, the abstraction of the Vitis hardware emulation is very close to but not fully cycle-accurate. Some details of the Versal ACAP platform are abstracted with TLM models for simulation speed purposes.

The scope of the Vitis hardware emulation also defines its purpose. Hardware emulation allows you to simulate the entire design and test the interactions between the PL, PS, and AI Engine prior to implementation. Because hardware emulation provides full debug visibility into all aspects of the application, it is easier to debug complex problems in this environment than in real hardware.

Hardware emulation is available through the Vitis unified software platform. For more information, see the [Vitis Accelerated Software Development Flow Documentation](#) in the Application Acceleration Development flow of the *Vitis Unified Software Platform Documentation* (UG1416) and the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

Note: Hardware emulation is only possible in the platform-based design flow.

NoC Emulation

NoC emulation support is provided with behavioral models in either SystemVerilog (RTL in GUI) or SystemC (TLM in GUI). The simulation time with the SystemC model is much faster but is cycle approximate and less accurate compared to the SystemVerilog model.

Although you can use both the SystemC and SystemVerilog models to verify functionality, the SystemVerilog model is recommended for performance analysis. Performance analysis using the SystemVerilog model is within $\pm 5\%$ of hardware.

You can emulate the NoC using simulators in the Vivado tools or using the hardware emulation flow provided by the Vitis environment.



IMPORTANT! For more information the NoC simulation settings and performance tuning, see this [link](#) and this [link](#) in the Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide ([PG313](#)).

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.

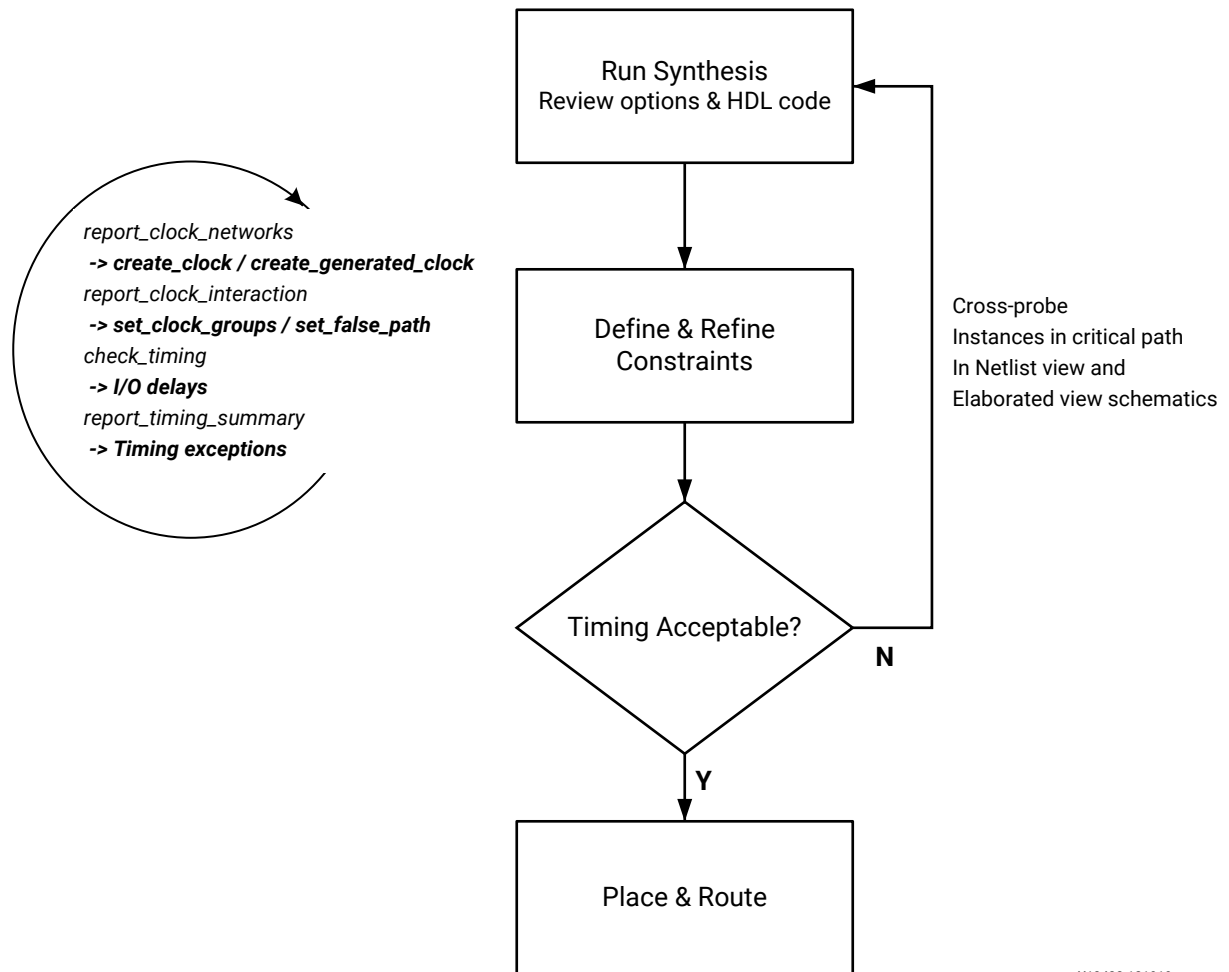


TIP: For more information on reports mentioned in this chapter, see Vivado Design Suite User Guide: Design Analysis and Closure Techniques ([UG906](#)).

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 1: Design Methodology for Rapid Convergence



X13422-121919

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

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](#).

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.

See this [link](#) in *Vivado Design Suite User Guide: Design Analysis and Closure Techniques (UG906)* for more information on some of the following checks.

Also, see the [Adoption Of The Methodology Report](#) blog series to get an understanding on how `report_methodology` helps resolves issues and saves you 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 2: 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
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

Table 2: Timing Closure Methodology DRCs (cont'd)

Check	Severity	Description
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 <code>set_clock_groups</code> 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 3: 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 <code>set_max_delay -datapath_only</code> 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 <code>set_max_delay -datapath_only</code> command
TIMING-29	Warning	Inconsistent pair of multicycle paths

Table 3: Signoff Quality Methodology DRCs (cont'd)

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
TIMING-53	Critical Warning	No common phase between related clocks from DPLL

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:

- The FMAX (MHz) = $\max(1000/(T_i - WNS_i))$

Where:

- T_i is the target clock period (ns) used during the implementation run "i"
- WNS_i 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.

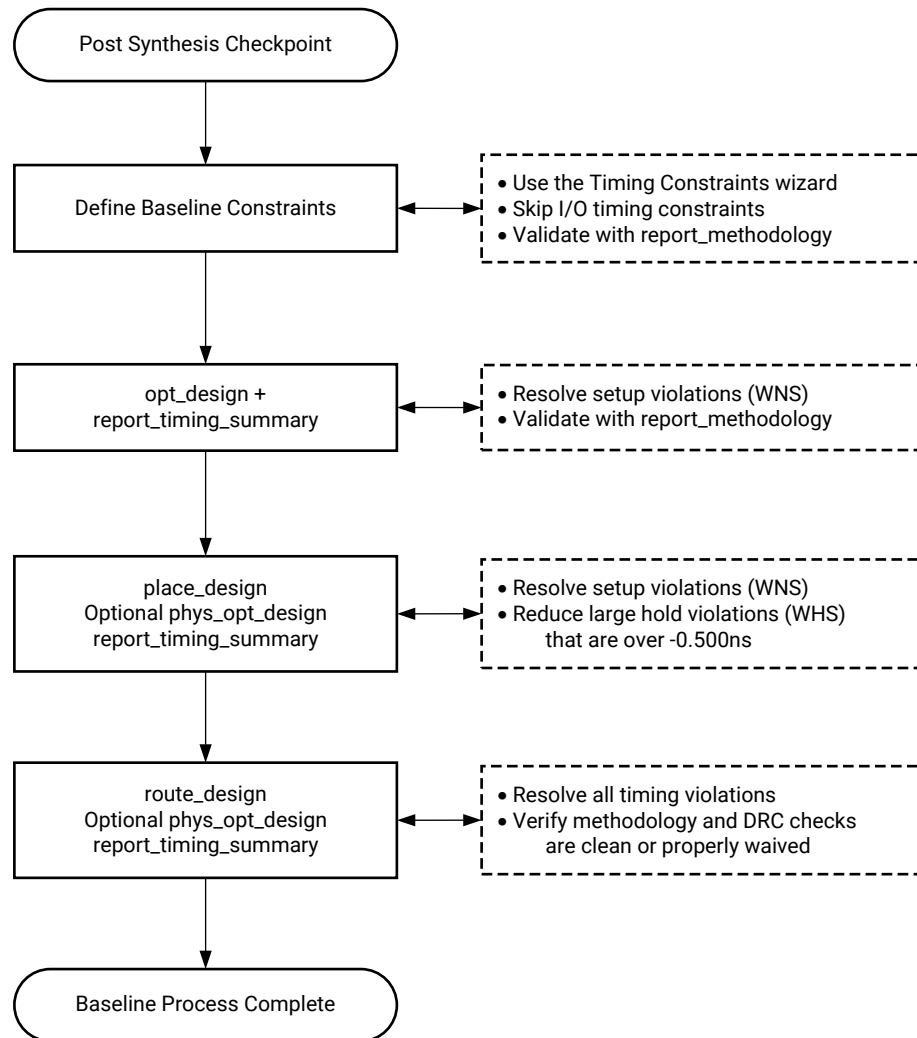
For a given design implementation, the maximum operating frequency on hardware across temperature and voltage ranges supported by the target device speed grade is defined by $1000/(T - WNS)$, with WNS positive or negative. When operating under nominal temperature and voltage conditions, typically in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

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 2: Baselineing the Design



X20037-021821

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, GTs, and clock buffers are 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 3: 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

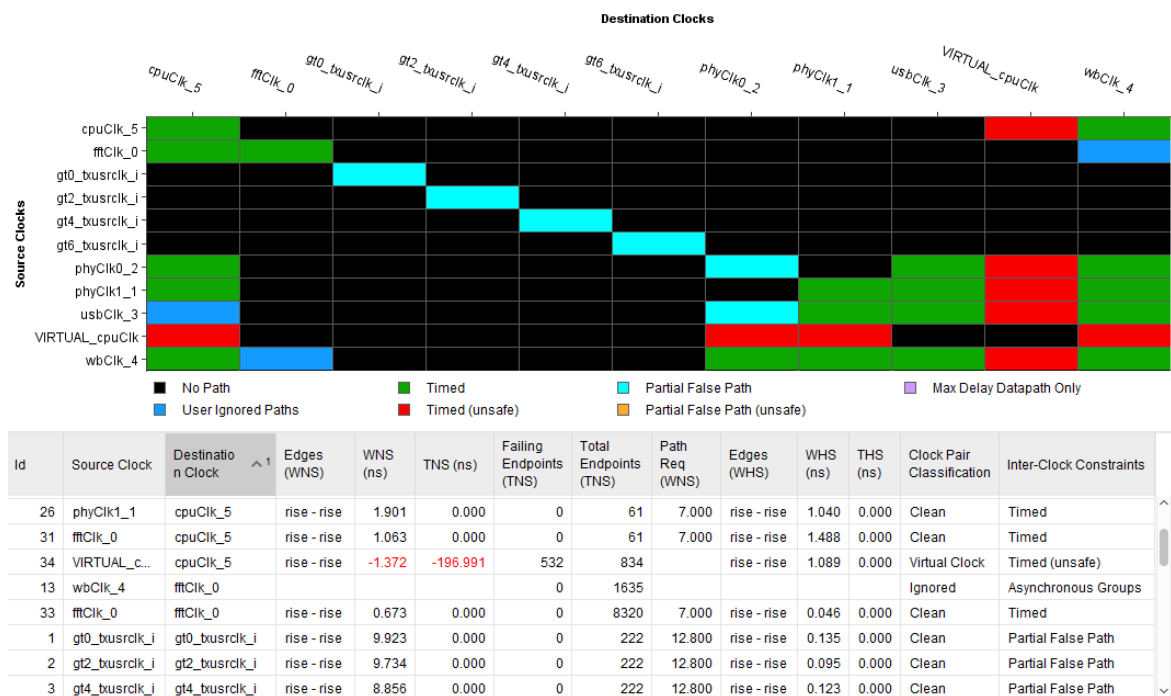
Upon verification of the clocking constraints, you must identify asynchronous and over-constrained clock domain crossing paths.

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 4: Sample Clock Interaction Report



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

Table 4: report_clock_interaction Colors

Color	Label	Meaning	What Next
Black	No path	No interaction among these clock domains.	Primarily for information unless you expected these clock domains to be interacting.
Green	Timed	There is interaction among these clock domains, and the paths are getting timed.	Primarily for information unless you do not expect any interaction among the clock domains.

Table 4: **report_clock_interaction Colors** (cont'd)

Color	Label	Meaning	What Next
Cyan	Partial False Path	Some of the paths for the interacting domains are not being timed due to user exceptions.	Ensure that the timing exceptions are really desired.
Red	Timed (unsafe)	There is interaction among these clock domains, and the paths are being timed. However, the clocks appear to be independent (and hence, asynchronous).	Check whether these clocks are supposed to be declared as asynchronous, or whether they are supposed to be sharing a common primary source.
Orange	Partial False Path (unsafe)	There is interaction among these clock domains. The clocks appear to be independent (and hence, asynchronous). However, only some of the paths are not timed due to exceptions.	Check why some paths are not covered by timing exceptions.
Blue	User Ignored Paths	There is interaction among these clock domains, and the paths are not being timed due to clock groups or false path timing exceptions.	Confirm that these clocks are supposed to be asynchronous. Also, check that the corresponding HDL code is written correctly to ensure proper synchronization and reliable data transfer across clock domains.
Light blue	Max Delay Datapath Only	There is interaction among these clock domains, and the paths are getting timed through: <code>set_max_delay -datapath_only</code> .	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`

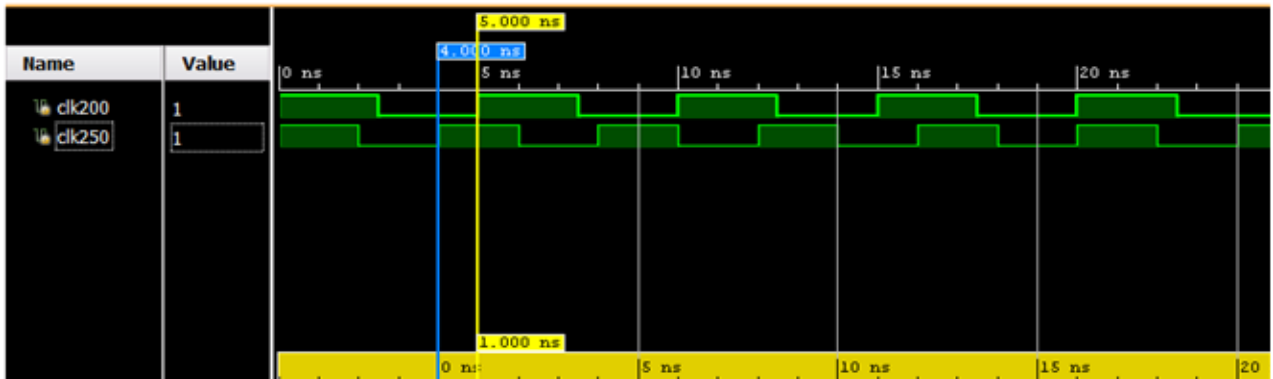
Use one of these constraints when treating the clock pair as asynchronous. Use the `report_cdc` Tcl command to validate that the clock domain crossing circuitry is safe.

- `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.

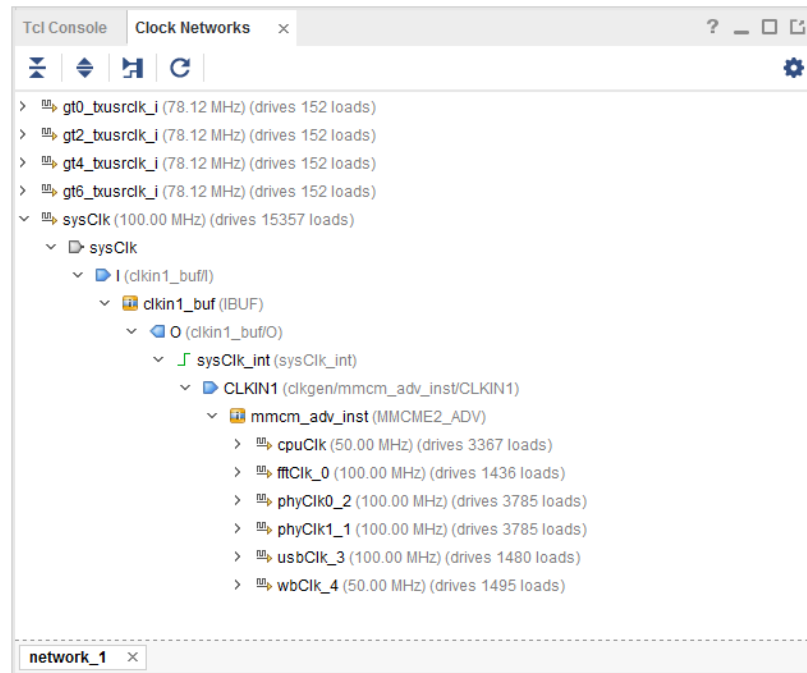
Figure 5: Clock Domain Crossing from 250 MHz to 200 MHz



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 6: Report Clock Networks



```
### Decouple asynchronous clocks
set_clock_groups -asynchronous \
-group [get_clocks sysClk -include_generated_clocks] \
-group [get_clocks gt0_txusrclk_i -include_generated_clocks] \
-group [get_clocks gt2_txusrclk_i -include_generated_clocks] \
-group [get_clocks gt4_txusrclk_i -include_generated_clocks] \
-group [get_clocks gt6_txusrclk_i -include_generated_clocks]
```

Limiting I/O Constraints and Timing Exceptions

Most timing violations are on internal paths. I/O constraints are not needed during the first baselining iterations, especially for I/O timing paths in which the launching or capturing register is located inside the I/O bank. You can add the I/O timing constraints after the design and other constraints are stable and the timing is nearly closed.



TIP: 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 5: Timing Summary Report for Synthesized Design

	WNS	TNS	Num Failing Endpoints	WHS	THS	Num Failing Endpoints
Synth						

3. Open the post-synthesis `report_timing_summary` text report and record the `no_clock` section of `check_timing`.
Number of missing clock requirements in the design: _____
4. Run `report_clock_networks` to identify primary clock source pins/ports in the design.
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 6: Timing Summary Report

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

12. Run `report_exceptions -ignored` to identify if there are constraints that overlap in the design. Record the results.

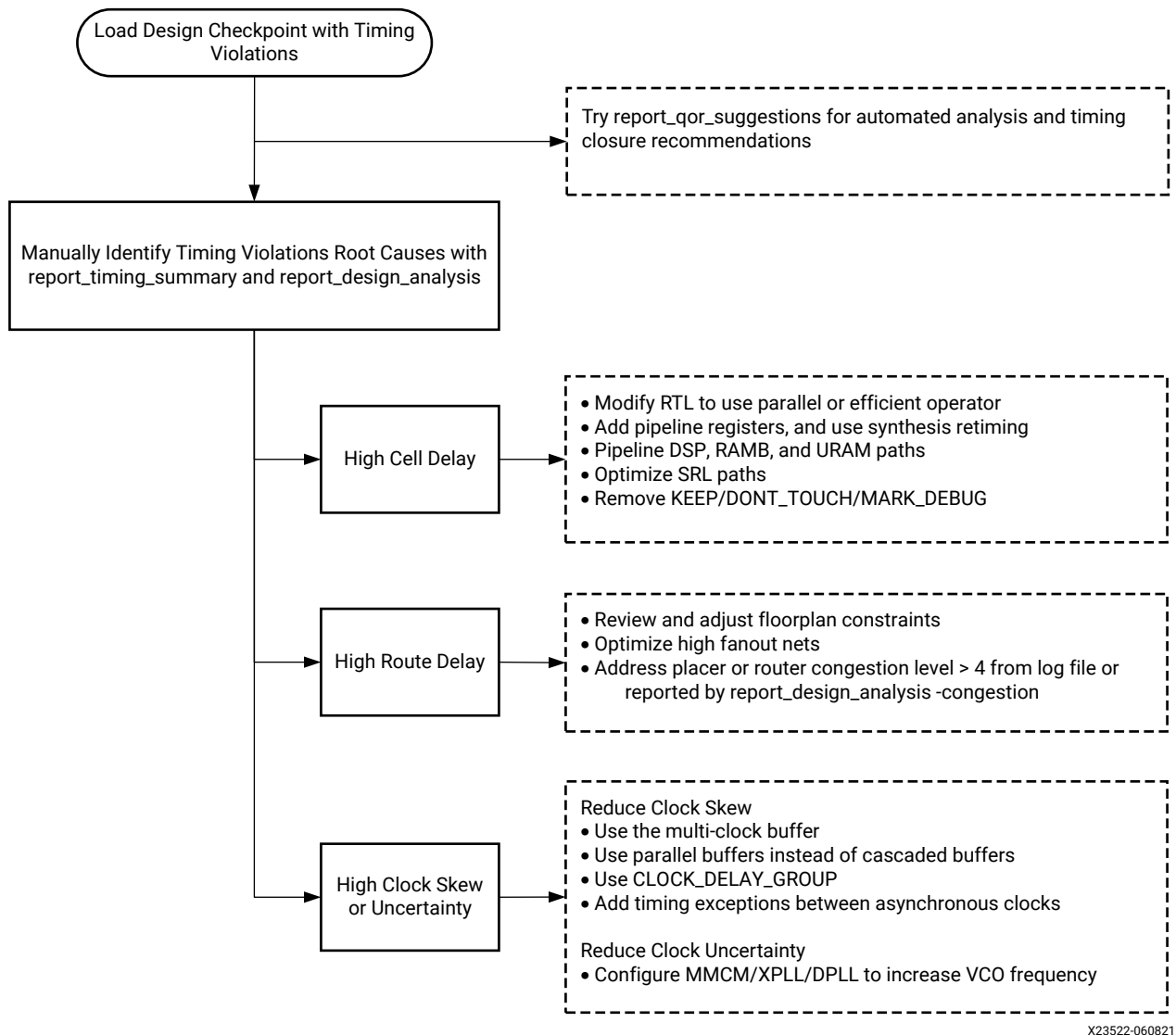
Analyzing and Resolving Timing Violations

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 7: Analyzing and Resolving Timing Violations



Note: For more information on the use of multi-output clock buffers (MBUFG primitives), see this [link](#) in the *Versal ACAP Hardware, IP, and Platform Development Methodology Guide* (UG1387).

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.

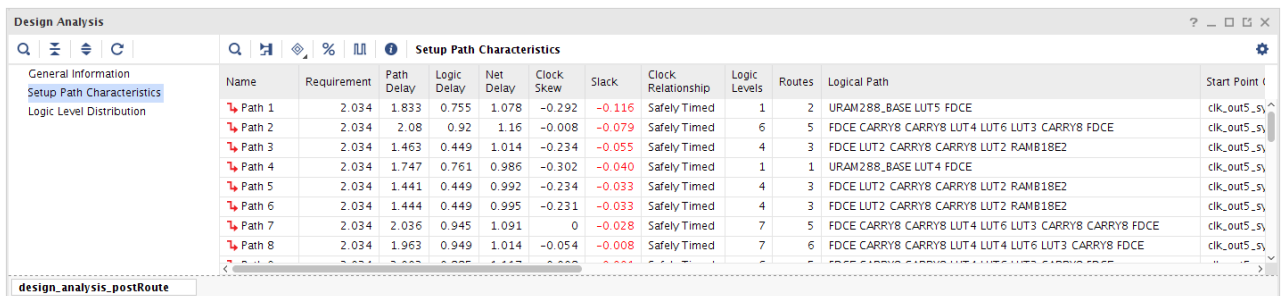
Analyze Path Characteristics

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 8: Report Design Analysis Timing Path Characteristics Post-Route



Name	Requirement	Path Delay	Logic Delay	Net Delay	Clock Skew	Slack	Clock Relationship	Logic Levels	Routes	Logical Path	Start Point (t)
Path 1	2.034	1.833	0.755	1.078	-0.292	-0.116	Safely Timed	1	2	URAM288_BASE LUT5 FDCE	clk_out5_s1
Path 2	2.034	2.08	0.92	1.16	-0.008	-0.079	Safely Timed	6	5	FDCE CARRY8 CARRY8 LUT4 LUT6 LUT3 CARRY8 FDCE	clk_out5_s1
Path 3	2.034	1.463	0.449	1.014	-0.234	-0.055	Safely Timed	4	3	FDCE LUT2 CARRY8 CARRY8 LUT2 RAMB18E2	clk_out5_s1
Path 4	2.034	1.747	0.761	0.986	-0.302	-0.040	Safely Timed	1	1	URAM288_BASE LUT4 FDCE	clk_out5_s1
Path 5	2.034	1.441	0.449	0.992	-0.234	-0.033	Safely Timed	4	3	FDCE LUT2 CARRY8 CARRY8 LUT2 RAMB18E2	clk_out5_s1
Path 6	2.034	1.444	0.449	0.995	-0.231	-0.033	Safely Timed	4	3	FDCE LUT2 CARRY8 CARRY8 LUT2 RAMB18E2	clk_out5_s1
Path 7	2.034	2.036	0.945	1.091	0	-0.028	Safely Timed	7	5	FDCE CARRY8 CARRY8 LUT4 LUT6 LUT3 CARRY8 CARRY8 FDCE	clk_out5_s1
Path 8	2.034	1.963	0.949	1.014	-0.054	-0.008	Safely Timed	7	6	FDCE CARRY8 CARRY8 LUT4 LUT4 LUT6 LUT3 CARRY8 FDCE	clk_out5_s1

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)
- 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 9: Report Design Analysis Timing Path Characteristics Pre-Place

Tcl Console		Design Analysis		Logic Level Distribution																	
General Information		End Point Clock	Requirement	0	1	2	3	4	5	6	7	8	9	10	11-15	16-20	21-25	26-30	31+		
Logic Level Distribution		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 >50% in Versal™ devices.

Can the path be modified to be shorter or to use faster logic cells? See [Reducing Logic Delay](#).

- Route delay is >50% in Versal 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.
- o 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.

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.

- 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)*.

Note: For more information, see this [link](#) in the *Versal ACAP Hardware, IP, and Platform Development Methodology Guide (UG1387)*.

Optimizing Paths with Dedicated Blocks and Macro Primitives

Paths from/to/between dedicated blocks and macro primitives (e.g., DSP, block RAM, UltraRAM, NoC master unit/slave unit (NMC/NSU), AI Engines, and XPIO) 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 1.2 ns without the optional output register and 0.3 ns with the optional output register. Review the data sheet of your target device series for complete details.
- Higher clock-to-output timing arc values for NoC output pins. For example, a NoC NSU has a clock-to-output delay around 0.65 ns.
- Higher routing delays than regular FD/LUT connections.
- Higher clock skew variation than regular FD-FD paths.
- Higher routing delays between the fabric and dedicated blocks on the top/bottom of the device (for example, AI Engines, dedicated blocks within the XPIO, such as XPHY logic, I/O logic, and clocking modifying blocks).

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.

- Use the boundary logic interface (BLI) flip-flops for the placement of pipeline flip-flops interfacing with AI Engines and dedicated blocks within the XPIO, such as XPHY logic, I/O logic, and clock-modifying blocks. Some IP provide an option to utilize the BLI flip-flops.

Related Information

[AI Engine-PL Interface Techniques for Timing](#)

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.

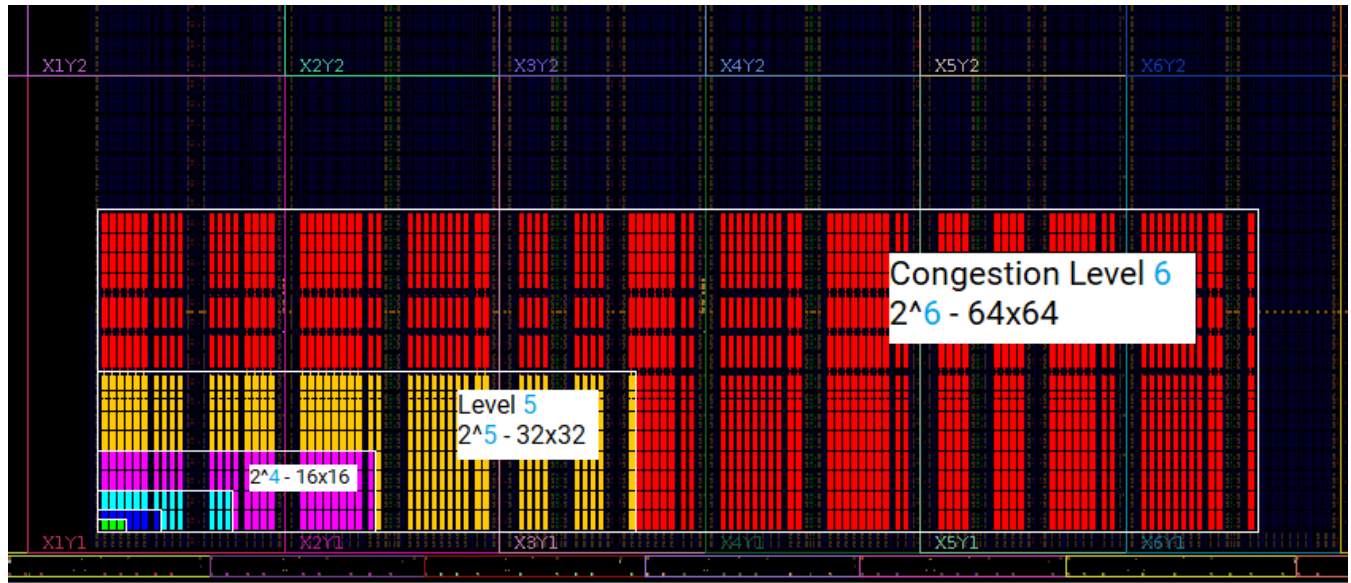
Reducing Net Delay Caused by Congestion

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 10: 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 7: 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

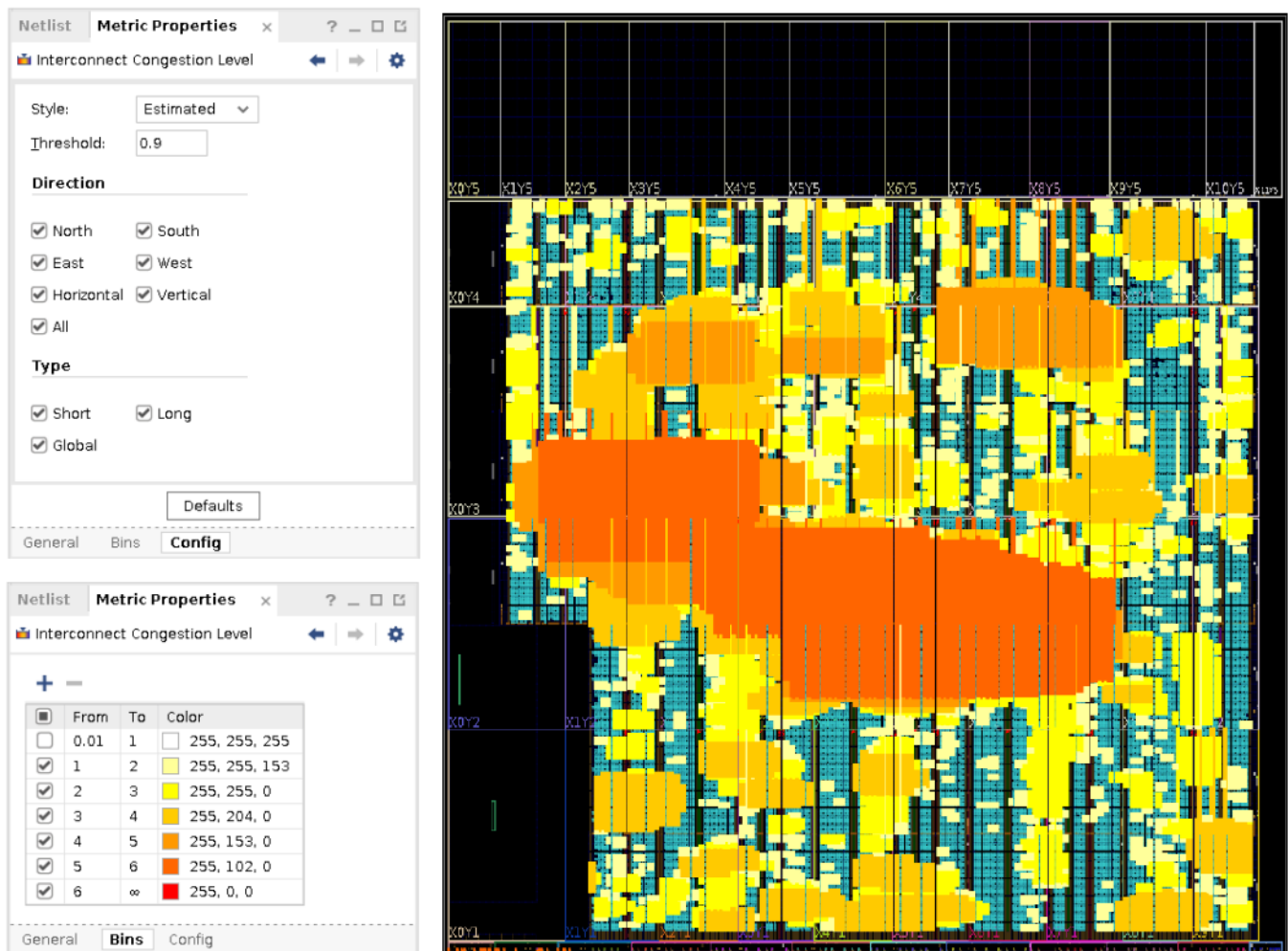
The Interconnect Congestion Level metric highlights the largest contiguous area in which routing resources are overused. By default, this metric is based on estimation, which is similar to the congestion level after initial routing. Actual routing can also be displayed if routing exists. After placement 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 11: Example of Interconnect Congestion Level 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 12: Initial Estimated Congestion Table

INFO: [Route 35-449] Initial Estimated Congestion

Direction	Global Congestion		Long Congestion		Short Congestion	
	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.

Related Information

Addressing Congestion

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 13: Congestion Table

Tcl ConsoleMessagesMetric ResultsDesign Analysis xTiming? _ □ □

Q

≡

⚙

C

Q

🏠

🔍

Placed Maximum

⚙

General Information

▼ Congestion

Placed Maximum

Placed Tile Based (V)

Placed Tile Based (H)

Window	Direction	Congestion Level	Congestion	Cell Names Top Cell 1	Top Cell 2	Top Cell 3	Combined LUTs	LUT6	LUT5	Flop	MUXF	RAMB
Window 1	North	4	120	inst_1022144/inst_1022144/inst_102	inst_1022144/inst_1022134/inst_1018559/inst_990436 (10%)		26%	37%	22%	43%	0%	NA
Window 2	East	4	107	inst_1022144/inst_cv_33 (17%)			26%	29%	7%	60%	1%	50%
Window 3	South	4	131	inst_1022144/inst_1022144/inst_102	inst_1022144/inst_1022134/inst_1018559/inst_990436/inst_979691 (6%)		32%	38%	18%	54%	0%	50%
Window 4	West	2	143	inst_1022144/inst_1022144/inst_102	inst_1022144/inst_1022134/inst_1018559/inst_887992 (9%)		84%	40%	1%	60%	0%	NA

design_analysis_1

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 14: Complexity Report

Instance	Module	Rent	Average Fanout	Total Instances	LUT1	LUT2	LUT3	LUT4	LUT5	LUT6	Memory LUT	DSP	RAMB	MUXF	URAM
cv_33	cv_33	0.41	2.91	1131310	0.7%	11.9%	18.4%	15.7%	17.2%	36.1%	22141	125	913	23685	82
> Inst_1022144 (cv_71)	cv_71	0.42	2.86	1011347	0.6%	11.7%	18.6%	15.8%	17.2%	36.1%	17807	122	810	21452	82
> Inst_1029467 (cv_13905)	cv_13905	0.37	3.44	7236	0.7%	11.9%	10.2%	24.6%	16.2%	36.4%	1472	0	0	3	0
> Inst_1036789 (cv_13934)	cv_13934	0.41	3.44	7236	0.7%	11.9%	10.2%	24.6%	16.2%	36.4%	1472	0	0	3	0
> Inst_1051863 (cv_13963)	cv_13963	0.47	3.01	61384	1.6%	9.1%	19.6%	13.4%	17.7%	38.6%	22	0	68	1892	0
> Inst_1052499 (cv_13973)	cv_13973	0.63	3.16	1366	0.7%	13.3%	12.6%	9.6%	27.5%	36.3%	8	1	4	9	0
> Inst_1055086 (cv_13982)	cv_13982	0.42	2.64	2525	2.3%	25.4%	12.7%	21.7%	11.4%	26.4%	4	0	6	0	0
> Inst_1059242 (cv_13998)	cv_13998	0.25	2.32	4076	2.7%	39.1%	18.6%	16.2%	9.7%	13.6%	0	0	12	0	0
> Inst_1071723 (cv_14030)	cv_14030	0.5	3.3	10914	0.4%	12.2%	15.4%	11.7%	16.4%	43.8%	912	2	8	204	0
> Inst_1075799 (cv_14081)	cv_14081	0.41	3.1	4001	0.2%	18.3%	10.7%	14.6%	21.2%	35.0%	128	0	0	72	0
> Inst_1077925 (cv_14087)	cv_14087	0.67	3.43	2067	0.2%	12.5%	15.1%	10.1%	14.1%	48.1%	256	0	0	18	0
> Inst_130 (cv_39)	cv_39	0.16	4.2	17216	2.6%	26.4%	22.0%	13.6%	13.1%	22.4%	60	0	0	4	0

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

Table 8: Rent Exponent Ranges

Range	Meaning
0.0 to 0.65	This range is low to normal.
0.65 to 0.85	This range is high, especially when the total number of instances is above 15,000.
Above 0.85	This range is very high, indicating that the design might fail during implementation if the number of instances is also high.

The following table shows the typical ranges for the Average Fanout.

Table 9: 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.
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.

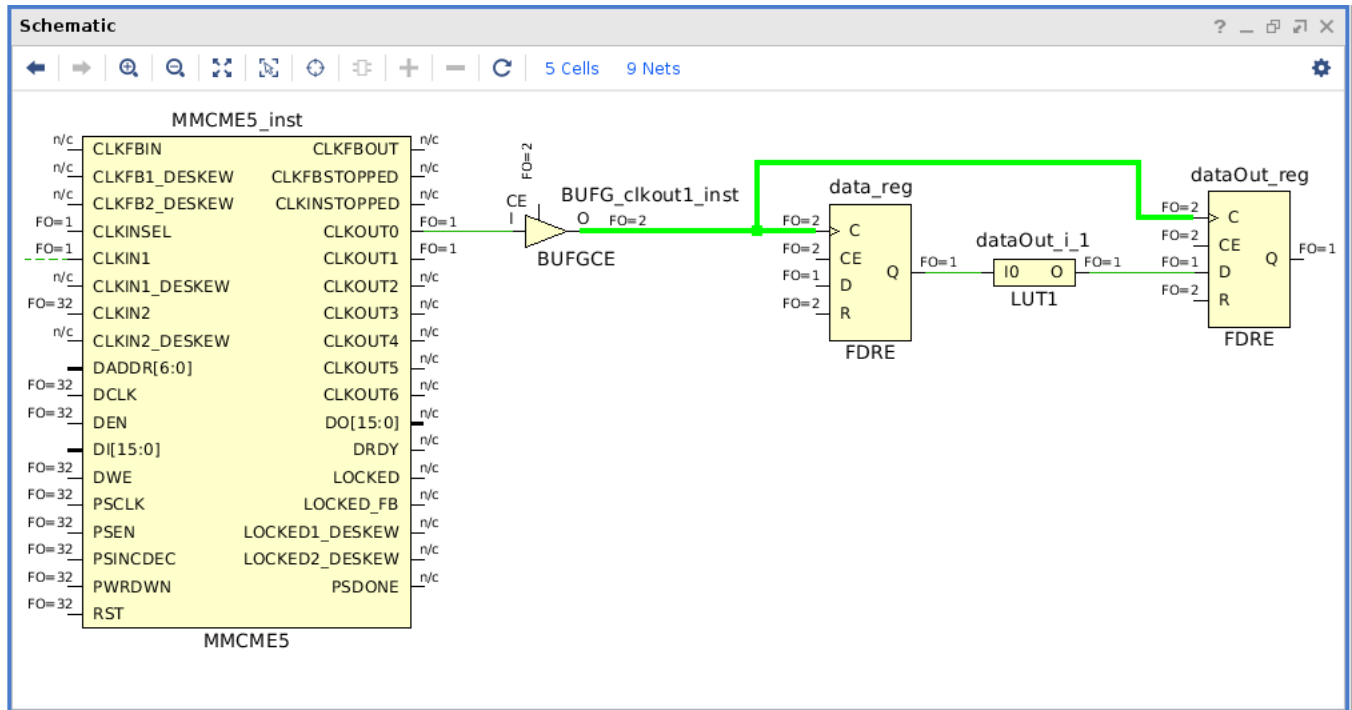
Note: For more information, see this [link](#) in the *Versal ACAP Hardware, IP, and Platform Development Methodology Guide* ([UG1387](#)).

2. Review the clock tree topologies and placement of timing paths impacted by higher clock skew than expected, as described in the following sections.
3. Identify the possible clock skew reduction techniques, as described in the following sections.

Using Intra-Clock Timing Paths

Timing paths with the same source and destination clocks that are driven by the same clock buffer typically exhibit very low skew. This is because the common node is located on the dedicated clock network, close to the leaf clock pins, as shown in the following figure.

Figure 15: Typical Synchronous Clocking Topology with Common Node Located on Green Net



When analyzing the clock path in the timing report, the delays before and after the common node are not provided separately because the common node only exists in the physical database of the design and not in the logical view. For this reason, you can see the common node in the Device window of the Vivado IDE when the Routing Resources are turned on but not in the Schematic window. The timing report only provides a summary of skew calculation with source clock delay, destination clock delay, and credit from clock pessimism removal (CPR) up to the common node.

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.

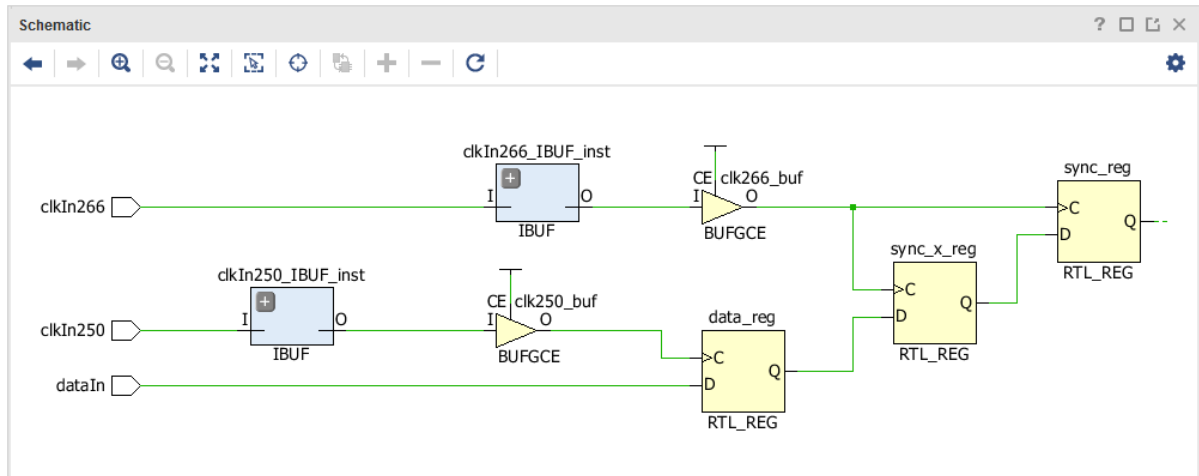
You must review all timing paths between asynchronous clocks to ensure the following:

- 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.

Note: For more information, see this [link](#) in the Versal ACAP Hardware, IP, and Platform Development Methodology Guide (UG1387).

Figure 16: Asynchronous CDC Paths with Proper CDC Circuitry and No Common Node



Applying Common Techniques for Reducing Clock Skew

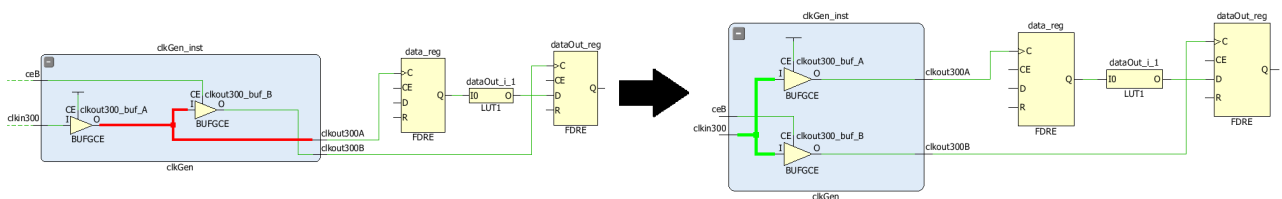


TIP: Given the flexibility of the Versal device clocking architecture, the `report_methodology` command contains checks to aid you in creating an optimal clocking topology.

The following techniques cover the most common scenarios:

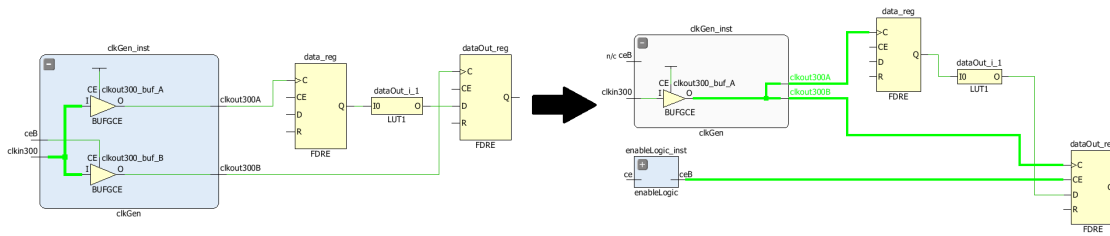
- Avoid timing paths between cascaded clock buffers by eliminating unnecessary buffers or connecting them in parallel as shown in the following figure.

Figure 17: 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.

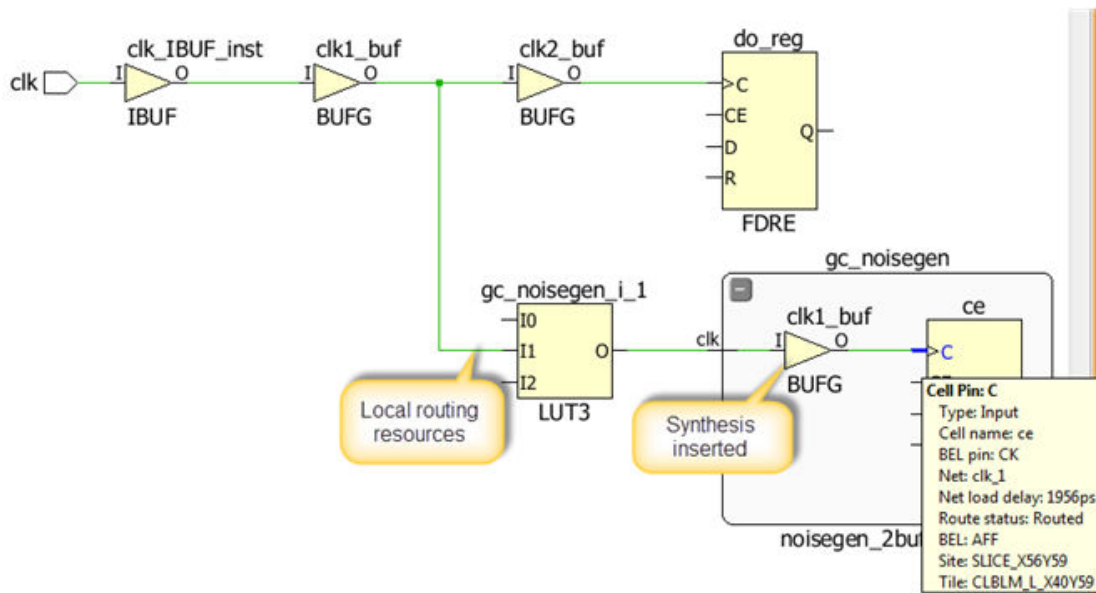
Figure 18: Synchronous Clocking Topology with Parallel Clock Buffer Recombined into a Single Buffer



- 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 19: Skew Due to Local Routing on Clock Network



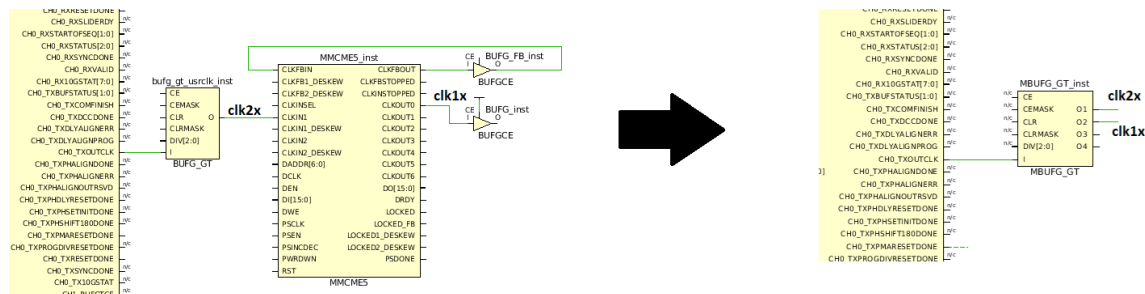
IMPORTANT! To learn best practices and verify that your design complies with the clocking guidelines, see this [link](#) in the Versal ACAP Hardware, IP, and Platform Development Methodology Guide ([UG1387](#)).

Improving Skew in Versal Devices

Following are general recommendations for reducing clock skew when working with the Versal architecture. For more information, see this [link](#) in the *Versal ACAP Hardware, IP, and Platform Development Methodology Guide* (UG1387).

- Avoid using an MMCM, XPLL, or DPLL to perform simple division of a BUFG_GT clock. BUFG_GT cells can divide down the input clock. When providing more than one simple division of the BUFG_GT clock to the fabric, the MBUFG_GT cells can divide down the clock using leaf-level division to minimize resource usage and improve QoR. The following figure shows how to save an MMCM resource and implement balanced clock trees for two clocks originating from a GT*_QUAD cell using the MBUFG_GT.

Figure 20: Implementing Balanced Clock Trees Using Versal MBUFG_GT



- If frequency synthesis of a GT clock is necessary, a DPLL exists in the clock regions that have GT*_QUAD resources.
- For timing paths between synchronous clocks, use the MBUFGCE, MBUFGCE_DIV, MBUFGCTRL, MBUFG_PS, and MBUFG_GT primitives to take advantage of leaf-level division, minimize resource usage, and improve timing QoR.
- When using parallel clock buffers, 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 a resource column or clock region. If this is the case, physical constraints such as Pblocks can be used to force the source and destination into a single clock region or to prevent the crossing of a resource column, such as a network on chip (NoC), 100G multirate Ethernet MAC (MRMAC), or integrated block for PCIe (Gen4 x16).
- Verify that clock nets with `CLOCK_DEDICATED_ROUTE=FALSE` constraint are routed with global clocking resources. Use `ANY_CMT_REGION` 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.

Reducing Clock Delay in Versal Devices

In 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 drives clock rows in each clock region via three levels of vertical distribution tracks creating a balanced H-tree to minimize clock skew in the vertical direction. The horizontal clock distribution is segmented at the clock region boundaries and there are programmable delays at the clock region boundaries that provide skew balancing in the horizontal direction. In addition to this static clock network deskew, Versal devices also offer calibration of the programmable delay values at device configuration to account for transistor versus interconnect delay differences, further minimizing clock skew.

The programmable delays are largest at the clock root and decrease from there to the edges of the clock network. For some clocking topologies it might be more important to reduce clock insertion delays rather than minimizing clock skew. For example, for synchronous CDC clocking paths where MBUFG cannot be used and parallel BUFG_GT or BUFGCE_DIV clock buffers drive the synchronous clocks, it can be important to minimize insertion delay to reduce minimum/maximum delay variation between the related clocks. In this case, a `CLOCK_DELAY_GROUP` property should be applied to the parallel clocks to match the clock routing and programmable delays should be disabled to minimize insertion delay by setting the `GCLK_DESKEW` property of the clock nets to OFF. Assigning the clock root next to the loads that require the minimal insertion delay using the `USER_CLOCK_ROOT` property further minimizes the skew between the synchronous CDC clocks.

Reducing Clock Uncertainty

Clock uncertainty is the amount of uncertainty relative to an ideal clock. Uncertainty can come from user-specified external clock uncertainty (`set_clock_uncertainty`), system jitter, or duty cycle distortion. Clock-modifying blocks such as the MMCM and PLL also contribute to clock uncertainty in the form of Discrete Jitter, and Phase Error if multiple related clocks are used.

The Clocking Wizard provides accurate uncertainty data for the specified device and can generate various MMCM clocking configurations for comparing different clock topologies. To achieve optimal results for the target architecture, Xilinx recommends regenerating clock generation logic using the Clocking Wizard rather than using legacy clock generation logic from prior architectures.

Using MMCM Settings to Reduce Clock Uncertainty

When configuring an MMCM for frequency synthesis, 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.

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.

Using MBUFGCE to Reduce Clock Uncertainty

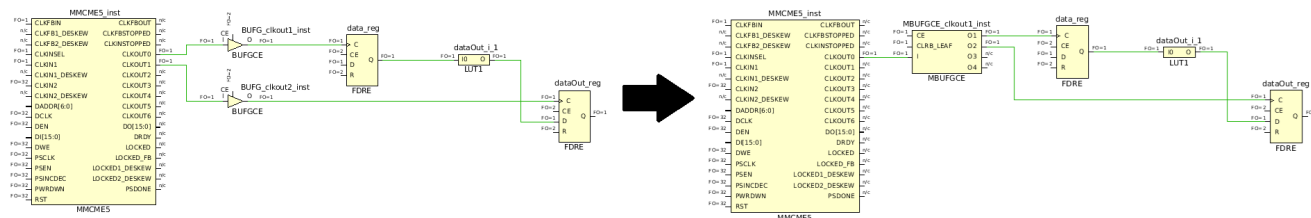
Note: This optimization technique is automatically applied by the `report_qor_suggestions` Tcl command.

In Versal devices, you can use MBUFGCE cells 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.

Note: The parallel BUFGCE_DIV topology recommended for UltraScale™ devices can be used in Versal devices. However, the parallel BUFGCE_DIV topology uses more power, uses more clocking resources, and incurs more skew versus using the leaf-level division of the MBUFGCE.

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, an MBUFGCE can be connected to the 300 MHz MMCM output and divide the clock by 2 at the leaf-level.

Figure 21: Improving the Clock Topology for a Versal ACAP Synchronous CDC Timing Path



Following are improvements 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 close to the leaf-level dividers, which reduces some clock pessimism.

The following tables compare the clock uncertainty for setup and hold analysis of a Versal ACAP synchronous CDC timing path.

Table 10: Comparison of Clock Uncertainty for Setup Analysis of a Versal ACAP Synchronous CDC Timing Path

Setup Analysis	MMCM Generated 150 MHz Clock		MBUFGCE 150 MHz Clock
	Clock Jitter (CJ)	0.405 ns	0.403 ns
	Phase Jitter (PJ)	0.000 ns	0.000 ns
	Phase Error (PE)	0.120 ns	0.000 ns
	Clock Uncertainty	0.322 ns	0.202 ns

Table 11: Comparison of Clock Uncertainty for Hold Analysis of a Versal ACAP Synchronous CDC Timing Path

Hold Analysis	MMCM Generated 150 MHz Clock		MBUFGCE 150 MHz Clock
	Clock Jitter (CJ)	0.402 ns	0.000 ns
	Phase Jitter (PJ)	0.000 ns	0.000 ns
	Phase Error (PE)	0.120 ns	0.000 ns
	Clock Uncertainty	0.322 ns	0.000 ns

Applying Common Timing Closure Techniques

The following techniques can help with design closure on challenging designs. Before attempting these techniques, ensure that the design is properly constrained and that you identify the main issue that affects the top violating paths.



RECOMMENDED: Xilinx recommends running the `report_qor_suggestions` Tcl command to identify and apply many of these techniques automatically.

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 benefit from the use of FF resources instead of SRLs to implement pipelining in the device, but the rest of the design might benefit from implementation of logic in SRLs rather than FFs to reduce spreading. In this case, set the `ALTERNATE_ROUTABILITY` strategy for the module that requires the use of FF resources, and synthesize the rest of the design using the Default strategy.

Note: For more information, see this [link](#) in the *Versal ACAP Hardware, IP, and Platform Development Methodology Guide* (UG1387).

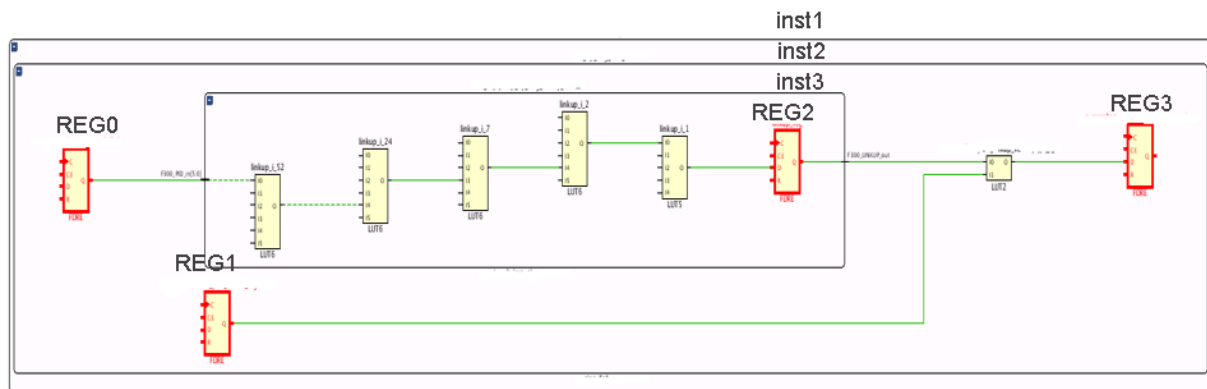
Improving Logic Levels

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 22: 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 23: 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 24: 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 Versal device half-slice 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 logic 1 clock enable can be provided for free from interconnect control multiplexers feeding the CE pins.

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.

Although the Versal device CLB contains four times the resources when compared to an UltraScale device, the control set comparison between a Versal device half-slice and UltraScale device half-CLB is similar. Therefore, Xilinx recommendations are the same for both architectures.

Note: For more information, see this [link](#) in the *Versal ACAP Hardware, IP, and Platform Development Methodology Guide (UG1387)*.

Follow Control Set Guidelines

The following table provides a guideline for the recommended number of control sets, depending on the target device size.

Table 12: 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
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

Determine the number of control sets in a design using `report_control_sets -verbose`.



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 `control_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.

Note: For additional synthesis attributes and recommendations on control signals, see this [link](#) in the Versal ACAP Hardware, IP, and Platform Development Methodology Guide (UG1387).

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.

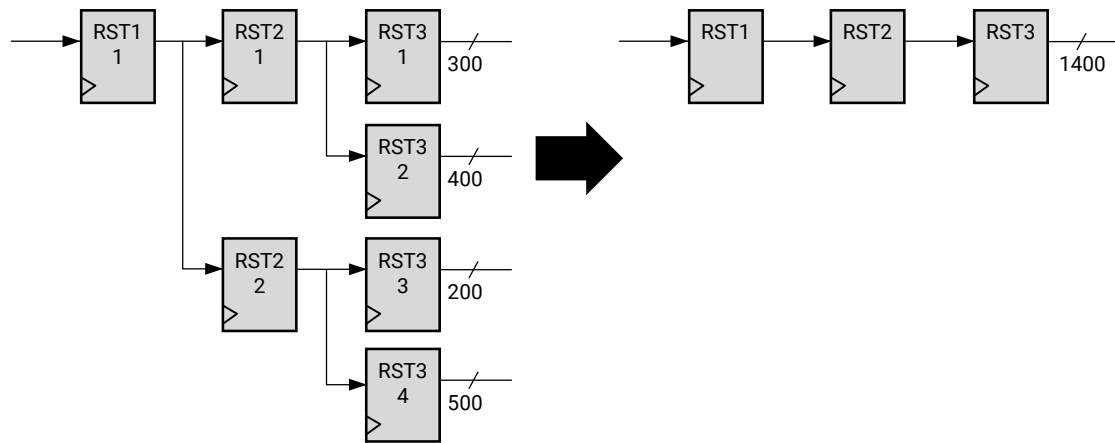
To restructure and reduce the number of control set trees and high fanout nets, you can use the `opt_design` Tcl command with one of the following options:

- **-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 25: Control Set Merging Using `opt_design -control_set_merge`



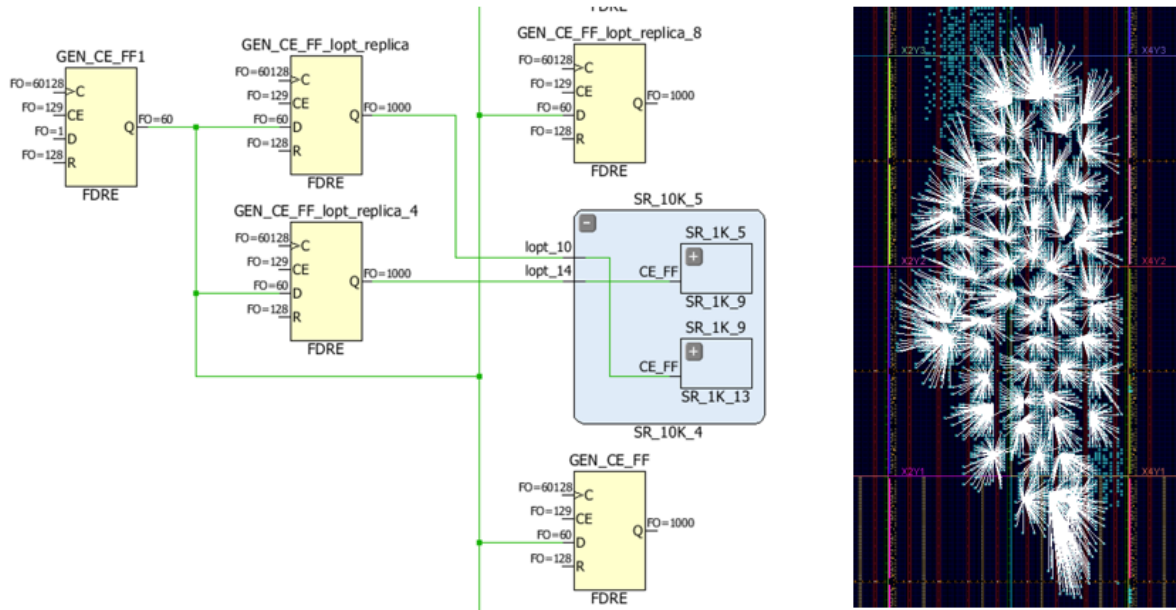
X20035-122019

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 26: 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).

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. To support high fanout nets on global routing in Versal devices, BUFG_FABRIC cells can be placed on BUFG_FABRIC sites that exist throughout the device in NoC columns.

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_FABRIC [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.

Note: Review the use of dedicated synthesis attributes to control local clock enable and set/reset optimizations as described in this [link](#) in the *Versal ACAP Hardware, IP, and Platform Development Methodology Guide* ([UG1387](#)).

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.

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.

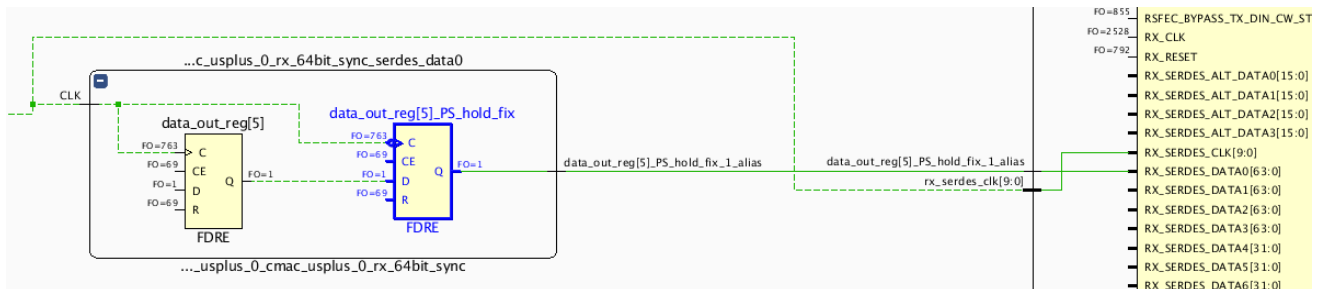
Fixing Large Hold Violations Prior to Routing

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 27: Fixing Hold Violation with Negative Edge Register Insertion



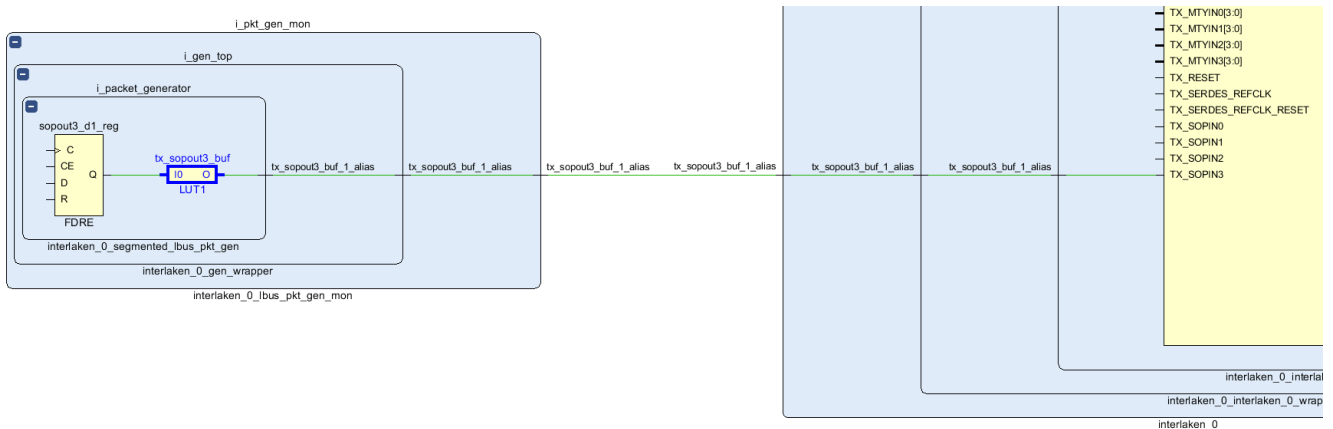
You can also insert LUT1 delays onto datapaths to reduce hold violations. To insert LUT1 delays, use one of the following options during the `phys_opt_design` implementation step:

- **-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 28: 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

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

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.

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.

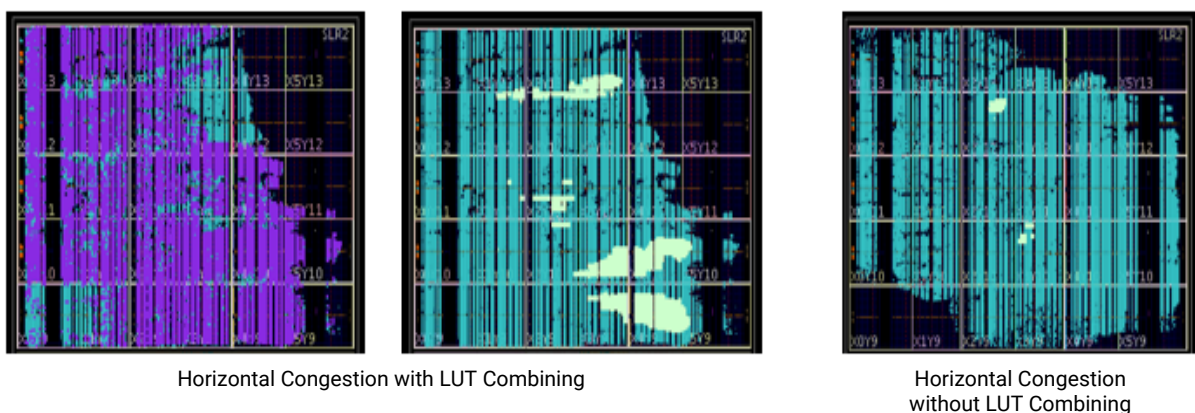
Note: If you are using Synplify Pro for synthesis, you can use the **Enable Advanced LUT Combining** option in the Implementation Options under the Device tab. This option is on by default. If you are modifying the Synplify Pro project file (*.prj), the following is specified: `set_option -enable_repacking 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 29: 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.

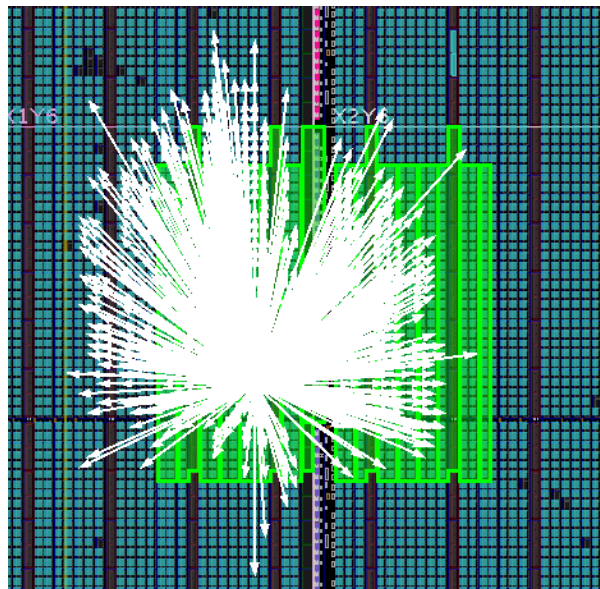
High fanout nets that have tight timing constraints require tightly clustered placement to meet timing. This can cause localized congestion as shown in the following figure. High fanout nets can also contribute to congestion by consuming routing resources that are no longer available for other nets in 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 30: 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_FABRIC` 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 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.

Predefined Strategies

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

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.



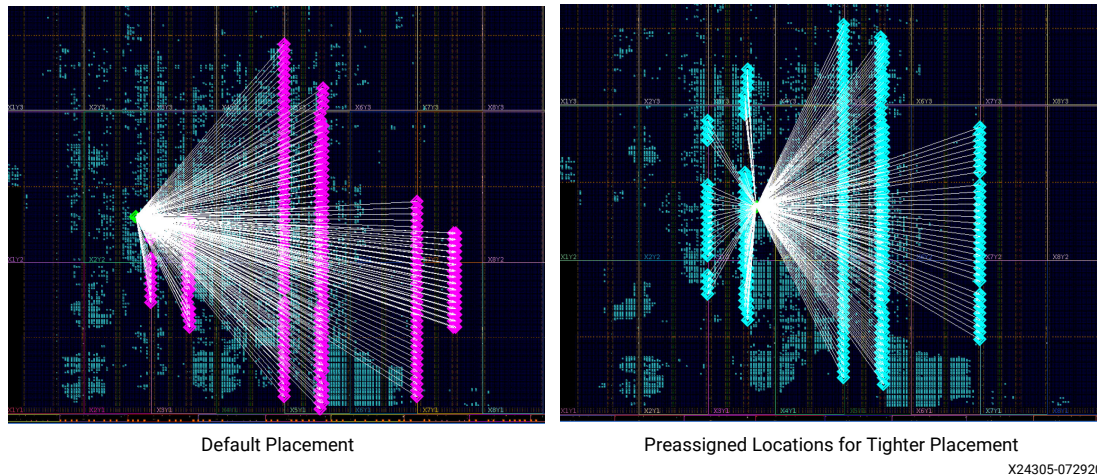
TIP: When upgrading to a newer version of the Vivado Design Suite, first try compiling without Pblocks or with minimal Pblocks (i.e., only SLR level Pblocks) to see if there are any timing closure challenges. Pblocks that previously helped to improve the QoR might prevent place and route from finding the best possible implementation in the newer version of the tools.

Grouping Critical Logic

Grouping critical logic to avoid crossing SLRs can help improve the critical path of a design. The following figure shows two examples of a large block RAM implemented with 512 RAMB36E5 primitives. The critical path is from a flip-flop to the ADDRADD* pin of every RAMB36E5 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. RAMB36E5 primitives are marked in pink.
- On the right, the example shows that the placer was able to meet timing, because the RAMB36E5 blocks were grouped together. RAMB36E5 primitives are marked in light blue.

Figure 31: Critical Logic Grouping



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.

```
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.

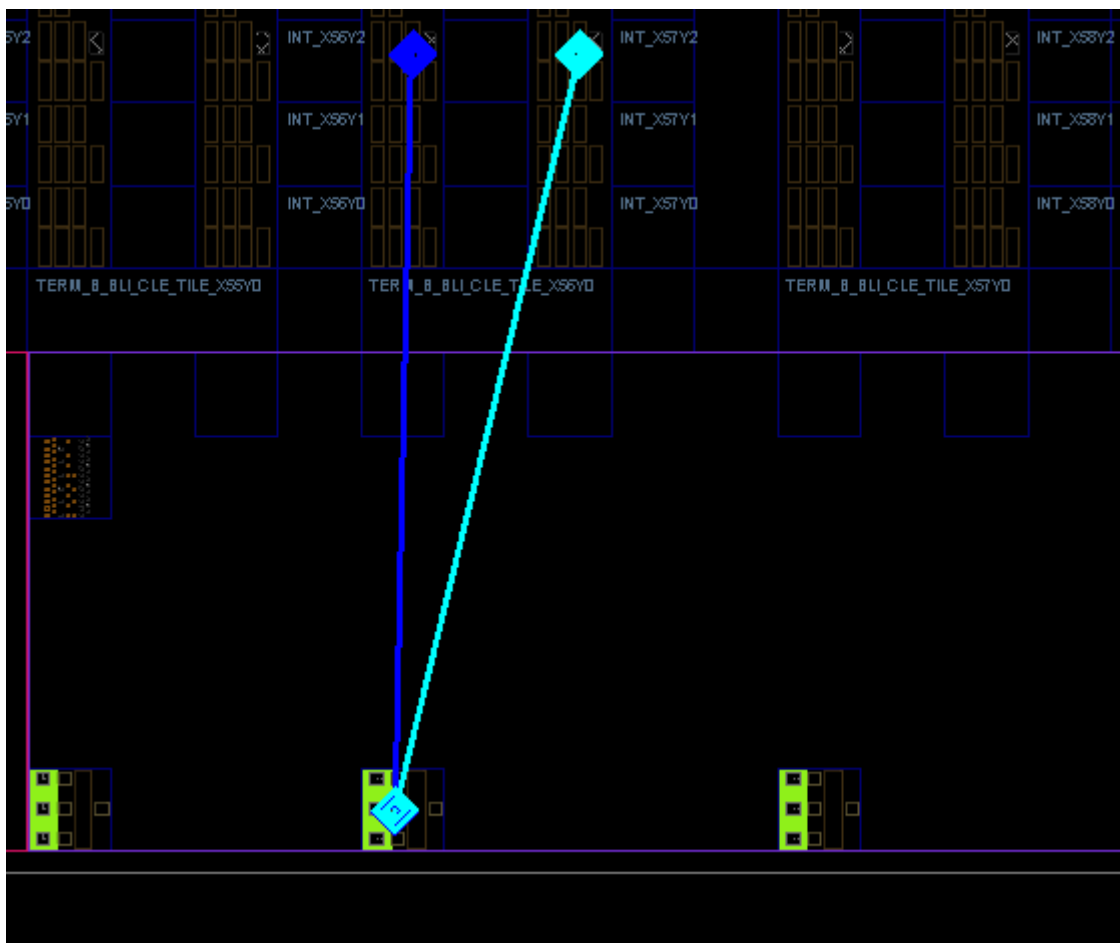
XPIO-PL Interface Techniques for Timing

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 {idder_Q1-BLI-reg idder_Q2-BLI-reg}]
```

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

Figure 32: Placement of XPIO-PL Interface BLI Flip-Flops for ODDRE1 and IDDRE1



AI Engine-PL Interface Techniques for Timing

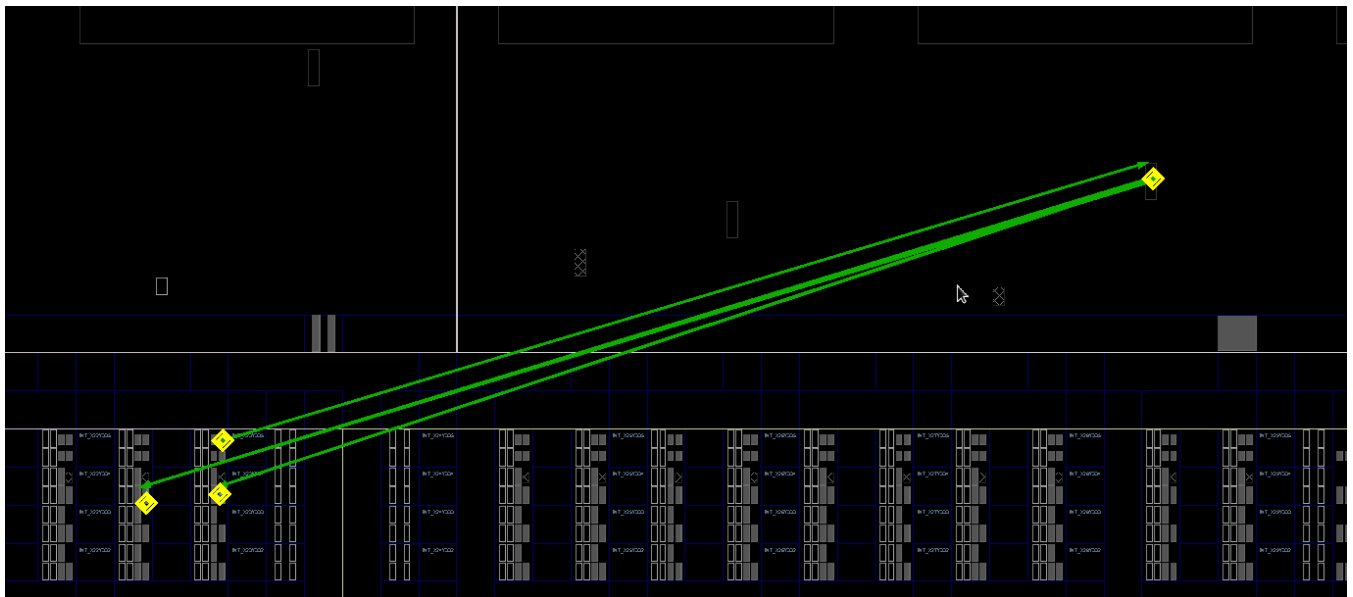
Boundary logic interface flip-flops exist in hardware between the AI Engine-programmable logic (PL) interface, which you can use to improve timing. You can apply boundary logic interface (BLI) constraints to flip-flops in your design to automatically take advantage of this hardware feature during design placement. In this example, the AXI4-Stream interface for the AI Engine (AXI_PL_M_AXIS64) has connections to and from the fabric driven by flip-flops. Following is an example of the BLI constraints:

```
set_property BLI TRUE [get_cells m_axis_tdata_reg[*]]
```

```
set_property BLI TRUE [get_cells m_axis_*_reg]
```

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

Figure 33: Placement of AI Engine-PL Interface BLI Flip-Flops



For timing critical designs, enabling the BLI registers helps to achieve the highest performance. To control the inference of BLI registers across the AI Engine-PL channels, use the following AI Engine compiler options:

- `--pl-freq=<number>`

Specify clock frequency for PL kernels in MHz. The default value is 1/4 of AI Engine core frequency, which varies for each speed grade.

Following are examples:

- Same AI Engine-PL 300 MHz frequency for all AI Engine-PL interfaces:

```
--pl-freq=300
```

- Different AI Engine-PL frequency for different interfaces. Each interface is associated to a different AI Engine graph PLIO. The constraints must refer to the PLIO name, not the PL kernel names:

```
--pl-freq=plio_user_port_0:153 -pl-freq= plio_user_port_0:307.2
```

- `--pl-register-threshold=<number>`

Specify frequency threshold for registered AI Engine-PL crossing in MHz. The default value is 1/8 of the AI Engine frequency based on speed grade.

Following is an example:

- `-pl-register-threshold=125`

Any PLIO with an AI Engine-PL frequency higher than this setting (125 MHz in this case) is mapped to high-speed channels with the BLI registers enabled. If not, any AI Engine-PL channel can be used.

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](#)).

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.

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.

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 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, ..}
```

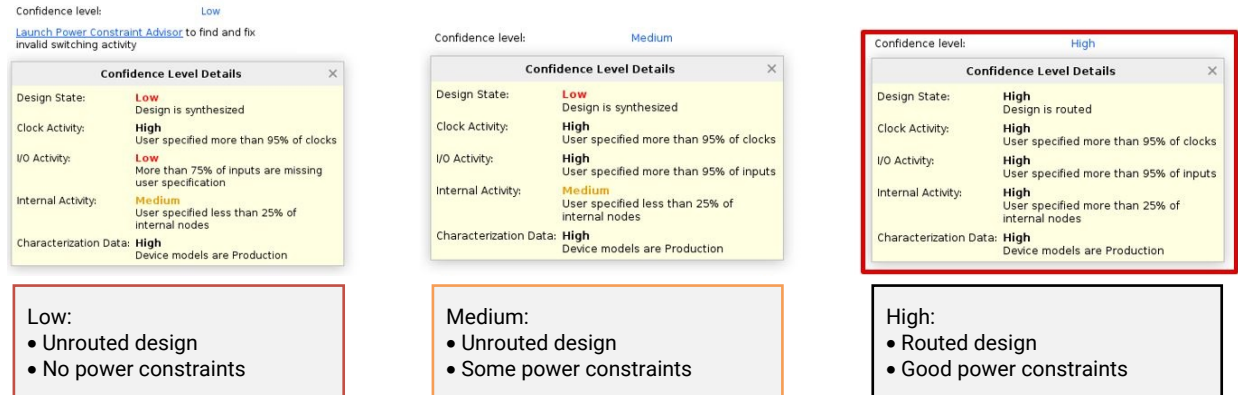
Defining Current Budget

```
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 34: Power Analysis Confidence Level



X25127-021621

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.

Power Timing Slack

When closing a design for timing it is much more efficient and effective to also close a design from a power perspective simultaneously as this allows for the best run selection that satisfies both criteria. This can be done by simply adding the report power constraint to the script being run.



POWER TIP: For more details and an example script to implement this, see Xilinx [Answer Record 76056](#).

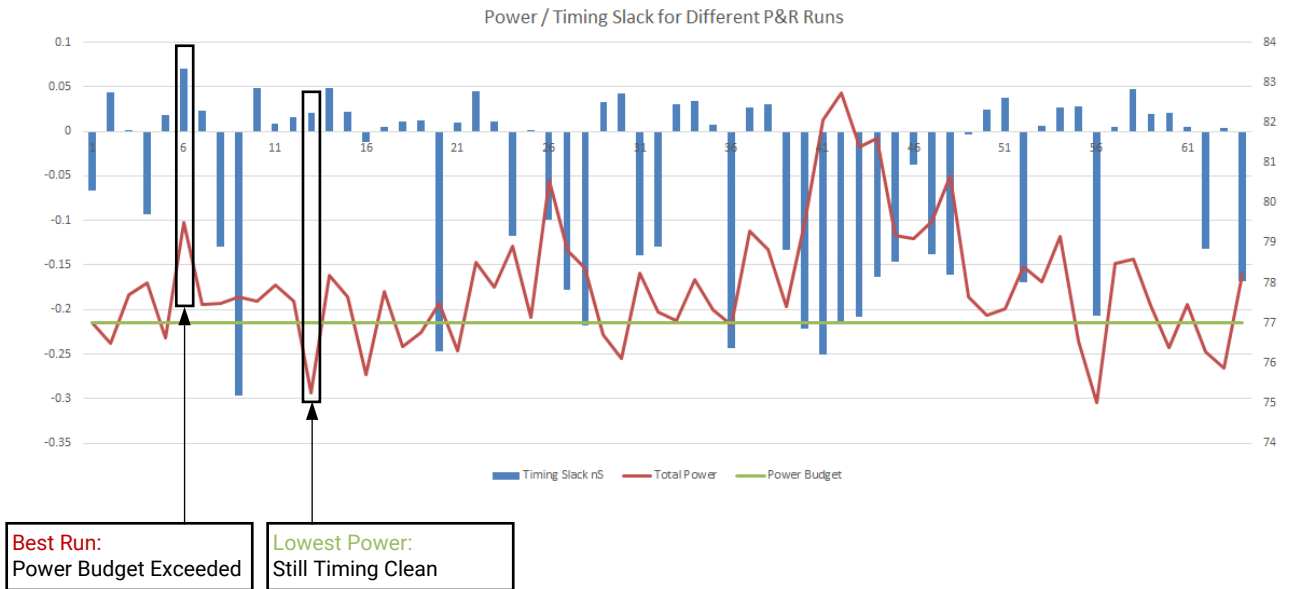
The following figure shows an example of how powerful this can be, for all 64-timing closure runs report power was also run, and these were all plotted together.

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.

So, understanding the design from both a timing and power perspective allows for the best run for both to be selected, without impacting the timing result and in this example enabled a 4W power saving.

Figure 35: Power and Timing Slack for Different Place and Route Runs



X25400-060421

System Performance Closure

Versal™ devices are built around heterogeneous compute engines connected to each other by the NoC or the PL, and connected to the external system through high-performance transceivers and I/Os. During the system application planning and mapping phases, the device interfaces and overall compute requirements are used to specify the target performance of each compute and control function implemented in the device. Each function is designed and mapped to the best suitable or available hardware resource using the corresponding programming language and compilation software (e.g., system software for the embedded processor system, C/C++ for AI Engines or PL kernels, RTL for higher performance PL kernels or firmware, etc.).

Individual design teams must validate both functionality and expected performance at the function level prior to integrating in a subset of the system application or the complete system. During the integration phase, functionality can break and performance can degrade. Due to the complexity and heterogeneous nature of the system applications supported by the Versal™ devices, the analysis and debug methodology must be understood and planned ahead of time. The Vitis™ and Vivado® tools are comprehensive and complementary design environments, which provide all necessary features to simulate functionality, report design characteristics, and measure or probe data in hardware.

The following sections recommend step-by-step analysis methods to identify application bottlenecks, identify performance mismatches around one or several functions, and address common performance issues based on the targeted device resource.

Analyzing System Performance for Platform-Based Designs

You can analyze system performance in simulation and in hardware as described in the following sections.

Analyzing System Performance in Simulation

When analyzing system performance in simulation, the critical data paths are between the AI Engines, PL, NoC (DDR), and high-speed transceivers. Before the system is integrated, you must ensure that each block meets its performance estimates.

Note: For traditional designs without a platform for use in the Vitis tools, system performance analysis is based on RTL simulation and hardware debug features, such as ChipScope™ debug IP cores and dedicated hard block debug features.

Analyzing AI Engine Performance in Simulation

After the application architecture is finalized and the design partitioning between the AI Engine and PL is complete, the next step is to develop the AI Engine application. You can use the `aiesimulator` to measure application performance using trace profiling features. You can also measure the performance using the `aiesimulator` output.

Following is an example in which performance is measured as (end-time - start-time)/number of samples where:

- Each line represents a 64b number (2 cint16s). There are 51200 64b numbers (for example, 102400 32b samples).
- Throughput = $102400 / (182452500 \text{ ps} - 5790 \text{ ns}) \text{ Samples/s} = 579.636 \text{ MSps}$.

```

1 T 5790 ns
2 0 0 0 0
3 T 5792500 ps
4 0 0 0 0
5 T 5795 ns
6 0 0 0 0
7 T 5797500 ps
8 0 0 0 0
9 T 5800 ns
...
...
...
102495 -5107 2007 -32768 -18047
102496 T 182450 ns
102497 -25374 -19023 3957 3067
102498 T 182452500 ps
102499 TLAST
102500 18230 14818 11355 -5427

```

To further analyze AI Engine performance bottlenecks, Xilinx recommends running the `aiesimulator` or hardware emulation with AI Engine trace and profile options. You can open the run summary file generated for the simulation run which includes the trace and profile data in the Vitis Analyzer. This generates trace and profile views which helps you identify performance root causes. For more information, see [Performance Analysis of AI Engine Graph Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

In addition, you can obtain detailed profiling data on AI Engine graph bandwidth, throughput, and latency using the AI Engine Run-Time Event API. For more information, see [Run-Time Event API for Performance Profiling](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

Analyzing PL Kernel Performance in Simulation

HLS

All kernels developed by HLS can be optimized using compiler directives and HLS pragmas. The Vitis HLS compiler generates detailed reports containing Fmax, resource utilization, and performance information. In addition to the summary reports, the schedule viewer provides a visual representation of how the design is built and how the operations are scheduled. You can use this view to help identify suboptimal portions of the synthesized design.

You can supplement these compile-time reports by running the HLS cosimulation flow. When you run this flow, Vitis HLS automatically extracts performance data from the simulation results and reports additional performance information such as minimum, maximum, and average running times for FIFO high watermarks. Xilinx recommends using all of these analysis capabilities before integrating the HLS kernel in the system.

Note: A kernel that does not meet performance in a standalone context will not meet performance in the complete system.

Many factors influence the performance of an HLS kernel, including interface properties, loop-level parallelism, task-level parallelism, and more. In particular, understanding the concepts of initiation interval (II) and dataflow are essential to achieve good results. Initiation interval is measured in clock cycles and indicates how often a particular loop or process restarts. For example, if a loop is successfully synthesized with II=1, then in the resulting RTL a new loop iteration starts every cycle. II is closely related to throughput, a key performance metric. Dataflow is a performance optimization that takes advantage of task-level parallelism. Whenever possible, dataflow allows different sub-processes in the design to run concurrently instead of sequentially. Achieving optimal results with dataflow requires a suitable code structure. For more information about initiation interval, dataflow, and other HLS performance optimizations, see the *Vitis High-Level Synthesis User Guide* ([UG1399](#)).

RTL

All RTL kernels you develop must be simulated at the block level, either using custom RTL test benches or using the Xilinx® LogiCORE™ AXI Verification IP (VIP) provided in the Vivado® IP Catalog. For more information, see the *AXI Verification IP LogiCORE IP Product Guide* ([PG267](#)).



TIP: Additional performance counters can be written in RTL to count cycles in the PL and calculate latency and throughput to/from the AI Engines.

Using Hardware Emulation to Analyze System Performance

Versal ACAP systems that include AI Engines, PS, and PL can be integrated using the Vitis linker and then simulated together using Vitis hardware emulation. Hardware emulation allows you to observe and measure the combined effects of AI Engines, PS, and PL interactions on system performance. In hardware emulation, the PL kernels run as RTL, the AI Engine kernels run in the aiesimulator, and the PS code runs in the Xilinx Quick Emulator (QEMU). Some infrastructure blocks are abstracted using transaction level models (TLM) for simulation speed purposes. Hardware emulation is nearly but not fully cycle accurate and provides a valuable representation to analyze, debug, and validate major system performance considerations prior to implementation.

Hardware emulation runs automatically and generates various performance-related reports based on user settings, such as the profile summary and the application timeline trace. You can view these reports in the Vitis Analyzer for useful insights into performance, such as data transfer size and efficiency, kernel run times, stall information, and more. In addition to these reports, Vitis Analyzer provides detailed activity waveforms, allowing you to conduct custom, fine-grained analysis of specific parts of the system.

For more information on hardware emulation and Vitis Analyzer, see the *Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#)) and *Versal ACAP AI Engine Programming Environment User Guide* ([UG1076](#)).

Additional Hardware

For a synthesizable way of measuring throughput, you can design RTL IP to count cycles elapsed for a particular transaction (e.g., time taken to transmit a payload of "B" beats of data from source to destination). This can be one by triggering a counter to start events from first TVALID to TLAST. Alternatively, you can leverage AXI Performance Monitor (APM) IP provided in the Vivado IP Catalog to count the events.

Software

A lightweight PS application must be developed to run the entire system (e.g., trigger PL IP, start traffic, etc.), which is application dependent. For example, some applications might utilize on PL reset deassertion and do not require any PS application. In this case, a simple PS application to print Hello World can be used to trigger the system emulation.

Baremetal applications are sufficient. Linux based hardware emulation is also possible but does not provide additional benefits at this stage.

Analyzing System Performance in Hardware

For system comprised of AI Engines, PL, and PS, ensure your design follows these prerequisites to analyze performance:

- Hardware
 - Use PL IP, such as APM IP, RTL/HLS kernels, to calculate throughput latency in the system to measure cycle-accurate performance.
- Software
 - Use the Linux operating system (OS) to use the profiling APIs provided by aiecompiler, available as part of the software platform. For more information on enabling Linux and building collaterals as well as details on the profiling APIs, see the *Versal ACAP AI Engine Programming Environment User Guide* ([UG1076](#)).
 - Use a PS application to coordinate the entire system that runs on Arm® Cortex®-A72 (e.g., start data traffic from PL/DDR, start counters, etc.). The PS application needs to be cross-compiled against the G++/GCC compiler for the Cortex-A72. The PS application can also be used to read out RTL/HLS/APM counters to measure throughput and latency.
 - After the design is implemented on silicon, use the Linux OS to take advantage of profiling application programming interfaces (APIs) provided by aiecompiler.

Measuring Performance with RTL or HLS Monitors

You can use custom RTL or HLS kernels to count the cycles elapsed between the start and end of AXI4-Stream transactions. In the PS application, you can read/write these counters to measure performance at run time.

Measuring AI Engine Performance

Measuring Performance with AI Engine Run Time Event APIs

After the graph is compiled using the Vitis tools or aiecompiler, each AI Engine array interface (or shim tile) can be monitored to count for specific events. You can use a few profiling events to count valid AXI4-Stream data transactions within the AI Engine array interface. When the APIs are called, the PS issues a sequence of AXI4-MM commands to configure the AI Engine array interface to count for valid events. The event counters in the AI Engine array interface provide a helpful way to measure the system without adding any additional hardware to the system.

Note: Each AI Engine array interface has only two performance counters, but there are fourteen 64b streams in each AI Engine array interface. Therefore, only two AI Engine-PL interfaces can be monitored at one time using these probing APIs.

The following example uses the `io_stream_start_to_bytes_transferred_cycles` event API to measure the throughput of the graph. This API uses two performance counters to track both the bytes transferred and cycles taken. This event API captures and calculates the sum of the total active, stall, and idle cycles that transfer the specified amount of data through the graph. This API can be used on both input and output streams.

```
gr.init();
event::handle handle = event::start_profiling(plio_out,
event::io_stream_start_to_bytes_transferred_cycles, 256*sizeof(int32));
gr.run(8);
gr.wait();
long long cycle_count = event::read_profiling(handle);
event::stop_profiling(handle);
double throughput = (double)256 * sizeof(int32) / (cycle_count * 1e-9); //
byte per second
```

You can use an alternative event API when the number of bytes being transferred is unknown. The following example uses the `io_stream_running_event_count` event API to measure the throughput of the graph. The streams run for a specific interval of time, and the number of stream active events is captured.

```
...
...
using namespace adf;
event::handle handle_0;
PLIO duc_plio[2] = {*duc_in0, *duc_out0};
d=0;
while(d < NUM_DUC_SLAVES) {
    long long throughput_out_min = 990000000; // initial value to some high
number
    long long throughput_out_max = 0;
    int iter=0;
    while(iter < 5) {
        long long count_start, count_end;
        long long throughput;
        handle_0 = event::start_profiling(duc_plio[d],
event::io_stream_running_event_count);
        count_start = event::read_profiling(handle_0);
        //precision of usleep is dependent on linux system call
        usleep(1000000); //1s
        count_end = event::read_profiling(handle_0);
        event::stop_profiling(handle_0);
        if (count_end > count_start) throughput = (count_end-count_start);
        else throughput = (count_end-count_start+0x100000000); //roll over
correction for 32b performance counter
        if (throughput<throughput_out_min) throughput_out_min = throughput;
        if (throughput>throughput_out_max) throughput_out_max = throughput;
        iter++;
    }
    printf("[throughput] %d\tMin:%llu\tMax:%llu\tRange:%llu\n", d,
throughput_out_min, throughput_out_max, throughput_out_max-
throughput_out_min );
    d++;
}
printf("[main] Performance measurements Done ... \n");
...
...
```


For information, see [Run-Time Event API for Performance Profiling](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

Measuring Performance with Event Trace Debug

For information, see [Performance Analysis of AI Engine Graph Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

Debugging AI Engine-PL Performance on Silicon

Following are recommended methods of debugging AI Engine-PL performance:

- Break the AI Engine graph into smaller graphs to analyze bottlenecks on silicon. For example:
 - If the graph has kernels in the AI Engine and PL, compartmentalize the graph into sub-graphs to verify functionality and performance. With this method, you can localize performance bottlenecks.
 - If compute kernels (either in the AI Engine or PL) receive data from multiple AXI4 streams, the kernels might underperform due to the variable time of data arrival on different streams. This can happen due to either backpressure or to the different compute complexities of previous kernels in the graph. The graph can be broken down at the kernel level to verify that all the streams perform optimally.

Note: Alternatively, you can analyze bottlenecks using kernel-level performance measurement and debug.

- Replace the AI Engine graph with a simple pass-through system.
- Use the event trace debug feature to count memory stalls in different kernels. For more information, see this [link](#) in the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

Improving Performance in the PS

In the Versal ACAP processing system (PS), you can control quality of service (QoS) using the following interconnect traffic types: low-latency datapaths and high-throughput datapaths. For information, see this [link](#) in the *Versal ACAP Technical Reference Manual* (AM011).

When thinking of the processing system and memory subsystems performance, the first thoughts that come to mind are about the processors performances, their clock frequencies, memory hierarchy, caches, and more generally internal and external memory capabilities. While these considerations are important, understanding the processing system architecture is equally so.

The external shared DDR memory subsystem and the on-chip OCM are the main two memory subsystems present in Versal devices. The masters accessing the memory subsystems can be in PL or in PS. The PL master can greatly impact PS masters performances when traversing the PS to access OCM or external DDR memory.

The PL masters traffic routes are:

- PL master (soft IP) attached to NoC via a NoC NMU
- PL master attached to PS AXI interfaces
- PL master attached to PS ACP

The AI Engine and CPM can also require to route traffic via the PS interconnect and close attention must be paid to the multiple ways traffic can be routed.

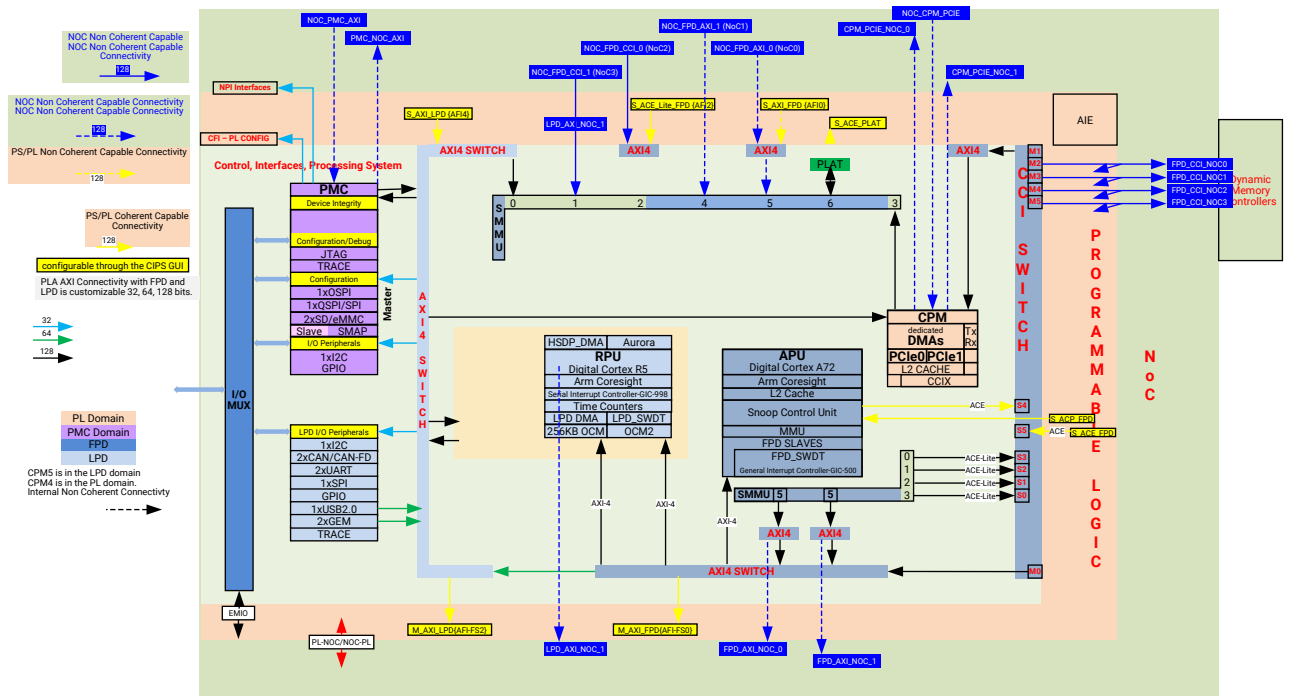
The internal PS master (APU, RPU, DMA, etc.) generates traffic and leverages the PS interconnect that connects to the NoC.

- Direct PL to PS interfaces (S_AXI_FPD, S_ACE_LITE_FPD, S_AXI_LPD)
 - S_AXI_FPD (in FPD) is virtualized and non-coherent
 - S_ACE_LITE_FPD (in FPD) is virtualized and coherent
 - S_AXI_LPD (in LPD) can be configured as physical, or as virtualized and coherent
- NoC to PS interfaces (NoC_FPD_AXI0, NoC_FPD_AXI_1, NoC_FPD_CCI_0, NoC_FPD_CCI_1, all in FPD)
 - NoC_FPD_AXI0 and NoC_FPD_AXI1 are virtualized and non-coherent
 - NoC_FPD_CCI_0 and NoC_FPD_CCI_1 are virtualized and coherent

Understanding and leveraging the multiple ways a master can reach the external DDR memory or OCM is critical to optimized the routing and avoiding PS interconnect congestion.

For example, selecting to route the traffic of multiple masters through the CCI can be detrimental to performances, as a maximum of four CCI500 AXI4 master ports connect to four NoC NMUs. In the following figure these are M2, M3, M4, and M5.

Figure 36: Master Ports Connected to NoC NMUs



X25052-071921

The SMMU and CCI can also impact performance. Each have share internal resources that may contribute to adding latency and therefore reducing throughput. If you are optimizing for high performance and if virtualization/isolation and coherency are not required, using the CCI500 and SMMU are not recommended.

Traffic regulation mechanisms are available at the source, which is the NMU in the NoC switches, and at the destination slaves (MC). You can use these mechanisms to apply the desired QoS scheme.

The PS interconnect does not support virtual channels, so physical separation must be used.

In the ingress direction (external masters to PS slaves), each NSU supports a single traffic class, so different classes must enter the PS by different physical ports. Similarly, in the egress direction, the internal PS interconnect network will carry different traffic classes over different physical channels into NMUs that connect to the horizontal NoC. For example, two of the four CCI to NoC channels can carry LL (M2 and M3 in the above figure), while the other two may carry BE (M4 and M5 above).

Improving Performance in the PL

The following design characteristics define PL performance:

- Maximum clock frequency achieved by the Vivado implementation tools
- Latency through the various RTL blocks, either due to finite state machines (FSMs) or amount of pipelining
- NoC efficiency given the achieved QoS and traffic pattern of the connected blocks
- Power consumption given the thermal setup, design toggle rate, resource utilization, operating clock frequency, and target speed grade

To improve maximum PL clock frequency when meeting your power budget, do the following:

- Use traditional analysis and closure techniques
- Analyze the data movement inside the NoC
- Validate that the measured bandwidth and traffic pattern in hardware matches the simulated results generated during design creation

Related Information

[Design Closure](#)

[Improving Performance Through the NoC](#)

Reducing Latency

The design latency usually depends on the amount of pipelining present in the RTL, which is traditionally used to improve the design maximum clock frequency. Following are the various types of pipelining:

- Pipelines required for operating special primitives, such as DSP, block RAM, or UltraRAM, at the maximum frequency listed on the device data sheet.

Although these registers are important for high clock frequency designs, not all registers are needed for slower designs and can be removed to reduce latency.

- Pipelines required for reducing the maximum number of logic levels or route levels on the longest paths in the design given the target frequency.

These pipelines are usually mapped to SLICE registers. When the register utilization percentage exceeds 50% at the device level or at the SLR level for SSI devices, the logic placement can become more difficult to legalize and Fmax can degrade. Also, if the overall latency through an RTL module or design is high, you must reduce the register utilization on paths with zero or one logic level, especially if these paths are locally placed and routed.

- Pipelines required for balancing latency with some other paths.

Versal devices have an even distribution of SRL cells that you must use as much as possible by default. The Vivado implementation tools support several physical optimization that pull registers out of SRLs whenever needed to help meet timing.

- Pipelines introduced by the Vitis HLS tools to improve logic levels and maximize the chance of closing timing at the predicted target frequency.

You can control the maximum latency via QoS constraints for specific functions by setting attributes in C/C++. You can also reduce the target frequency for Vitis HLS and perform a pre-placement timing analysis to verify that the design can meet timing based on ideal placement and logic level information. For more information on the Vitis HLS tools, see the *Vitis High-Level Synthesis User Guide* ([UG1399](#)).

In addition to these types of pipeline registers, most Xilinx or third-party IP provide interface register, latency, or target frequency options. You must refine all available settings to meet the right latency and Fmax trade-off by reviewing the guidelines provided in each IP product guide. In addition, you can synthesize and implement each IP standalone to validate that timing closure can be achieved, preferably with 5 to 15% Fmax margin.

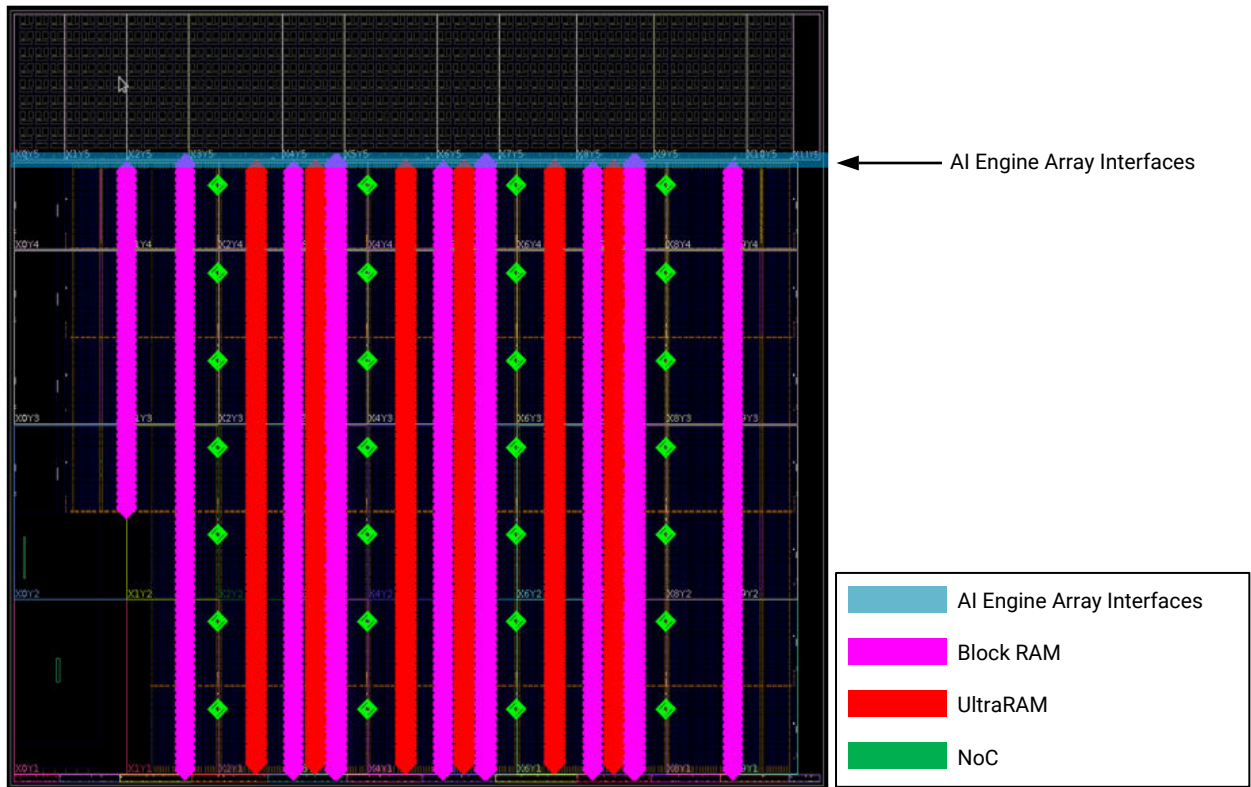
Addressing Floorplanning Impact on Performance

Before you begin floorplanning, ensure that key resources are available on your Versal device. These resources can include the AI Engine array interface (or shim tile), block RAM, UltraRAM, DSP, NoC, and DDRMC.

The following figure shows the XCVC1902 in the Vivado IDE Device window with key resources highlighted. The figure also shows how the fabric memory resources are distributed across the device, such as block RAM or UltraRAM. Aligning the floorplan based on the connectivity of the input/output stream of logic is crucial to achieving better system performance. Following are some examples for input and output stream of logic:

- AI Engine-PL-NoC-DDRMC
- AI Engine-PL
- PS-NoC-DDRMC

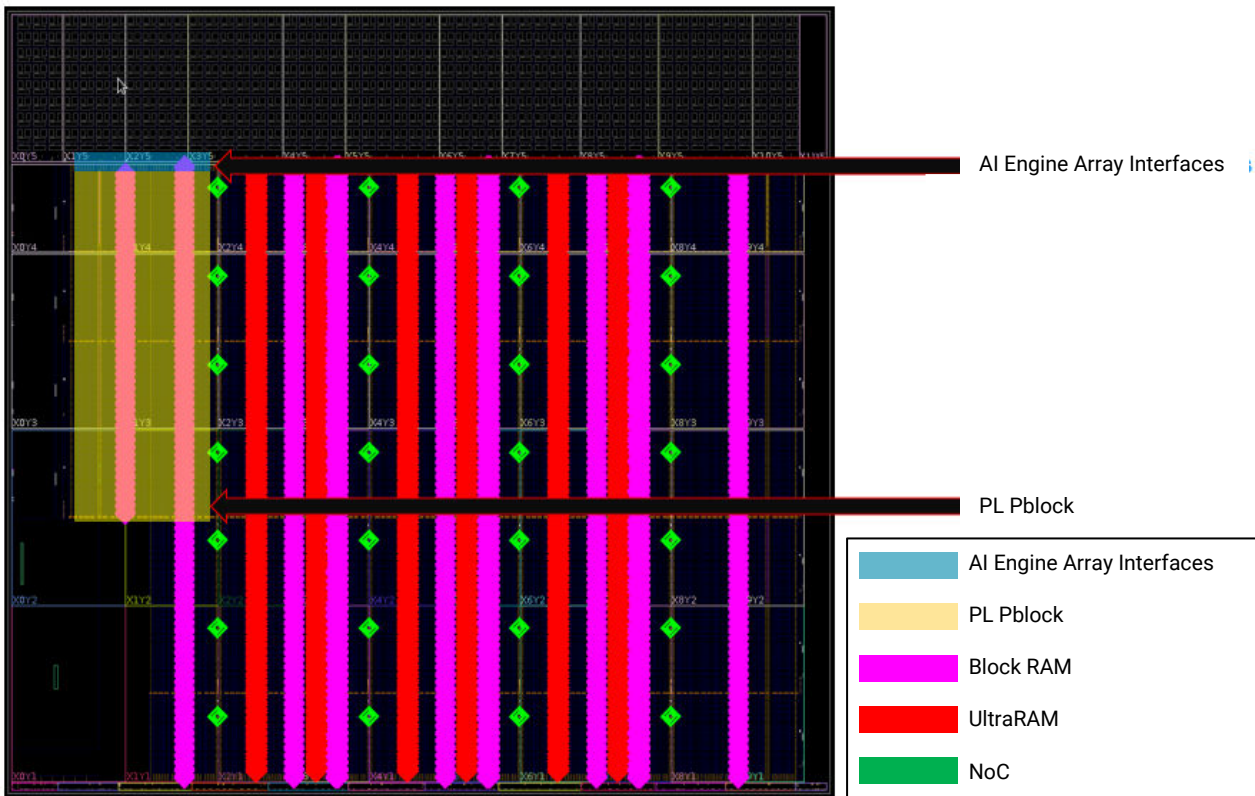
Figure 37: Versal ACAP Key Resources



X24301-081320

The following figure shows an example of floorplanning the AI Engine and PL. In this design, there is AI Engine-PL dataflow, and the PL does not use UltraRAM resources. If only block RAM resources are used as part of the dataflow interacting with the AI Engine array interface, the AI Engine streams can be floorplanned in a region where the PL is mapped to a portion of the device without UltraRAM columns.

Figure 38: AI Engine and PL Floorplanning

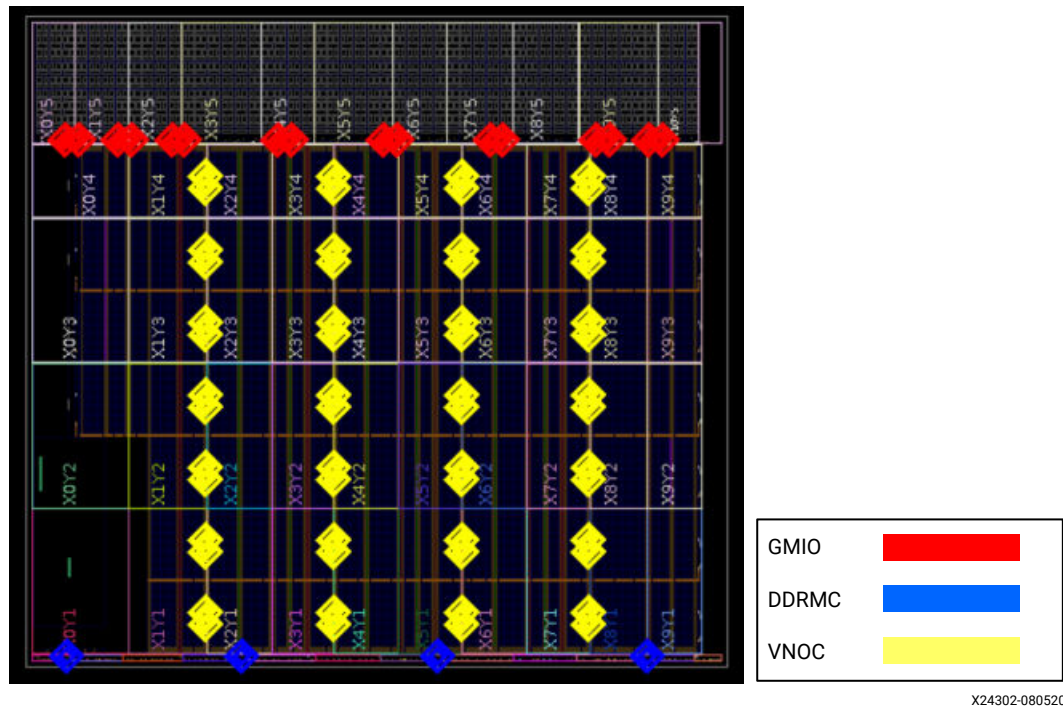


X24303-081320

The following figure addresses designs with GMIOs in the AI Engine that read/write to the DDRMC through the NoC. There are specific columns in the AI Engine that support GMIOs. The following figure shows an XCVC1902 in the Device window with 16 GMIO-capable columns. Following are considerations for accessing GMIO-capable columns:

- Supply proper QoS requirements, allowing the NoC compiler to maximize bandwidth allocated to the paths.
- Allow the aiecompiler to select GMIO columns and the NoC compiler to select the appropriate DDRMC location based on QoS settings.
- If needed, constrain GMIOs to appropriate columns above the vertical NoCs (VNoCs), and also constrain DDRMC below the VNoCs to minimize latency through the NoC.

Figure 39: GMIO-Capable Columns



Following is an example of how to floorplan a design with AI Engine and PL kernels:

- AI Engine array interface Pblock (JSON file)

Determine the width for the DDRMCs and create a `<constraints>.json` file to be passed on to the aiecompiler.

- Pblock (XDC file)

Determine the size of the VNoCs and use the standard Vivado Vivado Design Suite XDC-based Pblock approach. For more information, see the *Vivado Design Suite Properties Reference Guide* (UG912) and *UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs* (UG949).

Following is an example snippet of a `<constraints>.json` with the Pblock specified. This file is given as an argument to the AI Engine compiler with the `--constraints` option.

```
{
  "GlobalConstraints": {
    "areaGroup": {
      "name": "unique_area_group_name",
      "exclude": true/false
    },
    "nodeGroup": ["rx.*"],
    "tileGroup": ["(col,row):(col,row)"],
    "shimGroup": ["col: col:"]
  }
}
```

In this example:

- **nodeGroup:** Lists all the kernels and PLIOs to Pblock. `rx*` covers all instances starting with the name `rx`.
- **tileGroup:** Gives the Pblock range on the AI Engine array for placing the kernels.
- **shimGroup:** Gives the Pblock range on the AI Engine array interface for placing the array interface instance, which can be PLIOs.

Following are additional considerations:

- If there is any data flow from the AI Engine to the PL, look at the DSP, block RAM, and UltraRAM columns, and use the AI Engine channels, which allow you to maximize the resources.

For example, a PL module that does not require DSPs does not require AI Engine channels, which are closer to DSP column.

- Make sure to insert the pipeline stages on the macro based on the way the stages are distributed across the device.

For example, data flows through AI Engine-PL channels and is stored in memory. This is a standard requirement, but there are not enough memories aligned to the AI Engine-PL channels.

- Align the floorplan based on the connectivity of the input stream of logic and output stream of logic.

For example, align the floorplan to PL-NoC-DDR for the output stream and to DDR-NoC-PL-AI Engine for the input stream.

Improving Performance Through the NoC

To improve NoC performance, see the following common issues.

No Communication between Master and Slave

- Check the NoC Connectivity tab. Is the master connected to the correct slaves?
- Check the address editors to ensure the master is sending to the correct system address range. For example, is the traffic generator set up to have the correct address range matching the address editor?

In case of the LPD connected to the NoC, verify that if the RPU uses this path. It can only access the first 32 bits of the address range and should not be used to access any resources (PL or DDR) above this range. AI Engine is an exception that the tools handle.

- Check that the memory controller passed calibration. See this [link](#) in the *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide (PG313)*.

- Check that the PMC PLL reference clock frequency is correct and matches the CIPS Wizard input clock frequency.
- Check the NoC frequency in the Clocks tab of CIPS.

Lower Than Expected Bandwidth

- Check the Connectivity tab. For best performance, use all four NSU ports of the DDRMC.
- Check the bandwidth values, the traffic class, and the following settings in the NoC QoS tab.

Note: Bandwidth values return to default values if you change the Connectivity tab.

- **Best Effort:** Default setting and the lowest priority. Use for general purpose masters.
- **Low Latency:** High priority read-only setting. This setting has priority over Best Effort in NPS/DDRMC and is only recommended for used APU cache refills. Too much use can decrease system performance.
- **Isochronous:** High priority setting. This setting has priority over Best Effort and Low Latency with a timer.
- Check the master behavior.
 - Masters that issue short AXI bursts to random access generally have lower bandwidth.
 - Masters that request more bandwidth than allocated can cause other masters to have lower than expected bandwidth.
 - A single NMU cannot saturate the bandwidth of a DDRMC. Increase the number of masters issuing requests to increase bandwidth from the DDR.
 - Make sure you know the data width for the master (e.g., IP integrator might inherit the wrong data width), and maximize the data width if needed.
 - A short burst size must match the NoC packet.
- Check the slave behavior.
 - The most common slave for the NoC is the integrated DDRMC. For information, see the *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* ([PG313](#)).
 - Verify that the bus width and AXI clock frequencies match between the ingress master to the NoC and the egress slave connection.
 - Check for large average latency stackup. Measure this at both the master and the slave.
 - Start with the slave. If there is already excessive latency, the issue is with the slave.
 - If the slave and master are within 5% and both have an average latency greater than 1000 clks, the issue is likely within the NoC, and the NoC might need to be constrained further.

- Perform a secondary analysis.
 - Inspect traffic through the NoC and look for switch contention, such as virtual channel (VC) assignment to QoS traffic classes.
 - Run integrated logic analyzer (ILA) on slave AXI4-Stream interfaces, and compute bus efficiency.
 - Redesign with AXI performance monitor (APM) on slave and master AXI4-Stream interfaces. Set up and capture extended metrics: bus efficiency, dead cycles, and slave-ready delays.
- The DDRMC slaves contain information about activates, bus turn-arounds, etc. For information, see *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313).

Improving Performance in the AI Engine

You can use the Xilinx Runtime (XRT) APIs to measure performance metrics like platform I/O port bandwidth, graph throughput, and graph latency. Use these APIs in the host application code with the AI Engine graph object. This object is used to initialize, run, update and exit graphs. In addition, you can use these APIs to profile graph objects to measure bandwidth, throughput, and latency. For more information, see [Run-Time Event API for Performance Profiling](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

AI Engine performance analysis typically involves system performance issues such as missing or mismatching locks, buffer overruns, and incorrect programming of direct memory access (DMA) buffers. It also includes memory/core stalls, deadlocks, and hot spot analysis. The AI Engine architecture has direct support for generation, collection, and streaming of events as trace data during simulation or hardware emulation. This data can then be analyzed for functional issues and latency problems between kernels, memory stalls, deadlocks, etc. For more information, see [Performance Analysis of AI Engine Graph Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

Improving Performance Through the CPM and PL PCIe

You can improve performance through the CPM and PL PCIe by examining I/O behaviors and routes, optimizing set-up for maximum performance, and debugging performance bottlenecks.

I/O Behaviors and Routes

The following CPM ports are available:

- **Master AXI4 MM Ports:** Only active in CPM DMA/AXI4 Bridge use modes:
 - There are two Master AXI4 MM ports at the CPM boundary. Modes:
 - **DMA (XDMA, QDMA):** May use both ports. Although not required to use both ports if the aggregated PCIe link throughput is below or equal to the NoC NMU limit for that particular device/silicon speed grade.
 - **AXI4 Bridge:** Only uses one port (AXI-MM0). If AXI4 Bridge is used in conjunction with DMA port, AXI4 Bridge traffic will only use AXI-MM0 while DMA traffic can use both ports.
 - In Endpoint mode both of these AXI4 Masters route to the NoC directly.
 - In Root Port mode (inherently AXI4 Bridge mode), one of these AXI4 Master (AXI-MM0) will route to the CCI-500 Interconnect directly. NoC can be reached via CCI-500.
- **Slave AXI4 MM Port:** Only active in CPM DMA/AXI4 Bridge use modes:
 - Only one Slave AXI4 MM port at the CPM boundary. Modes:
 - **All (XDMA, QDMA, AXI Bridge):** Can be used to access the internal registers or Bus Mastering (Read/Write) to the PCIe Link.
 - In both Endpoint and Root Port modes this port connects directly to the NoC.
- **Master and Slave AXI-ST Ports:** Only usable in CPM PCIe use mode.
 - Used in both Endpoint and Root Port mode.
 - Connects directly to the PL region.

The following PL PCIe ports are available:

- **Master and Slave AXI-ST Ports:** Only usable in CPM PCIe use mode.
 - Used in both Endpoint and Root Port mode.
 - Connects directly to the PL region.

Performance Setup

To maximize performance when using CPM, you must consider the following setup:

- **Master AXI4 Ports:** Because there are two AXI4 MM ports, you must balance the packets accordingly and maximize bus utilization at both ports.

- When to use:
 - Calculate your aggregated PCIe link throughput: This is PCIe Link Speed * PCIe Link Width
 - Calculate your AXI4 MM port throughput on one of the port: This is 128-bit * CPMTOPSWCLK frequency (CPM GUI selection. Speed grade dependent - consult your device/silicon datasheet).
 - If PCIe link throughput is greater than AXI4 MM port throughput, you must use both ports.

Note: Take into account the following considerations vs design complexity in using both ports if your bandwidth is nearly equal.

- PCIe link has some TLP overhead typically ~20-25% depending on packet sizes, Max Payload Size, and Max Read Request Size settings. Unaligned address transfers and/or scattered host memory might also affect this number due to inefficient DMA transfers.
- NoC has some overhead typically ~6% on the Write side due to metadata insertion but nearly optimum on the Read side.
- If using DDR memory, there might be additional overhead depending on the traffic pattern and the DDR bank/column/row settings.
- How to use:
 - Packets must not split into both ports. They must operate independently as much as possible to avoid Head of Line blocking due to AXI4 ID and PCIe tags ordering.
 - **QDMA:** Split your traffic based on Queue ID. Allocate some queues to use the first AXI-MM0 and the rest on the second AXI-MM1.
 - **XDMA:** Traffic will be split automatically based on DMA channel ID. Even DMA channels route to AXI-MM0 and odd DMA channels route to AXI-MM1.
 - **AXI4 Bridge:** Only use one port AXI-MM0. Therefore performance is expected to max out at the AXI4 MM port throughput only, and might not be up to the PCIe link throughput capability.
- **Slave AXI4 MM Port:** Because there is only one AXI4 MM port, performance through this port is expected to max out at AXI4 MM port throughput only and might not be up to the PCIe link throughput capability.
- **Master and Slave AXI-ST Ports:** Because CPM can only use AXI-ST ports in this mode directly to PL, therefore users are only required to operate their design at the same frequency and data bus width as the AXI-ST interface from the CPM or PCIe PL IP.

PL PCIe can only use AXI-ST ports, therefore users are only required to operate their design at the same frequency and data bus width as the AXI-ST interface from the CPM or PCIe PL IP.

Debug Steps

Performance bottlenecks can occur in various places. Most issues can be categorized into four separate debug sections:

- **Masters:** The device that is initiating the transfers.
 - Check their capabilities. This is achieved by calculating the AXI data bus width * AXI frequency of that Master.
 - If CPM is mastering:
 - Check if it is using both AXI4-MM ports or just a single AXI4-MM port. Calculate the aggregated bandwidth through these ports.
 - Check CPMTOPSWCLK frequency. CPM AXI4-MM ports are fixed 128-bit wide.
 - Check traffic pattern. Make sure packets are split by transfer IDs (can be Queue IDs or Channel IDs). *Do not split packet using the same AXI4 IDs.*
- **Slaves:** The device that is receiving the transfers.
 - Check their capabilities. This is achieved by calculating the AXI4 data bus width * AXI4 frequency of that Slave.
 - If CPM is receiving:
 - CPM only has one AXI4-MM port. Calculate the bandwidth through this port.
 - Check CPMTOPSWCLK frequency. CPM AXI4-MM ports are fixed 128-bit wide.
 - Check traffic pattern. Slave port is used to access internal registers, which internally uses AXI4-Lite interface with one AXI4 outstanding transaction, and also to Bus Master (Read/Write) to PCIe link. Do not interleave these transaction destinations to avoid Head of Line blocking.
- **Interconnects:** All interconnects and switches the packet has to go through.
 - Analyze all interconnects in the transfer path. It can be NoC, CCI-500, SmartConnect, etc. Check their throughput capabilities. This is achieved by calculating the AXI4 data bus width * AXI4 frequency of those interconnects.
 - Check the AXI4 outstanding transactions settings. The higher your system's latency or the smaller your AXI4 packet is, the higher you want this number to be to avoid credit starvation.
 - Check traffic pattern. *Ensure that "slow" and "fast" datapaths do not interleave* to avoid Head of Line blocking.
- **External/Software:** Factors that are outside of the Xilinx device.
 - **Software/Driver/Apps:** Software and drivers are typically much slower than hardware. To maximize throughput, you must ensure that there is minimal "maintenance" required from software during transfers:

- Maximize available Descriptor Queue ring size or Descriptor Chain size.
- Maximize Transfer Size (including Max Payload Size and Max Read Request Size settings at the host).
- Maximize number of Queues and DMA Channels.
- Minimize Interrupts and avoid excessive use of Poll Mode.
- Minimize Pointer or hardware updates.
- Avoid Bus Mastering from software. Let hardware do DMA or Bus Mastering.
- Avoid excessive copy from user to kernel level memory. Pin a memory at the host to be used for transfers.
- **Switches/IOMMU/Processor Links:** When transactions are going in and out of the host, there are various common modules along the paths that can increase latency and therefore your overall throughput:
 - Pick a path that has minimal PCIe switches. Analyze the available PCIe slots and their bus topology. Ensure the software or driver is running at the CPU attached to that PCIe bus. Use memory devices (disks, DDR memory, etc.) that are directly attached to that CPU.
 - Pick CPUs and PCIe switches that have high numbers of PCIe credits or ones that can use Extended PCIe Tags. This greatly improves the amount of PCIe packets outstanding, which will be required to compensate the high latency at the host. Enterprise grade systems typically advertise higher values compared to desktop or workstation systems.
 - Ensure the active PCIe slot and PCIe switch's link width and speed matches your device. Verify the PCIe link is trained to the optimum link speed and width.
 - IOMMU is often required in multifunction devices, however it adds latency to translate those PCIe addresses. Avoid relying on IOMMU unless it is required.
 - Disable Low Power State on PCIe and CPUs. While these features save power and can be significant in a large data center environment, repeatedly entering and exiting these power states can slow down transfers and increase latency.

Limitations

- Performance debug in simulation may not reflect the whole system behavior. Only simulating a CPM block can result in a fairly accurate performance model, however users must note the following simulation model limitations:
 - High power domain (HPD) and PS9 are modeled with a BFM. This BFM does not represent hardware in a cycle-accurate manner. This BFM also might require users to manually set parameters based on their IP settings, and might not propagate the same values used in hardware. The BFM might use a different clock frequency to speed-up simulation times through certain events. Or for simplicity, it might be modeled with a single clock domain that otherwise would not be in Hardware.

- When simulating CPM or PL PCIe as an Endpoint, Xilinx provides a Root Port PCIe model. It's not a BFM but it is based on PL PCIe IP architecture and most likely will have a more responsive turnaround time compared to a regular host system.
- Hardware probing using ILA internal to the CPM or PS or NoC block is not possible, but NoC NMU and NSU may provide internal statistics such as packet counts. However, users must keep in mind that one bottleneck in the data pipeline will eventually spread throughout the entire data path. That is, if the Slaves are throttling, the Interconnects and Masters may also look like they're throttling. Therefore this data might require more qualifications with other data before a final conclusion can be made.
- Similar to the above point, in DMA operation, everything is done in a loop. If software or host is throttling, hardware will eventually throttle and vice versa.

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](#))

Configuration

You must first successfully synthesize and implement your design to create a programmable device image (PDI). Once the PDI has been generated and all DRCs are analyzed and corrected, you can load the PDI onto the device using one of the following methods:

- **Direct Programming:** The PDI is loaded directly to the device using a cable, processor, or custom solution.
- **Indirect Programming:** The PDI is loaded into an external flash memory. The flash memory then loads the PDI into the device.

You can use the Vivado tools to accomplish the following:

- Create the PDI (.pdi).
- Select **Tools** → **Edit Device** to review the configuration settings for PDI generation.
- Format the PDI into flash programming file format (.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.



IMPORTANT! The Vivado Design Suite Device Programmer can use JTAG to read the JTAG_STATUS register data on Xilinx devices. In case of a configuration failure, the JTAG_STATUS register captures the specific error conditions that can help identify the cause of a failure. In addition, the JTAG_STATUS register allows you to verify the Mode pin settings MODE[3:0], check key supplies are detected, and determine the SelectMAP bus width. For details on the JTAG_STATUS register, see the Versal ACAP Technical Reference Manual ([AM011](#)).

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 PS/PMC

JTAG Status and Error Status

You can capture the overall PMC status and error status using the `jtag_status` and `error_status` commands from XSCT.

```
xsc% device status jtag_status
```

- Check BOOT_MODE [15:12] bits
- Check DONE [34] bit – This indicates configurations completed
- Ensure the voltages supplies are detected successfully – Bits [8:11] should be asserted

```
xsc% device status error_status
```

- Expected `error_status` value is 0x0 after board is power cycled and Power-on reset (POR) is asserted

Table 13: JTAG_STATUS Register Format

Bit	Field	Description
35	RESERVED	Reserved
34	DONE	Boot and configuration status indicator A value of 1 on DONE indicates boot and configuration is complete
33	JRDBK ERROR	JTAG readback status indicator A value of 1 on JRDBK indicates an error reading data from SBI
32	JCONFIG ERROR	JTAG data load error indicator A value of 1 means the SBI is not ready to accept data
31:28	PMC VERSION	PMC version
27:24	RESERVED	Reserved
23	JTAG SEC GATE	Security gate status A value of 1 means DAP AXI transactions are allowed
22	RESERVED	Reserved
21	PMC SCAN CLEAR DONE	Scan clear done indication A value of 1 means the scan clear is complete
20	PMC SCAN CLEAR PASS	Scan clear pass indication A value of 1 means the scan clear passed
19:16	RESERVED	Reserved
15:12	BOOT MODE [3:0]	Boot mode value captured from the MODE pins at release of POR_B
11	VCC_PMC DETECTED	VCC_PMC supply detected
10	VCC_PSLP DETECTED	VCC_PSLP supply detected
9	VCCINT DETECTED	VCCINT supply detected
8	VCC_SOC DETECTED	VCC_SOC supply detected
7	AES KEY ZEROIZED	AES key zeroized indicator A value of 1 indicates all keys are zeroized

Table 13: JTAG_STATUS Register Format (cont'd)

Bit	Field	Description
6	BBRAM KEY ZEROIZED	BBRAM key zeroized indicator A value of 1 indicates that the BBRAM key is zeroized
[5:4]	SELECTMAP BUS WIDTH	SelectMAP boot mode bus width detected 00 = No bus width detected 01 = SelectMAP 8-bit 10 = SelectMAP 16-bit 11 = SelectMAP 32-bit
3	SBI JTAG ENABLED	SBI JTAG indicator A value of 1 indicates the SBI is configured to receive data from the JTAG interface
2	SBI JTAG BUSY	SBI JTAG BUSY indicator A value of 1 indicates the SBI is BUSY and cannot accept data when in JTAG mode
1	RSVD_READS_0	Reserved, returns 0
0	RSVD_READS_1	Reserved, returns 1

Table 14: ERROR_STATUS Register Format

Bit	Field	Description
159:155	RSVD_READS_0	Reserved, returns 0
154:148	RESERVED	Reserved
147:136	BOOTROM FIRST ERROR	BootROM first error code detected
135:124	BOOTROM LAST ERROR	BootROM last error code detected
123:110	PLM MAJOR ERROR	PLM major error code
109:94	PLM MINOR ERROR	PLM minor error code
93:64	GSW ERROR	General software error code for PLM
63	RESERVED	Reserved
62	BOOTROM NCR	BootROM non-correctable error Set by RCU BootROM during boot
61	PLM CR	Platform loader and manager boot correctable error Set by PLM during boot
60	PLM NCR	Platform loader and manager boot non-correctable error Set by PLM during boot
59	GSW CR	General software correctable error after boot
58	GSW NCR	General software non-correctable error after boot
57	CFU ERROR	CFU error
56	CFRAME ERROR	CFRAME error
55	PSM CR	PSM correctable error
54	PSM NCR	PSM non-correctable error
53	DDRMC MB CR	DDRMC MicroBlaze correctable ECC error
52	DDRMC MB NCR	DDRMC MicroBlaze non-correctable ECC error
51	NOC CR	NoC correctable error

Table 14: **ERROR_STATUS** Register Format (cont'd)

Bit	Field	Description
50	NOC NCR	NoC non-correctable error
49	NOC USER ERROR	NoC user error
48	MMCM LOCK ERROR	MMCM lock error
47	AIE CR	AI Engine correctable error
46	AIE NCR	AI Engine non-correctable error
45	DDRMCMC MC ECC CR	DDRMCMC MC (memory controller) correctable ECC error
44	DDRMCMC MC ECC NCR	DDRMCMC MC (memory controller) non-correctable ECC error
43	GT CR	GT correctable error
42	GT NCR	GT non-correctable error
41	SYSMON CR	SYSMON correctable error
40	SYSMON NCR	SYSMON non-correctable error
39	USER PL0 ERROR	User-defined PL error
38	USER PL1 ERROR	User-defined PL error
37	USER PL2 ERROR	User-defined PL error
36	USER PL3 ERROR	User-defined PL error
35	NPI ROOT ERROR	NPI root error
34	SSIT ERROR3	SSI technology SLR error
33	SSIT ERROR4	SSI technology SLR error
32	SSIT ERROR5	SSI technology SLR error
31	PMC APB ERROR	PMC APB error. Includes errors from registers: PMC_LOCAL, PMC_GLOBAL, CRP, PMC_IOP_SECURE_SLCR, PMC_IOP, BBRAM_CTRL, PMC_ANLG, RTC
30	PMC BOOTROM ERROR	PMC BootROM validation error
29	RCU HARDWARE ERROR	RCU hardware error
28	PPU HARDWARE ERROR	PPU hardware error
27	PMC PAR ERROR	PMC switch and PMC IOP parity errors
26	PMC CR	PMC correctable errors
25	PMC NCR	PMC non-correctable errors
24	PMC SYSMON0 ALARM	PMC temperature shutdown alert and power supply failure detection errors from SYSMON
23	PMC SYSMON1 ALARM	PMC temperature shutdown alert and power supply failure detection errors from SYSMON
22	PMC SYSMON2 ALARM	PMC temperature shutdown alert and power supply failure detection errors from SYSMON
21	PMC SYSMON3 ALARM	PMC temperature shutdown alert and power supply failure detection errors from SYSMON
20	PMC SYSMON4 ALARM	PMC temperature shutdown alert and power supply failure detection errors from SYSMON
19	PMC SYSMON5 ALARM	PMC temperature shutdown alert and power supply failure detection errors from SYSMON
18	PMC SYSMON6 ALARM	PMC temperature shutdown alert and power supply failure detection errors from SYSMON

Table 14: **ERROR_STATUS** Register Format (cont'd)

Bit	Field	Description
17	PMC SYSMON7 ALARM	PMC temperature shutdown alert and power supply failure detection errors from SYSMON
16	PMC SYSMON8 ALARM	PMC temperature shutdown alert and power supply failure detection errors from SYSMON
15	PMC SYSMON9 ALARM	PMC temperature shutdown alert and power supply failure detection errors from SYSMON
14	CFI NCR	CFI non-correctable error
13	SEU CRC ERROR	SEU CRC error
12	SEU ECC ERROR	SEU ECC error
11:10	RSVD_READS_0	Reserved, returns 0
9	RTC ALARM	RTC alarm error
8	NPLL ERROR	PMC NPLL lock error
7	PPLL ERROR	PMC PPLL lock error
6	CLOCK MONITOR ERROR	Clock monitor errors
5	PMC TIMEOUT ERROR	PMC interconnect timeout errors from interconnect mission interrupt status register, interconnect latent status register, and timeout interrupt status register
4	PMC XMPU ERROR	PMC XMPU errors from register access error on APB. Includes read permission violation, write permission violation or security violation
3	PMC XPPU ERROR	PMC XPPU errors from register access error on APB. Includes Master ID not found, read permission violation, Master ID access violation, Master ID parity error, TrustZone violation
2	SSIT ERROR0	SSI technology SLR error
1	SSIT ERROR1	SSI technology SLR error
0	SSIT ERROR2	SSI technology SLR error

Rails Voltage Status

Confirm that power rails are detected successfully. To ensure that the power rails voltage value is correctly configured on board, read the SysMon registers. 0xF1110100 (PWR_STATUS) and 0xF111010C (PWR_SUPPLY_STATUS) are the registers that supply the power rail.

To read the registers, run the following:

```
xsc%mrdd -force 0xF1110100
xsc%mrdd -force 0xF111010C
```

Confirm that power rails are detected successfully. To ensure that the power rails voltage value is correctly configured on board, read the SysMon register 0xF111010C (PWR_SUPPLY_STATUS) by running the following:

```
xsc%mrdd -force 0xF111010C
```

Table 15: 11.1.21 Register (PMC_GLOBAL) PWR_STATUS

Register Name	Address	Width	Type	Reset Value	Description
PWR_STATUS	0xF1110100	32	RO	0x00000000	This register provides the Domain Isolation Status. 1 = isolated 0 = non-isolated The register is only reset by the Power-on Reset and maintains its state through a System Reset or Internal Power-on Reset.

Table 16: Register PWR_STATUS Bit-Field Details

Field Name	Bits	Type	Reset Value	Description
Reserved	31:19	RO	0x0	Reserved
VCCAUX_VCCRAM	18	RO	0x0	VCCAUX-VCCRAM interfaces and signals
VCCRAM_SOC	17	RO	0x0	VCCRAM-SoC interfaces and signals
VCCAUX_SOC	16	RO	0x0	VCCAUX-SoC interfaces and signals
PL_SOC	15	RO	0x0	PL-SoC interfaces and signals
PMC_SOC	14	RO	0x0	PMC-SoC interfaces and signals
PMC_SOC_NPI	13	RO	0x0	PMC-SoC NPI interface and signals
PMC_PL	12	RO	0x0	PMC-PL interfaces and signals
PMC_PL_CFRAME	10	RO	0x0	PMC-PL interfaces and signals
PMC_LPD	9	RO	0x0	PMC-LPD interfaces and signals
LPD_PL	6	RO	0x0	LPD-PL interfaces and signals
LPD_CPM	4	RO	0x0	LPD-CPM interfaces and signals
FPD_SOC	2	RO	0x0	FPD-SoC interfaces and signals
FPD_PL	1	RO	0x0	FPD-PL interfaces and signals

Table 17: 11.1.22 Register (PMC_GLOBAL) PWR_SUPPLY_STATUS

Register Name	Address	Width	Type	Reset Value	Description
PWR_SUPPLY_STATUS	0xF111010C	32	RO	X	This register provides the Power Supply Status.

Table 18: Register PWR_SUPPLY_STATUS Bit-Field Details

Field Name	Bits	Type	Reset Value	Description
Reserved	31:8	RO	0x0	Reserved
VCC_RAM	7	RO	0x0	RAM main power supply (also known as VCCINT_RAM)

Table 18: Register PWR_SUPPLY_STATUS Bit-Field Details (cont'd)

Field Name	Bits	Type	Reset Value	Description
VCCINT	6	RO	0x0	PL main power supply (also known as VCCINT_PL)
VCCAUX_SOC	5	RO	0x0	Auxillary power supply to SoC (also known as VCCAUX)
VCC_SOC	4	RO	0x0	NoC and DDR main power supply (SoC) (also known as VCCINT_SOC)
VCC_LPD	3	RO	0x0	LPD main power supply (also known as VCCINT_LPD)
VCC_FPD	2	RO	0x0	FPD main power supply (also known as VCCINT_FPD)
VCC_PMC	1	RO	0x0	PMC main power supply (also known as VCCINT_PMC)
VCCAUX_PMC	0	RO	0x0	Auxiliary power supply to PMC

PS/PMC Peripheral Resets

Confirm that the configured peripherals are out of reset.

- Before initiating requests to any peripheral in the PS or PMC, make sure the peripheral is out of reset.
- Check all reset registers in CRx (CRL, CRF and CRP) modules.

To read the QSPI reset register using XSCT, run the following command:

```
xsc t% mrd -force 0xF1260300
```

The expected value from above is 0x0 when QSPI is out of reset.

Table 19: Register RST_QSPI

Register Name	Address	Width	Type	Reset Value	Description
RST_QSPI	0xF1260300	1	RW	0x00000001	Reset for Individual block

Table 20: Register RST_QSPI Bit-Field Details

Field Name	Bits	Type	Reset Value	Description
RESET	0	RW	0x1	Block will be reset when asserted 1

PS/PMC Clocks

Confirm if the PS/PMC PLLs are stable and locked, clocks are in active state, and their divisor, multiplier, and source are correctly configured.

- Check the clocks configured correctly to each module with respect to speed grade.
- Read the CRx (CRL, CRF and CRP) modules registers for checking clocking.

To read the QSPI Clock Register using XSCT, run the following command:

```
xsct%mrdr -force 0xF1260118
```

Table 21: Register QSPI_REF_CTRL

Register Name	Address	Width	Type	Reset Value	Description
QSPI_REF_CTRL	0xF1260118	32	RW	0x01000400	This register controls this reference clock

Table 22: Register QSPI_REF_CTRL Bit-Field Details

Field Name	Bits	Type	Reset Value	Description
Reserved	31:25	RW	0x0	Reserved
ClkAct	24	RW	0x1	Clock active signal. 0 = disable the clock. 1 = enable the clock.
Reserved	23:18	RW	0x0	Reserved
Divisor	17:8	RW	0x4	10-bit divider value: 0 = divide by 1 1 = divide by 1 2 = divide by 2 etc. 1023 = divide by 1023 Note: Reset value (0x4) is divided by 4.
Reserved	7:3	RW	0x0	Reserved
SrcSel	2:0	RW	0x0	000: PPLL 011: NPLL Others: reserved

Debugging the NoC

The Versal ACAPs include a programmable AXI-interconnecting network on chip (NoC) used for sharing data between IP endpoints in the programmable logic (PL), the control, interface, and processing system (CIPS), and other integrated blocks. NoC is composed of a series of interconnected horizontal NoC (HNoC) and vertical NoC (VNoC) paths, supported by customizable and configurable hardware implemented components. The NoC components comprise NoC master unit (NMU), NoC slave unit (NSU), NoC packet switches (NPS), and NoC Inter-Die Bridge (NIDB). For more information on the architecture and debug and performance analysis features, see the *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* ([PG313](#)).

The programmable NoC supports end-to-end data protection for AXI4 memory mapped transactions that includes data and address parity and SECDED ECC across the entire NoC packet.

Every connection through the NoC has an associated QoS requirement. The details of supported QoS settings and their impact for a particular traffic type is discussed in this [link](#) in the *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide (PG313)*. For guidance on NoC/DDRMC performance tuning, see the Performance Tuning Tutorial in [GitHub](#).

Debugging the AI Engine

Xilinx offers a variety of tools and flows to debug designs running on the Versal AI Engine processors. They range from functional debug to debugging performance issues in the AI Engine algorithm through the various stages of development of an AI Engine application. The various tools and flows available and recommendations on when to use them are outlined below.

Functional debug of the AI Engine kernels typically involves running the x86 simulator in Vitis. This is a simulator that can be used extensively for functional debug, though care must be taken since it does not keep track of program memory size. More information around the options available and methods to run the x86 simulator can be found in [Simulating an AI Engine Graph Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation (UG1416)*.

Functional debug of the AI Engine kernels can also be performed using the aiesimulator in Vitis. This is a cycle approximate simulator that can be used extensively for functional debug. It models the Versal device including the NoC, PS, and PL components using SystemC. You can use the built-in debugger to step through the AI Engine source code as well to further debug design functionality. printf capability is also available, though it might impact the program memory size, which is tracked by the aiesimulator. The Vitis IDE has a debug view which displays registers, variables, available breakpoints, variables to register/memory mapping, internal/external memory contents, disassemble view for instruction, and an instruction pipeline (pipeline view) for a single AI Engine kernel.

Because the aiesimulator is also cycle approximate, it is possible to do extensive performance debug of the design. The simulator offers profiling and event trace capability, which can be used to analyze stalls and root cause performance issues in the design.

In addition, the aiesimulator also has options that can be used to detect and report any out of bounds access violations in the kernel code. More information around the options available and methods to run the aiesimulator can be found in [Simulating an AI Engine Graph Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation (UG1416)*.

When the AI Engine kernel is integrated into the rest of the Versal device using a Versal platform via V++, you can run the hardware emulator. This emulator is cycle accurate and can simulate PL logic either using RTL or SystemC simulation models. This simulation step should be performed after the v++ link step, with the “hw_emu” target specified to the v++ link command line. It is also possible to debug the AI Engine kernel using the debugger and stepping through both the kernel code and PS host code. This step uses QEMU to simulate the PS and a SystemC simulation model of the NoC. More information around the options available and methods to run the hardware emulator can be found in the following:

- [Integrating the Application Using the Vitis Tools Flow](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416)
- [Simulating an AI Engine Graph Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416)

When you are ready to take the design to hardware, you have a variety of debugging and profiling tools at your disposal. You can obtain profiling data when you run your design in hardware using run time event APIs in your PS host code, or using performance counters built into the hardware using a compile time option. Analyzing this data helps you gauge the efficiency of the kernels, the stall and active times associated with each AI Engine, and pinpoint the AI Engine kernel whose performance might not be optimal. This also allows you to collect data on design latency, throughput, and bandwidth. Detailed heatmaps and histograms of the profile are available in the Vitis Analyzer tool. More information around the options available and methods to analyze design performance in hardware can be found in [Performance Analysis of AI Engine Graph Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

You can also debug the AI Engine kernel source in hardware. This can help debug functional issues in hardware. More information on running the debugger in Vitis IDE and the various features associated with it can be found in [Debugging the AI Engine Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

Debugging with SmartLynq+

The Xilinx SmartLynq+ module is a High-Speed Debug and Trace module that primarily supports Versal devices. You can use SmartLynq+ modules in your Versal ACAP designs to take advantage of High-Speed Debug Port (HSDP) for reduced configuration times and high-speed debugging connectivity. The SmartLynq+ module can also be used in many cases as a direct replacement for remote lab PC as it includes connectivity via Gigabit Ethernet for a direct connection to your local area network.

To use HSDP with SmartLynq+, the design typically has the HSDP interface enabled in the Control, Interface, and Processing (CIPS) IP along with JTAG over the USB connectivity at the board level to a USB-C connector to interface with the SmartLynq+.

For more information, see this [link](#) in the *SmartLynq+ Module User Guide* (UG1514).

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.

AXI4 Debug Hub Connectivity

To use the Vivado® debug cores, the design must contain an AXI4 debug hub. The AXI4 debug hub connects an AXI-MM interface of the CIPS with the AXI4-Stream interface. The interface connects to Vivado debug cores, which includes the following types of cores:

- AXI4-Stream Integrated Logic Analyzer (AXIS-ILA)
- AXI4-Stream Virtual Input/Output (AXIS-VIO)
- PCI Express® Link Debugger

If the design contains any Vivado debug cores and the CIPS has one or more PL resets enabled, the AXI4 debug hub is automatically inserted by Vivado during `opt_design` and connected to the CIPS using a NoC. It is possible to instantiate the AXI4 debug hub and connect it to the CIPS manually, but the automatic insertion will not take place.

Three different scenarios can determine if AXI4 debug hub auto-insertion will occur during `opt_design`:

Table 23: AXI4 Debug Hub Auto-Insertion

Synthesized Netlist Contents	AXI4 Debug Hub Insertion Action
CIPS with 1 or more PL Reset Enabled. No AXI4 debug hubs.	An AXI4 debug hub will be inserted during <code>opt_design</code> and connected to the CIPS using a NoC instance.
CIPS with 1 pre-existing AXI4 debug hub	Any debug cores in the design that are not found to be manually connected to an AXI4 debug hub will be connected to the preexisting debug hub during <code>opt_design</code> .
A CIPS with multiple AXI4 debug hubs	Automatic stitching will not occur without taking one of the two actions: <ul style="list-style-type: none"> • The AXI4-Stream interfaces on the debug cores are manually connected to the desired AXI4 debug hub. • The connectivity between each debug core and associated AXI4 debug hub is specified using the <code>connect_debug_core</code> constraint.

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 24: 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.	<ul style="list-style-type: none"> 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.
Netlist insertion probing flow	<p>Use one of the following two methods to identify the signal for debug:</p> <ul style="list-style-type: none"> 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. <p>Once the signal is marked for debug, use the Set up Debug wizard to guide you through the Netlist Insertion probing flow.</p>	<ul style="list-style-type: none"> 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 <code>set_property</code> 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.	<ul style="list-style-type: none"> 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.

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 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 clock chosen for the AXIS-ILA `clk` port 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.
- Ensure that the clock connected to the AXI4 debug hub is a free running clock and is synchronous to the AXI master that is connected to the S_AXI port. Failure to do so could result in an inability to communicate with the debug core when the design is loaded onto the device.
- 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 25: Impact of ILA Features on Design Timing and Resources

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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 1-2 for Basic • 1-4 for Advanced • +1 for Capture Control 	Medium to High Impact	<ul style="list-style-type: none"> • No additional block RAMs • Slight to moderate increase in LUT/FF count
Data Depth	To capture more data samples	High Impact	<ul style="list-style-type: none"> • Additional block RAMs per ILA core • Slight increase in LUT/FF count
ILA Probe Port Width	To debug a large bus versus a scalar	Medium Impact	<ul style="list-style-type: none"> • Additional block RAMs per ILA core • Slight increase in LUT/FF count
Number of Probes Ports	To probe many nets	Low Impact	<ul style="list-style-type: none"> • 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 AXIS-ILA by setting the number of input pipeline stages. This setting can be found in the Advanced tab of the AXIS-ILA GUI or set with the `C_INPUT_PIPE_STAGES` property when using Tcl insertion.

Using VIO Cores

The Virtual Input/Output (AXIS-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 AXIS-VIO debug core must be instantiated in the design and can be used in both IP integrator and RTL. The AXIS-VIO core is available in the IP catalog for RTL-based designs and in IP integrator.

For information on customizing the AXIS-VIO core, see the *Virtual Input/Output (VIO) with AXI4-Stream Interface LogiCORE IP Product Guide* (PG364). For information on taking measurements with an AXIS-VIO core, see this [link](#) in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

VIO Core Considerations

When using AXIS-VIO cores, consider the following:

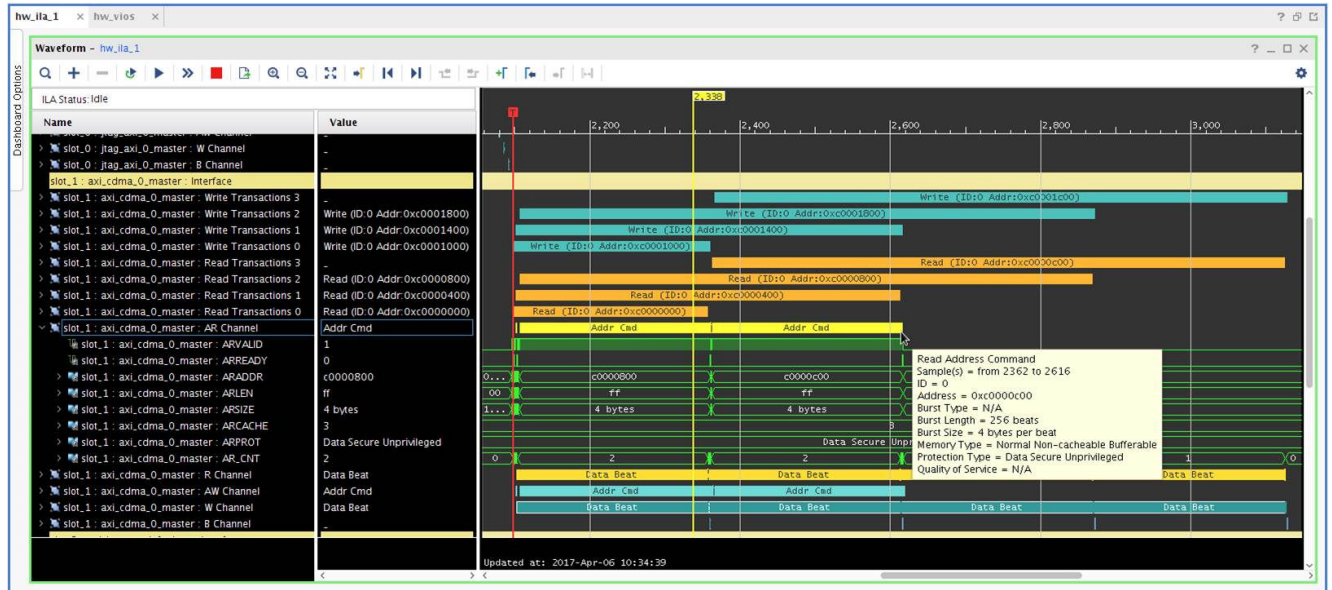
- Signals connected to AXIS-VIO input probes must be synchronous to the clock connected to the AXIS-VIO `clk` port on the AXIS-VIO core. Connecting signals that are not synchronous to the `clk` port results in a clock domain crossing at the AXIS-VIO input probe port.
- Signals driven from AXIS-VIO output probes are asserted and deasserted synchronous to the clock connected to the AXIS-VIO `clk` port on the AXIS-VIO core.
- The AXIS-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 AXI Interfaces in Vivado Hardware Manager

The AXIS-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 AXIS-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 AXIS-ILA.

Figure 40: Waveform Window



For more information on AXIS-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 IBERT GTY for Transceiver Link Characterization

Versal ACAP GTY transceivers support an Integrated Bit Error Ratio Tester (IBERT) which enables in-system serial I/O validation and debug. This allows the measuring and optimization of high-speed serial I/O links.

IBERT should be used when you are interested in measuring the quality of a signal after receiver equalization has been applied to the received signal.

Unlike previous architectures, the Versal ACAP IBERT GTY functionality is integrated into the GTY transceiver and requires only a design which uses the transceivers. For more information on IBERT functionality, see this [link](#) in the *Versal ACAP Transceivers Wizard LogiCORE IP Product Guide* (PG331).

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.
- CIPS does not have `pl_resetn0` enabled for debug hub insertion flow.

Generating AXI Transactions

AXI Transactions can be generated natively using the Debug Packet Controller (DPC) without the need for additional IP.

1. Load the Programmable Device Image (PDI) into the device.
2. Start the Xilinx System Debugger `xbdb`.
3. Connect to the hardware server using `connect`.

Note: Remote connections can use the `-host` switch to specify a remote host.

4. List the available targets by typing `targets`.
5. Set the current target to the DPC, `target <number next to DPC>`.

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 is possible if the device has a one or more free pins that can be used for this purpose.

1. Launch the Vivado tools, and open the routed design checkpoint.
2. Click **Layout** → **ECO** to switch to ECO mode.
3. Click **Create Port** in the ECO Navigator. Select the appropriate **IOSTANDARD**, and fill in the **Name**. In the Location dialog box enter the desired free package pin to be used for debugging. Click **OK**. A new I/O port appears in the Schematic window.
4. Click **Create Cell** in the ECO Navigator, and select **OBUF** as the cell type. Give the cell a **Name**, and click **OK**.
5. Hold the **Ctrl** key and select both the **O** port on the **OBUF** in the prior step and the port created in step 3. In the ECO Navigator, click **Create Net**. Fill in the field for a **Name** and click **OK**. This connects the **OBUF** to the I/O port.
6. In the Schematic window, locate the net in your design to be connected to the external port. Hold the **Ctrl** key, and select both the net from your design and the **I** pin on the **OBUF** created in the earlier step.
7. Click **Connect Net** in the ECO Navigator to connect the net to the **OBUF**.

These steps can be repeated for as many nets as are desired. After all of the nets are connected to the desired pins, use the following instructions to launch incremental route and write a new bitstream containing the changes implemented in the previous steps.

1. Click **Route Design** in the ECO Navigator. Select **Incremental Route**, and click **OK**.
2. After routing is complete, a new PDI can be written by selecting **Generate Bitstream** from the ECO Navigator. If the design contains any Vivado Debug Cores such as ILA or VIO, a new debug probes file should also be written by selecting **Write Debug Probes**.

Note: If desired, this routed checkpoint can be saved by clicking **Save Checkpoint As** under Programing the ECO Navigator.

3. The new bitstream contains the connection to the I/O ports.

Using Remote Debugging

To debug or upgrade your design remotely, use the Xilinx Hardware Server product to connect to a remote computer in the lab.

For more information on connecting remotely see this [link](#) in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

You can also use the SmartLynq+ module, which can be used in many cases as a direct replacement for a remote lab PC.

Using the ChipScoPy Python Client For Debugging

ChipScoPy is an open source project from Xilinx that enables high-level control of the Versal ACAP debug IP running in hardware. Using a simple Python API, you can control and communicate ChipScope™ debug such as the Integrated Logic Analyzer (ILA), Virtual Input/Output IO (VIO), device memory access, and more. For more information, see the GitHub repository at <http://www.github.com/Xilinx/chipscopy>.

Debugging the Software

Software debug methodologies largely depend on the following factors:

- Target processor for the application
- The type of Operating System (OS) used
- Application software/kernel debug
- Functionality/performance debug

Target Processor

The target processor for the application can be an Arm® Cortex®-A72 processor, Cortex-R5F processor, MicroBlaze™ processor in PL, or AI Engine core processor. Depending on the processor type, the debugger can connect to one of the processor targets and perform single-stepping, breakpoint insertion to debug specific portion of the code. If the target processor is Cortex-A72 or Cortex-R5F, it is possible to connect to the Arm CoreSight™ interface using HSDP or JTAG. Vitis debugger or third-party debuggers (for example, Lauterbach) can connect to the CoreSight interface and perform single-stepping, breakpoint insertion, instruction disassembly, core register status, etc.

For an AI Engine processor, Vitis debugger can connect to the AI Engine debug interface and perform single-stepping, breakpoint insertion, and view program/data memory. Vitis debugger can connect to the target processor using the XSCT interface. The XSCT interface provides a full set of debug commands to read internal registers and memory of a processor. For example, if software code hangs at a specific instruction, you can read the PC address and check the last instruction it executed. For more information about XSCT commands, see the [Xilinx Software Command-Line Tool](#) in the *Vitis Unified Software Platform Documentation* (UG1416).

Operating System

The type of operating system determines the software debug steps. If the application is based on bare-metal OS, it is easier to read specific peripheral address from the application. For example, if the application enables a specific kernel and kernel data flow is not occurring, it is possible to read the control register, length register of the kernel from application, and check if the programmed values are correct.

For bare-metal application, it is possible to dump the input/output buffer of the kernel based on the physical address that helps in debugging from the software point-of-view. If the input buffer itself does not match with the expected input, further triage can be done by performing write/readback to the same DDR buffer from the application. If the readback data does not match, the further debug can be focused on the DRAM interface.

For Linux-based application, dumping the buffer content to a file requires the allocated buffer to be memory mapped and a physical address can be returned. Then, the application can compare the content of the input and output buffer with the reference input. The application can access the memory mapped kernel register space using the device memory map (`/dev/mem`) to check if the kernel control, size, and address values are getting programmed correctly. If the values are incorrect, further debug can happen on the application to device driver interface.

Application Software/Kernel Debug

A Linux-based software stack might require debugging in both application and kernel layers. A typical Linux application interacts with the kernel space driver using IOCTL calls. If an I/O read/write response is not returned from the kernel driver, you can check the specific hardware registers to see if the address, length, interrupt-related mask/enable signals are programmed correctly. If the registers are programmed properly, the next debug step is to check if any interrupts have been registered for the specific hardware device by looking at `/proc/interrupts` using a `cat` command.

If the interrupts are not registered, further debugging needs to occur from the hardware side. If the interrupts are registered, you can add specific kernel print statements in the driver to debug interrupt service routines and other function calls in the device driver. The application level debug can be done using GDB debugger. Also, ensure that the application code is free of memory leak that typically occurs when a dynamically allocated memory is not freed up once its use is over. Such issues can be debugged using the Valgrind tool that helps detect such scenarios.

Functionality/Performance Debug

Depending on whether the debug is related to functionality or performance, different debug strategies can be adopted. For functionality debug, a few methods suggested above can be adopted depending on application type, OS usage, and application/kernel level debug. For performance debugging, if the application is in the control path, function profiling using a built-in timer (TTC or global counter in Cortex-A72 processor) can be done to check if the function that is performing accelerator callback is slowing down at some point and further debug can happen from the software point-of-view.

If the software application is in the data path, similar profiling to function profiling, you can perform if the buffer allocation is causing any overhead that is delaying the calling of accelerator function. Typically an accelerator requiring a large contiguous memory is allocated using contiguous memory allocator. If the contiguous memory is used by other accelerators, memory allocation might take longer. In such scenarios, a static allocation might help. Other debug strategies for prioritizing hardware/software performance issues can be counting number of interrupts in a specific timing window. If the number of interrupts are not smaller compared to expected, further debug can be done from the hardware point-of-view.

Validation

The different compute domains of Versal™ ACAP challenge traditional FPGA validation methods. In addition to programmable logic and a processor subsystem, Versal devices include AI Engines, making system validation a complex task compared to traditional FPGAs.

This validation methodology is built around the follow key concepts:

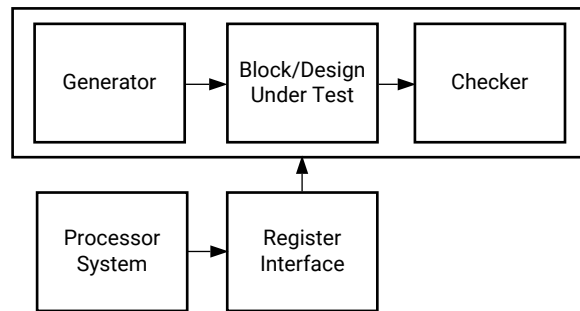
- **Block/IP Validation:** The individual RTL and HLS IP in the PL can be validated individually before the system is integrated.
- **AI Engine validation:** AI Engines at the interface level can be viewed as AXI-MM or AXI4-Stream IP.
- **System validation:** After the individual blocks are validated, the entire system can be validated, using processors to coordinate data flow, test vector generation, monitor, etc.

Block and IP Validation

The PL for the system can include different types of IP. If the IP is compliant with the Vitis™ environment (i.e., the interfaces to the IP are only AXI memory mapped, AXI4-Stream, clocks, and reset), you can use an existing Vitis platform from the installation area to integrate the blocks with a generator and checker logic. If the IP has unique ports that are not compliant with the Vitis environment, you can develop wrappers around the IP to make them compliant or use traditional Vivado® IP integrator methods to integrate the design with generators and checkers.

The following figure shows a generalized block validation design.

Figure 41: Generalized Block Validation Design



X25053-012121

Generator and Checkers

You can use any of the following options for generators and checkers:

- Off-the-shelf AXI-DMA or MCDMA IP from the Vivado IP integrator catalog, which can stream data from the DDR memory to the programmable logic. This method is the easiest way to bring the block to hardware. For more information, see the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994).
- LFSR generator and checkers from the Vivado tools language templates. Alternatively, you can design your own version of a pseudo random data generator checker.
- UltraRAM to block RAM, which can provide specific data to verify the functionality of the block.

Note: You can embed RAM with initial memory contents (MEM files) and with additional circuitry (such as AXI GPIO), which gives you the flexibility to control the test harness.

- AXI traffic generators (ATGs), which can generate pre-configured traffic types and can also be programmed to check for a desired length of data. Although this does not always provide a data integrity check, it is a quick way to validate the intent of the design under test.
- AXI or AXI4-Stream ILA, which can monitor data at the output of design under test. Using this method, you can visually analyze the data using the Vivado IDE and visualize waveforms without much overhead. For more information, see the *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

After a test harness is built, you can test the design on hardware without any additional software. All the register interfaces are available through the Xilinx® System Debugger `x_sdb` console in JTAG mode.

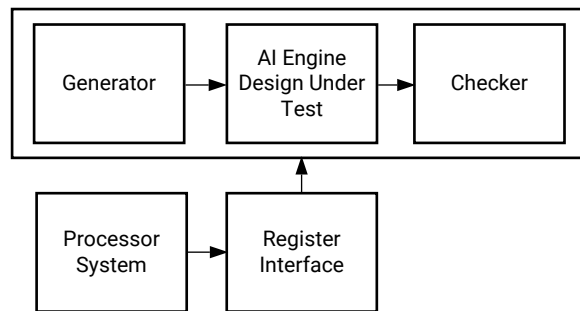
Alternatively, you can develop a software application to orchestrate generator and checker functions based on the register control interface. This method is particularly useful if the design process involves multiple IP and helpful if you want to reuse the validation infrastructure for different stages in design.

AI Engine Design Validation

You can validate an AI Engine design using validation techniques similar those used for block validation. One primary difference is that AI Engine is an AXI-interface based IP, making it easier to interface with the different IP in the IP integrator catalog or in Vitis HLS. One common method is to use S2MM and MM2S-based IP to generate DDR memory-based traffic to the AI Engine, retrieve the data from the AI Engine, and transmit the data back to the DDR memory.

The following figure shows the block diagram of an AI Engine design with test harness around it.

Figure 42: AI Engine Design with Test Harness



X25054-012121

Unlike the block IP validation methodology described in the previous section, AI Engine validation includes the following challenges:

- Number of AXI input and out streams from programmable logic to AI Engines.
- Bandwidth of data expected by AI Engine.

For simplicity, AI Engine design validation can be broadly classified into functional validation and performance validation.

Functional Validation

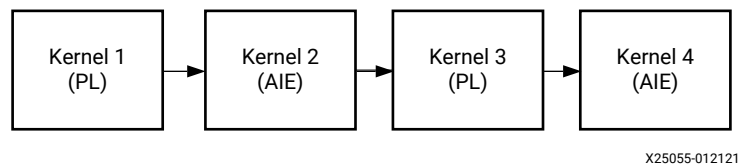
Functional validation involves checking the functional integrity of the graph in the AI Engine. This can be achieved using the validation methods described earlier, such as using ATGs/AXI DMA/MCDMA/S2MM-MM2S HLS based kernels to directly transmit test data from DDR memory to AI Engines. This is the simplest method to test the functionality of graphs within the AI Engines.

To program AI Engines, start the graphs, and orchestrate data flow, the design must be Vitis compliant. For more information, see the *Versal ACAP Design Guide* (UG1273). You can use the Vitis platforms, provided with the Vitis installation tools, and build the validation design based. Additionally, enabling AI Engines does not require a software host application, but a baremetal-based host application is required to leverage all the features of AI Engine tools as described in the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

Performance Validation

Performance validation can be challenging, but it is the most important step in verifying that you are meeting the AI Engine bandwidth. Consider a following example of a complex system design with multiple components in AI Engines as well as programmable logic.

Figure 43: Complex System Design



Any backpressure between Kernels 1, 2, 3 will result in under performance of Kernel 4, even though Kernel 4 could be designed to operate most optimally. This is an additional challenge, which was not always present in traditional FPGA based designs.

For the entire system to operate efficiently and correctly, verify the functionality and performance of each of the kernels described above using the methods described in this and the preceding sections before they are integrated together, to achieve required system design.

AI Engine performance validation requires the generators to be able to generate data at required throughput and checkers to be able to sink data at required throughput. As each stream interface between the AI Engine and programmable logic can operate at 4 Gb/s, if the performance validation metrics are not met, you can modify the interface to widen the bus and reduce the frequency. For example, a 64-bit/500 MHz between the AI Engines and PL can be modified to 128b/250 MHz interface without jeopardizing the performance of entire system. Such decisions can be made only after performance validation of Kernels 2 and 4 independently. For more information, see the [Performance Analysis of AI Engine Graph Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

The AI Engine tools include APIs to measure throughput of each interface in the AI Engine-PL shim interface without any additional hardware. This is the easiest method to get an idea of the performance numbers between the AI Engine and PL. For more information, see the [Performance Analysis of AI Engine Graph Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

There are two approaches to generating enough bandwidth for the AI Engine:

- Fine tune the test harness generators/checkers such as AXI4 DMA/S2MM-MM2S kernels to source and sink enough bandwidth as needed by the AI Engines.
- Design custom RTL-based kernels that generate LFSR or BRAM/URAM based vectors at the required clock frequency in the fabric.

For high performance designs, the latter method provides consistent bandwidth because there is no data flow between the DDR memory and programmable logic. For AI Engine based designs with faster bandwidth requirements, the latter method is the more efficient way to create test harness. However this does require some additional work.

Additional hardware debug can be performed using the Vitis tools, as described in the *Vitis Unified Software Platform Documentation* ([UG1416](#)).

System Validation

After individual subsystems are validated, you can integrate the entire system in the Vitis environment. You can use the validation methods described in this chapter to validate the complete system.

Design Debug

If there is a functional mismatch between validation and verification, you can debug the design as described in [Chapter 5: Configuration and Debug](#).

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

Solution Centers

See the [Xilinx Solution Centers](#) for support on devices, tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Documentation Navigator and Design Hubs

Xilinx[®] Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

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

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

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

Note: For more information on DocNav, see the [Documentation Navigator](#) page on the Xilinx website.

References

These documents provide supplemental material useful with this guide.

1. [Vivado® Design Suite Documentation](#)

Vivado Design Suite User and Reference Guides

1. *Vivado Design Suite Tcl Command Reference Guide* ([UG835](#))
2. *Vivado Design Suite User Guide: Design Flows Overview* ([UG892](#))
3. *Vivado Design Suite User Guide: Using the Vivado IDE* ([UG893](#))
4. *Vivado Design Suite User Guide: Using Tcl Scripting* ([UG894](#))
5. *Vivado Design Suite User Guide: System-Level Design Entry* ([UG895](#))
6. *Vivado Design Suite User Guide: Designing with IP* ([UG896](#))
7. *Vivado Design Suite User Guide: Embedded Processor Hardware Design* ([UG898](#))
8. *Vivado Design Suite User Guide: I/O and Clock Planning* ([UG899](#))
9. *Vivado Design Suite User Guide: Logic Simulation* ([UG900](#))
10. *Vivado Design Suite User Guide: Synthesis* ([UG901](#))
11. *Vivado Design Suite User Guide: Using Constraints* ([UG903](#))
12. *Vivado Design Suite User Guide: Implementation* ([UG904](#))
13. *Vivado Design Suite User Guide: Hierarchical Design* ([UG905](#))
14. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* ([UG906](#))
15. *Vivado Design Suite User Guide: Power Analysis and Optimization* ([UG907](#))
16. *Vivado Design Suite User Guide: Programming and Debugging* ([UG908](#))
17. *Vivado Design Suite User Guide: Dynamic Function eXchange* ([UG909](#))
18. *Vivado Design Suite User Guide: Getting Started* ([UG910](#))
19. *Vivado Design Suite Properties Reference Guide* ([UG912](#))
20. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* ([UG973](#))
21. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* ([UG994](#))
22. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* ([UG1118](#))
23. *Xilinx Power Estimator User Guide for Versal ACAP* ([UG1275](#))

Vivado Design Suite Tutorials

1. Vivado Design Suite Tutorial: Design Flows Overview ([UG888](#))
2. Vivado Design Suite Tutorial: Logic Simulation ([UG937](#))
3. Vivado Design Suite Tutorial: Embedded Processor Hardware Design ([UG940](#))
4. Vivado Design Suite Tutorial: Dynamic Function eXchange ([UG947](#))

Other Xilinx Documentation

1. Versal ACAP GTY and GTYP Transceivers Architecture Manual ([AM002](#))
2. Versal ACAP Clocking Resources Architecture Manual ([AM003](#))
3. Versal ACAP Configurable Logic Block Architecture Manual ([AM005](#))
4. Versal ACAP SelectIO Resources Architecture Manual ([AM010](#))
5. Versal ACAP Technical Reference Manual ([AM011](#))
6. Versal ACAP Packaging and Pinouts Architecture Manual ([AM013](#))
7. Versal ACAP CPM CCIX Architecture Manual ([AM016](#))
8. Versal Architecture and Product Data Sheet: Overview ([DS950](#))
9. Versal ACAP CIPS Verification IP Data Sheet ([DS996](#))
10. SmartConnect LogiCORE IP Product Guide ([PG247](#))
11. AXI Verification IP LogiCORE IP Product Guide ([PG267](#))
12. Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide ([PG313](#))
13. Advanced I/O Wizard LogiCORE IP Product Guide ([PG320](#))
14. Clocking Wizard for Versal ACAP LogiCORE IP Product Guide ([PG321](#))
15. Versal ACAP Transceivers Wizard LogiCORE IP Product Guide ([PG331](#))
16. Versal ACAP Integrated Block for PCI Express LogiCORE IP Product Guide ([PG343](#))
17. Versal ACAP DMA and Bridge Subsystem for PCI Express Product Guide ([PG344](#))
18. Versal ACAP PCIe PHY LogiCORE IP Product Guide ([PG345](#))
19. Versal ACAP CPM Mode for PCI Express Product Guide ([PG346](#))
20. Versal ACAP CPM DMA and Bridge Mode for PCI Express Product Guide ([PG347](#))
21. Control, Interface and Processing System LogiCORE IP Product Guide ([PG352](#))
22. Integrated Logic Analyzer (ILA) with AXI4-Stream Interface LogiCORE IP Product Guide ([PG357](#))
23. AXI4 Debug Hub LogiCORE IP Product Guide ([PG361](#))

24. *Virtual Input/Output (VIO) with AXI4-Stream Interface LogiCORE IP Product Guide (PG364)*
25. *Simulating FPGA Power Integrity Using S-Parameter Models* ([WP411](#))
26. *Extending the Thermal Solution by Utilizing Excursion Temperatures* ([WP517](#))
27. *Versal ACAP Schematic Review Checklist* ([XTP546](#))
28. *Mechanical and Thermal Design Guidelines for Lidless Flip-Chip Packages* ([XAPP1301](#))
29. *Versal ACAP PCB Design User Guide* ([UG863](#))
30. *Versal ACAP AI Engine Programming Environment User Guide* ([UG1076](#))
31. *AI Engine Kernel Coding Best Practices Guide* ([UG1079](#))
32. *Versal ACAP Design Guide* ([UG1273](#))
33. *Xilinx Embedded Design Tutorials: Versal Adaptive Compute Acceleration Platform* ([UG1305](#))
34. *Versal Architecture Prime Series Libraries Guide* ([UG1344](#))
35. *Versal Architecture AI Core Series Libraries Guide* ([UG1353](#))
36. *Vitis High-Level Synthesis User Guide* ([UG1399](#))
37. *Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#))
38. [Bootgen Tool](#) in the Embedded Software Development flow of the *Vitis Unified Software Platform Documentation* (UG1416)
39. [Creating Embedded Platforms in Vitis](#) in the Application Acceleration Development flow of the *Vitis Unified Software Platform Documentation* (UG1416)
40. [Debugging the AI Engine Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416)
41. [Integrating the Application Using the Vitis Tools Flow](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416)
42. [Performance Analysis of AI Engine Graph Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416)
43. [Programming the PS Host Application](#) in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416)
44. [Special Considerations for Embedded Platform Creation](#) in the Application Acceleration Development flow of the *Vitis Unified Software Platform Documentation* (UG1416)
45. [Using the Vitis IDE](#) in the Application Acceleration Development flow of the *Vitis Unified Software Platform Documentation* (UG1416)
46. [Vitis Accelerated Software Development Flow Documentation](#) in the Application Acceleration Development flow of the *Vitis Unified Software Platform Documentation* (UG1416)
47. [Vitis Embedded Software Development Flow Documentation](#) in the *Vitis Unified Software Platform Documentation* (UG1416)

48. [Vitis HLS Documentation](#) in the Application Acceleration Development flow of the *Vitis Unified Software Platform Documentation* (UG1416)
49. [Vitis HLS Methodology](#) in the Vitis HLS flow of the *Vitis Unified Software Platform Documentation* (UG1416)
50. *Vitis Unified Software Platform Documentation* ([UG1416](#))

Training Resources

1. [Vivado Design Suite QuickTake Video: Vivado Design Flows Overview](#)
2. [Vivado Design Suite QuickTake Video: Partial Reconfiguration in Vivado Design Suite](#)
3. [Vivado Design Suite QuickTake Video: Creating Different Types of Projects](#)
4. [Vivado Design Suite QuickTake Video: Managing Sources With Projects](#)
5. [Vivado Design Suite QuickTake Video: Using Vivado Design Suite with Revision Control](#)
6. [Vivado Design Suite QuickTake Video: Managing Vivado IP Version Upgrades](#)
7. [Vivado Design Suite QuickTake Video: Configuring and Managing Reusable IP in Vivado](#)
8. [Vivado Design Suite QuickTake Video: Design Analysis and Floorplanning](#)
9. [Vivado Design Suite Video Tutorials](#)

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at [https://](https://www.xilinx.com/legal)

www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2021 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. All other trademarks are the property of their respective owners.