See all versions
of this document

Vivado Design Suite User Guide

Dynamic Function eXchange

UG909 (v2026.1) July 1, 2026

AMD

Table of Contents

Chapter 1: Introduction **6**
- Navigating Content by Design Process 7
- Introduction to Dynamic Function eXchange 8
- Terminology 9
- Design Considerations 12
- Dynamic Function eXchange Licensing 17

Chapter 2: Common Applications **18**
- Networked Multiport Interface 18
- Configuration by Means of Standard Bus Interface 20
- Dynamically Reconfigurable Packet Processor 22
- Asymmetric Key Encryption 22
- Summary 24

Chapter 3: Vivado Software Flow **25**
- Dynamic Function eXchange Commands 26
- Dynamic Function eXchange Constraints and Properties 33
- Apply Reset After Reconfiguration 46
- Software Flow 48
- Tcl Scripts 54
- Nested Dynamic Function eXchange 54
- Abstract Shell for Dynamic Function eXchange 66

Chapter 4: Vivado Project Flow **81**
- RTL Project Flow in the Vivado IDE 81
- IP Integrator Using Block Design Containers 111
- Abstract Shell Project Flow 130

Chapter 5: Design Considerations and Guidelines for All AMD Devices **143**
- Dynamic Function eXchange IP 143
- Design Hierarchy 144

Embedded I/O Usage Guidelines....148
Partition Pin Placement....149
Active-Low Resets and Clock Enables....149
Decoupling Functionality....150
Initializing Reconfigurable Modules after DFX....151
Black Boxes....160
Effective Approaches for Implementation....161
Configuration Analysis Report....162
Managing Constraints for a DFX Design....165
Defining Reconfigurable Partition Boundaries....167
Avoiding Deadlock....168
Design Revision Checks....169
Simulation and Verification....169
ELF Association Within DFX....170

Chapter 6: Design Considerations and Guidelines for 7 Series and Zynq Devices....171
Design Elements Inside Reconfigurable Modules....171
Global Clocking Rules....172
Floorplanning for 7 Series Devices....173
Using High Speed Transceivers for 7 Series Devices....183
Dynamic Function eXchange Design Checklist (7 Series)....183

Chapter 7: Design Considerations and Guidelines for UltraScale and UltraScale+ Devices....187
Design Elements Inside Reconfigurable Modules....187
Floorplanning for UltraScale and UltraScale+ Devices....188
Global Clocking Rules....206
I/O Rules....206
Using High Speed Transceivers for UltraScale and UltraScale+ Devices....208
Dynamic Function eXchange Checklist for UltraScale and UltraScale+ Device Designs 209
Recommended Floorplanning Constraints....214

Chapter 8: Design Considerations and Guidelines for Versal Devices....226
DFX Design Migration from Standard Flow to Advanced Flow....226
Design Elements Inside Reconfigurable Modules....227
I/O Rules....228

Floorplanning for Versal Devices..... 230
Global Clocking Rules..... 260
Network on Chip.....273
DFX for SSI Technology Versal Devices.....291
Hardened DDR Memory Controllers.....302
Logical Decoupling.....304
ECO Flow Support for DFX Designs.....306
Debugging Versal Device DFX Designs..... 307
Recommended Floorplanning Constraints.....317
Using Report DFX Summary..... 329

Chapter 9: Configuring the Device.....339
Configuration Modes.....339
Bitstream Type Definitions.....341
Dynamic Function eXchange through ICAP for Zynq Devices.....346
Dynamic Function eXchange for Spartan UltraScale+ Devices.....347
Tandem Configuration and Dynamic Function eXchange..... 349
Formatting BIN Files for Delivery to Internal Configuration Ports..... 353
Summary of BIT Files for UltraScale Devices..... 354
System Design for Configuring an FPGA.....355
Partial BIT File Integrity..... 357
Configuration Frames.....358
Configuration Time..... 359
Configuration Debugging..... 361
Using Vivado Debug Cores.....363

Chapter 10: Configuration Solutions for Versal Devices.....370
Partial PDI Creation and Details.....370
Supported Boot Modes..... 371
Programming Image Compression..... 378
Segmented Configuration.....379

Chapter 11: Known Issues and Limitations.....387
Known Issues.....387
Known Limitations..... 388

Chapter 12: Hierarchical Design Flows..... 390

Appendix A: Supported Devices.....391

Appendix B: Additional Resources and Legal Notices **397**
Finding Additional Documentation 397
Support Resources 398
References 398
Training Resources 400
Revision History 400
Please Read: Important Legal Notices 401

Introduction

Dynamic Function eXchange (DFX) allows for the reconfiguration of modules within an active design. This flow requires the implementation of multiple configurations, which ultimately results in full bitstreams for each configuration and partial bitstreams for each reconfigurable module (RM). The number of configurations required varies by the number of modules that need to be implemented. However, all configurations use the same top-level, or static, placement, and routing results. These static results are exported from the initial configuration and imported by all subsequent configurations using checkpoints.

DFX is a comprehensive solution with many parts. These parts include the silicon capability for dynamic reconfiguration. They also include the AMD Vivado™ software flow that compiles designs from RTL to bitstream. They also include complementary features, such as IP. In this release, there is a mix of DFX and partial reconfiguration (PR) terminology. DFX represents the overall solution, and PR represents one component technology piece within that solution.

Complementary documentation, such as application notes, white papers, and videos, does not always use DFX terminology, but all documentation from 2020 onward uses DFX terminology.

This guide includes the following information:

- A description of DFX as implemented in the AMD Vivado™ Design Suite.
- An assumption of familiarity with FPGA design software, especially the Vivado Design Suite.

DFX supports the following products. For a complete list of supported devices, see Appendix A: Supported Devices.

- 7 series devices
  - DFX supports nearly all AMD Virtex™ 7, AMD Kintex™ 7, AMD Artix™ 7, and AMD Zynq™ 7000 SoC devices.

    ***Note*:** DFX does not support AMD Spartan™ 7 devices, Artix 7 A12T, and 7A25T devices.
- AMD UltraScale™ devices
  - Place and route, and bitstream generation is enabled for all production devices.

    ***Note*:** Expect higher memory usage on VU440 devices than on other devices. Memory usage can exceed 64 MB.
  - Bitstream generation is disabled by default for ES2 devices, but place and route can still be performed.

- AMD UltraScale+™ devices
  - You can run place and route and generate bitstreams for all production devices across all families. Artix UltraScale+, Kintex UltraScale+ Virtex UltraScale+ and Spartan UltraScale+. Zynq UltraScale+ MPSoC and RFSoC devices are also supported.
  - You can run place and route on many engineering silicon (ES1, ES2) versions of UltraScale+ devices. Bitstream generation is disabled by default on these devices.
- AMD Versal™ devices
  - You can use production DFX support on all Prime, AI Core, Premium, AI Edge, and HBM devices. See the device support table in Appendix A: Supported Devices for the complete list.
  - Some Versal devices are available in ES1 silicon. These devices are limited early access with license-gated implementation and parameter-gated device image generation. Contact AMD support if access to any of these devices are needed.

Related Information

Supported Devices

Navigating Content by Design Process

AMD Adaptive Computing documentation is organized around a set of standard design processes to help you find relevant content for your current development task. You can access the AMD Versal™ adaptive SoC design processes on the Design Hubs page. You can also use the Design Flow Assistant to better understand the design flows and find content that is specific to your intended design needs. This document covers the following design processes:

- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the AMD Vivado™ timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
  - Chapter 3: Vivado Software Flow
  - Chapter 4: Vivado Project Flow
  - Chapter 5: Design Considerations and Guidelines for All AMD Devices
  - Chapter 6: Design Considerations and Guidelines for 7 Series and Zynq Devices
  - Chapter 7: Design Considerations and Guidelines for UltraScale and UltraScale+ Devices
  - Chapter 8: Design Considerations and Guidelines for Versal Devices

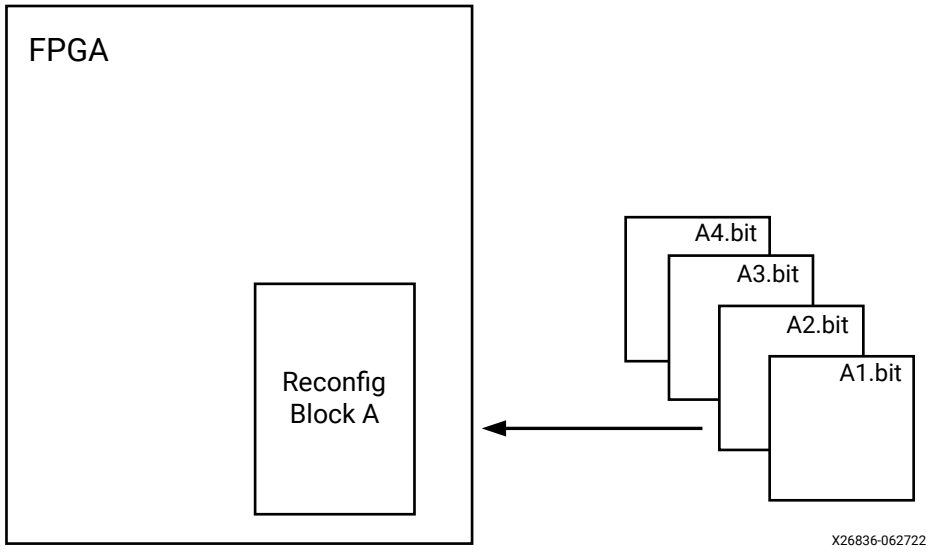
- **Board System Design:** Designing a PCB through schematics and board layout. Also involves power, thermal, and signal integrity considerations. Topics in this document that apply to this design process include:
  - Chapter 9: Configuring the Device
  - Chapter 10: Configuration Solutions for Versal Devices

Introduction to Dynamic Function eXchange

FPGA technology provides the flexibility of on-site programming and re-programming without going through re-fabrication with a modified design. Dynamic Function eXchange (DFX) takes this flexibility one step further, allowing the modification of an operating FPGA design by loading a dynamic configuration file, usually a partial BIT file. After a full BIT file configures the FPGA, partial BIT files can be downloaded to modify reconfigurable regions in the FPGA without compromising the integrity of the applications running on those parts of the device that are not being reconfigured.

The following figure illustrates the premise behind Dynamic Function eXchange.

Figure 1: **Basic Premise of Dynamic Function eXchange**



As shown, the function implemented in Reconfig Block A is modified by downloading one of several partial BIT files, `A1.bit`, `A2.bit`, `A3.bit`, or `A4.bit`. The logic in the FPGA design is divided into two different types, reconfigurable logic and static logic. The area of the block labeled "FPGA" represents static logic and the area of the block labeled "Reconfig Block A" represents reconfigurable logic. The static logic remains functioning and is unaffected by the loading of a partial BIT file. The reconfigurable logic is replaced by the contents of the partial BIT file.

There are many reasons why the ability to time multiplex hardware dynamically on a single FPGA is advantageous. These include:

- Reducing the size of the FPGA required to implement a given function, with consequent reductions in cost and power consumption
- Providing flexibility in the choices of algorithms or protocols available to an application
- Enabling new techniques in design security
- Improving FPGA fault tolerance
- Accelerating configurable computing
- Delivering updates (fixes and new features) to deployed systems

In addition to reducing size, weight, power and cost, Dynamic Function eXchange enables new types of FPGA designs that would be otherwise impossible to implement.

Terminology

The following terminology is specific to the Dynamic Function eXchange feature and is used throughout this document.

Block Design Containers (BDC)

Block Design Containers are hierarchical constructs in IP integrator that enables a block design to be placed within a block design. This feature is used to enable DFX flows in IP integrator for all architectures.



TIP: *This is the recommended flow for all Versal DFX designs.*

Bottom-Up Synthesis

Bottom-Up Synthesis is synthesis of the design by modules, whether in one project or multiple projects. In Vivado, bottom-up synthesis is referred to as out-of-context (OOC) synthesis. OOC synthesis generates a separate netlist (or DCP) per OOC module, and is required for Dynamic Function eXchange to ensure no optimization occurs across the module boundary. In OOC synthesis, the top-level (or static) logic is synthesized with black_box module definitions for each OOC module.

Configuration

A configuration is a complete design that has one RM for each reconfigurable partition (RP). There might be many configurations in a Dynamic Function eXchange FPGA project. Each configuration generates one full BIT file as well as one partial BIT file for each RM.

Configuration Frame

Configuration frames are the smallest addressable segments of the FPGA configuration memory space. Reconfigurable frames are built from discrete numbers of these lowest-level elements. In AMD devices, the base reconfigurable frames are one element (CLB, block RAM, DSP) wide by one clock region high. The number of resources in these frames vary by device family.

Internal Configuration Access Port (ICAP)

The internal configuration access port (ICAP) is essentially an internal version of the SelectMAP interface. For more information, see the *7 Series FPGAs Configuration User Guide* (UG470) or the *UltraScale Architecture Configuration User Guide* (UG570).

Media Configuration Access Port (MCAP)

The MCAP is dedicated link to the configuration engine from one specific PCIe® block per AMD UltraScale™ device. This entry point can be enabled when configuring the AMD PCIe IP.

Partition

A Partition is a logical section of the design, user-defined at a hierarchical boundary, to be considered for design reuse. A Partition is either implemented as new or preserved from a previous implementation. A Partition that is preserved maintains not only identical functionality but also identical implementation.

Partition Definition (PD)

This is a term used within the RTL project flow only. A Partition Definition defines a set of RMs that are associated with the module instance (or RP). A PD is applied to all instances of the module, and cannot be associated with a subset of module instances.

Partition Pin

Partition pins are the logical and physical connection between static logic and reconfigurable logic. The tools automatically create, place, and manage partition pins.

Partial Reconfiguration (PR)

Partial reconfiguration (PR) is the AMD silicon technology that enables users to modify a subset of logic in an operating FPGA design by downloading a partial bitstream. The overall solution name has changed to Dynamic Function eXchange, but the underlying capability of the silicon remains, so references to PR, especially in fundamental Tcl commands, can still be seen in Vivado.

Processor Configuration Access Port (PCAP)

The processor configuration access port (PCAP) is similar to the internal configuration access port (ICAP) and is the primary port used for configuring a Zynq 7000 SoC device. For more information, see the *Zynq 7000 SoC Technical Reference Manual* (UG585).

Programmable Unit (PU)

This is the minimum required resources for reconfiguration. The size of a PU varies by resource type. Because adjacent sites share a routing resource (or Interconnect tile) in the UltraScale architecture, a PU is defined in terms of pairs.

Reconfigurable Frame

Reconfigurable frames (in all references other than "configuration frames" in this guide) represent the smallest reconfigurable region within an FPGA. Bitstream sizes of reconfigurable frames vary depending on the types of logic contained within the frame.

Reconfigurable Logic

Reconfigurable logic is any logical element that is part of an RM. These logical elements are modified when a partial BIT file is loaded. Many types of logical components can be reconfigured such as LUTs, flip-flops, block RAM, and DSP blocks.

Reconfigurable Module

An RM is the netlist or HDL description that is implemented within an RP. Multiple RMs can exist for an RP.

Reconfigurable Partition

RP is an attribute set on an instantiation that defines the instance as reconfigurable. The RP is the level of hierarchy within which different RMs are implemented. Tcl commands such as `opt_design`, `place_design` and `route_design` detect the `HD.RECONFIGURABLE` property on the instance and process it correctly.

Static Logic

Static logic is any logical element that is not part of an RP. The logical element is never partially reconfigured and is always active when RPs are being reconfigured. Static logic is also known as top-level logic.

Static Design

The static design is the part of the design that does not change during partial reconfiguration. The static design includes the top-level and all modules not defined as reconfigurable. The static design is built with static logic and static routing.

Design Considerations

Dynamic Function eXchange is an expert flow within the Vivado Design Suite. The following requirements and expectations need to be understood before embarking on a DFX project.

Design Requirements and Guidelines

Following are the design requirements and guidelines when using DFX:

- Floorplanning is required to define reconfigurable regions, per element type.
  - For 7 series, vertically align Pblocks with frame/clock region boundaries. This produces the best results and allows RESET_AFTER_RECONFIG to be enabled.
  - For UltraScale and beyond, the floorplanning is more flexible. AMD recommends stopping the Pblock short of frame/clock region boundaries to allow for expanded routing, which can greatly improve routability and quality.
  - Horizontal alignment rules also apply. See Create a Floorplan for the Reconfigurable Region for more information.
  - Automatic expansion for routing resources is done for all UltraScale, UltraScale+, and Versal device targets.
- Bottom-up/OOC synthesis (to create multiple netlist/DCP files) and management of RM netlist files is the responsibility of the user.
  - For third party synthesis tools, I/O insertion must be disabled.
  - For Vivado OOC synthesis, I/O insertion is automatically disabled in the out_of_context mode.
- Standard timing constraints are supported, and additional timing budgeting capabilities are available if needed.
- A unique set of design rule checks (DRCs) has been established to help ensure successful design completion.
- A DFX design must consider the initiation of partial reconfiguration as well as the delivery of partial BIT or PDI files, either within the target device or as part of the system design.
- Multiple design flow environments are available for processing DFX designs. Versal device designs must use the block design container flow within IP integrator to manage CIPS and NoC IP, but for FPGA and SoC designs, RTL-based design flows can be used.
- The Vivado Design Suite includes support for the Dynamic Function eXchange Controller IP. This customizable IP manages the core tasks for partial reconfiguring any AMD FPGA. The core receives triggers from hardware or software, manages handshaking and decoupling tasks, fetches partial bitstreams from memory locations, and delivers them to the ICAP. More information on the DFX Controller IP is available on the Xilinx.com website.

- An RP must contain a super set of all pins to be used by the varying reconfigurable modules (RM) implemented for the partition. If an RM uses different inputs or outputs from another RM, the resulting RM inputs or outputs might not connect inside of the RM. The tools handle this by inserting a LUT1 buffer within the RM for all unused inputs and outputs. Additional information on LUT1 buffer insertion, including specific use cases where necessary, can be found on the design blog Vivado DFX - Handling Boundary Connections for a DFX design. The output LUT1 is tied to a constant value and the value of the constant can be controlled by `HD.PARTPIN_TIEOFF` property on the unused output pin. For more information on this property refer to Black Boxes.
- Black boxes are supported for bitstream generation. See Black Boxes for details about how to tie off ports with constant values.
- For user reset signals, determine if the logic inside the RM is level or edge sensitive. If the reset circuit is edge sensitive (as it may be in some IP such as FIFOs), the RM reset should not be applied until after reconfiguration is complete.
- DFX designs are compatible with the AMD Isolation Design Flow (IDF) for Zynq UltraScale+ MPSoC devices. For more information on solution details, please consult *Isolation Design Flow for UltraScale+ FPGAs and Zynq UltraScale+ MPSoCs* (XAPP1335).

Related Information

Create a Floorplan for the Reconfigurable Region
Black Boxes

Design Performance

Performance metrics vary from design to design, and the best results are achieved if you follow the Hierarchical Design techniques suggested in Hierarchical Design Flows. You can find additional design recommendations in the *UltraFast Design Methodology Guide for FPGAs and SoCs* (UG949).

However, the additional restrictions that are required for silicon isolation are expected to have an impact on most designs. The application of partial reconfiguration rules, such as routing containment, exclusive placement, and no optimization across RM boundaries, means that the overall density and performance is lower for a DFX design than for the equivalent flat design. The overall design performance for DFX designs varies from design to design, based on factors such as the number of RPs, the number of interface pins to these partitions, and the size and shape of Pblocks.

Any potential Dynamic Function eXchange design must have extra timing slack and resource overhead before considering this solution. See the Building Up Implementation Requirements section for more information on evaluating a design for DFX.

Related Information

Hierarchical Design Flows
Building Up Implementation Requirements

Design Criteria

Some component types can be reconfigured and some cannot.

- For 7 series devices, the component rules are as follows:
  - Reconfigurable resources include CLB, block RAM, and DSP component types as well as routing resources.
  - Clocks and clock modifying logic cannot be reconfigured, and therefore must reside in the static region.
    - Includes BUFG, BUFR, MMCM, PLL, and similar components
  - The following components cannot be reconfigured, and therefore must reside in the static region:
    - I/O and I/O related components (ISERDES, OSERDES, IDELAYCTRL)
    - Serial transceivers (MGTs) and related components
    - Individual architecture feature components (such as BSCAN, STARTUP, ICAP, XADC)
- For UltraScale and UltraScale+ devices, the list of reconfigurable component types is more extensive:
  - CLB, block RAM, and DSP component types as well as routing resources
  - Clocks and clock modifying logic, including BUFG, MMCM, PLL, and similar components
  - I/O and I/O related components (ISERDES, OSERDES, IDELAYCTRL)

    ***Note*:** The types of changes for I/O components is limited. See I/O Rules for more information.
  - Serial transceivers (MGTs) and related components
  - PCIe, CMAC, Interlaken, and SYSMON blocks
  - Bitstream granularity of these new components require that certain rules are followed. For example, partial reconfiguration of I/O require that the entire bank, plus all clocking resources in that frame are reconfigured together.
  - Only the configuration components (such as BSCAN, STARTUP, ICAP, and FRAME_ECC) must remain in the static portion of the design.
- For Versal devices, in addition to all elements in the programmable logic supported for UltraScale+, the Network on Chip (NoC) is dynamically reconfigurable.
- Global clocking resources to RPs are limited, depending on the device and on the clock regions occupied by these RPs.

- IP restrictions may occur due to components used to implement the IP or due to connections required by the IP. Examples include:
  - Vivado Debug Cores (See Using Vivado Debug Cores for more information on using debug cores inside of RMs)
  - IP modules with embedded global buffers or I/O (7 series only)
  - Memory IP controller (MMCM and BSCAN)
- RMs must be initialized to ensure a predictable starting condition after reconfiguration. For all devices other than 7 series, GSR is automatically applied after DFX completes. For 7 series devices, GSR can be turned on, after meeting Pblock requirements, with the `RESET_AFTER_RECONFIG` Pblock property.
- Decoupling logic is highly recommended to disconnect the reconfigurable region from the static portion of the design during the act of partial reconfiguration.
  - GSR events hold all logic inside the RM in reset until configuration completes. However, RM outputs can be random and all downstream logic should be decoupled. For 7 series, if `RESET_AFTER_RECONFIG` is not used, additional decoupling of clocks and inputs can be required to prevent unintended capture of erroneous data of during reconfiguration (for example spurious write to memory).
  - The Vivado Design Suite includes the DFX Decoupler IP. This IP allows users to easily insert multiplexers to efficiently decouple AXI4-Lite, AXI4-Stream, and custom interfaces. More information on the DFX Decoupler IP is available on the Xilinx.com website.
- An RP must be floorplanned with a Pblock, so the module must be a block that can be physically isolated and meet timing. If the module is complete, it is recommended to run this design through a non-DFX flow to get an initial evaluation of placement, routing, and timing results. If the design has issues in a non-DFX flow, these should be resolved before moving on to the DFX flow.
- Optimize an RP's interface as much as possible. An excessive number of interface pins on an RP can cause timing and routing issues. This is especially true if the partition pins are densely placed. This can happen for two reasons:
  1. RP Pblock is relatively small compared to the number of partition pins.
  2. All the partition pins are placed in a small area due to static connections.

  Consider the RP interface when designing and floorplanning for DFX.
- Virtex 7 SSI devices (7V2000T, 7VX1140T, 7VH870T, 7VH580T) have two fundamental requirements. These requirements are:
  - Reconfigurable regions must be fully contained within a single SLR. This ensures that the global reset events are properly synchronized across all elements in the RM, and that all super long lines (SLL) are contained within the static portion of the design. SLL are not partially reconfigurable.

    - If the initial configuration of a 7 series SSI device is done through an SPIx1 interface, partial bitstreams must be delivered to the ICAP located on the SLR where the RP exists, or to an external port, such as JTAG. If the initial configuration is done through any other configuration port, the master ICAP can be used as the delivery port for partial bitstreams.
- UltraScale devices have a new requirement related to partial reconfiguration events. Before a partial bitstream for a new RM is loaded, the current RM must be "cleared" to prepare for reconfiguration. UltraScale+ devices do not have this limitation. For more information, see Summary of BIT Files for UltraScale Devices.
- Dedicated encryption support for partial bitstreams is available natively. See Known Limitations for specific unsupported use cases for UltraScale devices.
- Devices can use a per-frame CRC checking mechanism, enabled by `write_bitstream`, to ensure each frame is valid before loading.
- Optimization across the DFX boundary is prohibited by the implementation tools. Often the WNS paths in a DFX design are high fanout control/reset signals that cross the RP boundary. Avoid high fanout signals crossing the RP boundary because the drivers cannot be replicated. To allow the tools maximum flexibility of optimization/replication, consider the following:
    - For inputs to the RP, make the signal crossing the RP boundary a single fanout net, and register the signal inside the RM before the fanout. This can be replicated as necessary inside the RM (or put on global resources).
    - For outputs, again make the signal crossing the DFX boundary a single fanout net. Register the signal in static before the fanout for replication/optimization.
- For design with multiple RPs, AMD recommends not having direct connections between two RPs. This includes connections that go through asynchronous static logic (not registered in static). If direct connections exist between two RPs, all possible configurations must be verified in static timing analysis to ensure timing is met across these interfaces. This can be done for closed systems that are fully owned and maintained by a single user, but can be impossible to verify for designs where different RMs are developed by multiple users. Adding a synchronous endpoint in static ensures timing is always met on any configuration, as long as the configuration where the RM was implemented met timing.

Dynamic Function eXchange is a powerful capability within AMD devices, and understanding the capabilities of the silicon and software is instrumental to success. While trade-offs must be recognized and considered during the development process, the overall result is a more flexible implementation of your FPGA design.

Related Information

I/O Rules
Using Vivado Debug Cores
Summary of BIT Files for UltraScale Devices
Known Limitations

Dynamic Function eXchange Licensing

Dynamic Function eXchange is a feature included in the Vivado Design Suite. No specific license code is required to utilize this feature across all editions of Vivado.

For versions prior to 2019.1, a Partial Reconfiguration license is included with every System Edition and Design Edition seat, and can be purchased for Standard Edition seats.

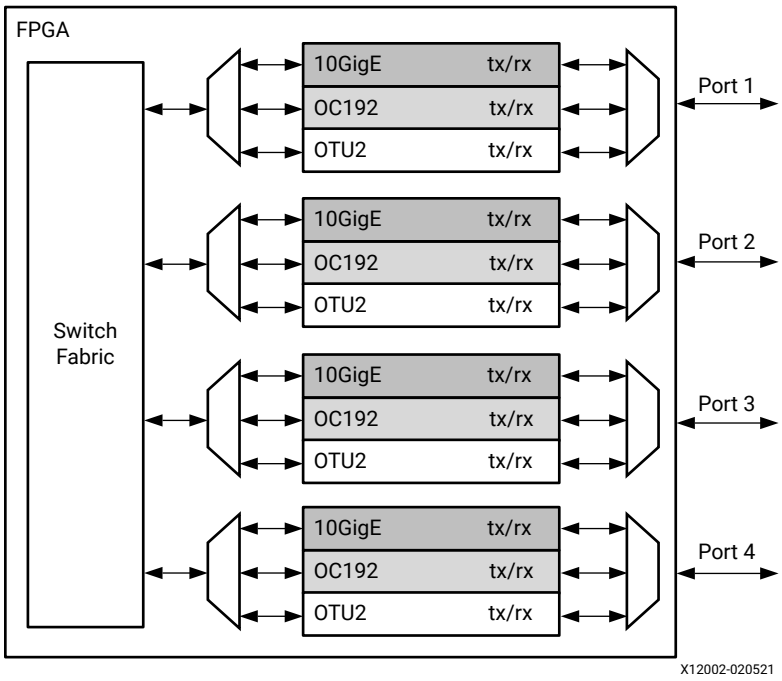
Common Applications

The basic premise of Dynamic Function eXchange (DFX) is that the device hardware resources can be time-multiplexed similar to the ability of a microprocessor to switch tasks. Because the device is switching tasks in hardware, it has the benefit of both flexibility of a software implementation and the performance of a hardware implementation. Several different scenarios are presented here to illustrate the power of this technology.

Networked Multiport Interface

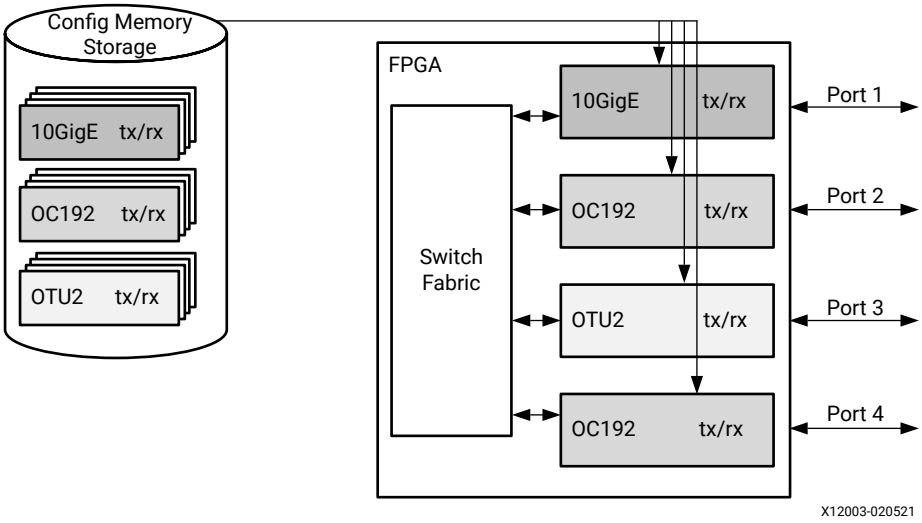
Dynamic Function eXchange optimizes traditional FPGA applications by reducing size, weight, power, and cost. Time-independent functions can be identified, isolated, and implemented as reconfigurable modules (RM) and swapped in and out of a single device as needed. A typical example is a 40G OTN muxponder application. The ports of the client side of the muxponder can support multiple interface protocols. However, it is not possible for the system to predict which protocol will be used before the FPGA is configured. To ensure that the FPGA does not have to be reconfigured and thus disable all ports, every possible interface protocol is implemented for every port, as illustrated in the following figure.

Figure 2: **Network Switch Without Partial Reconfiguration**



This is an inefficient design because only one of the standards for each port is in use at any point in time. Dynamic Function eXchange enables a more efficient design by making each of the port interfaces an RM, as shown in the following figure. This also eliminates the MUX elements required to connect multiple protocol engines to one port.

Figure 3: **Network Switch With Partial Reconfiguration**



A wide variety of designs can benefit from this basic premise. Software defined radio (SDR), for example, is one of many applications that has mutually exclusive functionality, and which sees a dramatic improvement in flexibility and resource usage when this functionality is multiplexed.

There are additional advantages with a dynamically reconfigurable design other than efficiency. In the first example, a new protocol can be supported at any time without affecting the static logic, the switch fabric in this example. When a new standard is loaded for any port, the other existing ports are not affected in any way. Additional standards can be created and added to the configuration memory library without requiring a complete redesign. This allows greater flexibility and reliability with less down time for the switch fabric and the ports. A debug module could be created so that if a port was experiencing errors, an unused port could be loaded with analysis/correction logic to handle the problem real-time.

In the second example, a unique partial BIT file must be generated for each unique physical location that could be targeted by each protocol. Partial BIT files are associated with an explicit region on the device. In this example, sixteen unique partial BIT files to accommodate four protocols for four locations.

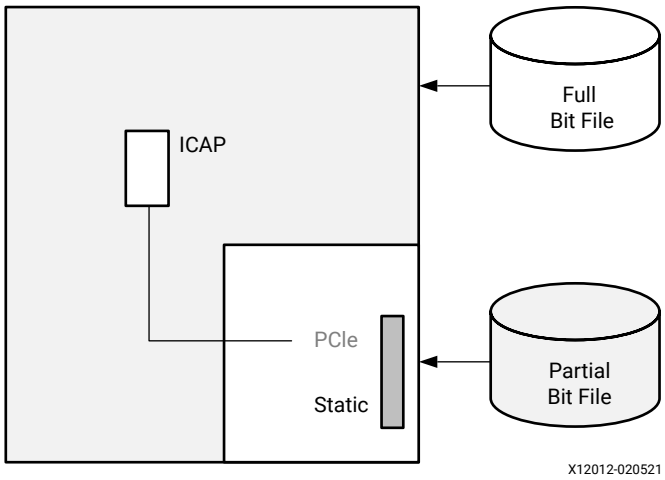
Configuration by Means of Standard Bus Interface

Dynamic Function eXchange can create a new configuration port using an interface standard more compatible with the system architecture. For example, the FPGA could be a peripheral on a PCIe® bus and the system host could configure the FPGA through the PCIe connection. After power-on reset the FPGA must be configured with a full BIT file. However, the full BIT file might only contain the PCIe interface and connection to the internal configuration access port (ICAP).

Bitstream compression can be used to reduce the size and therefore configuration time of this initial device load, helping the FPGA configuration meet PCIe enumeration specifications.

The system host could then configure the majority of the FPGA functionality with a partial BIT file downloaded through the PCIe port as shown in the following figure. An example of fast configuration over PCIe is shown in *Fast Partial Reconfiguration Over PCI Express Application Note* (XAPP1338), with an example targeting AMD UltraScale+™ FPGAs included.

Figure 4: **Configuration by Means of PCIe Interface**



The PCIe standard requires the peripheral (the FPGA in this case) to acknowledge any requests even if it cannot service the request. Reconfiguring the entire FPGA would violate this requirement. Because the PCIe interface is part of the static logic, it is always active during the dynamic reconfiguration process, thus ensuring that the FPGA can respond to PCIe commands even during reconfiguration.

Tandem Configuration is a related solution that at first glance appears to be the same as is shown here. However, the solution using Dynamic Function eXchange differs from Tandem Configuration in two regards:

- The configuration process with DFX is a full device configuration, made smaller and faster through compression, followed by a partial bitstream that overwrites the black box region to complete the overall configuration. Tandem Configuration is a two-stage configuration where each configuration frame is programmed exactly once.
- Tandem Configuration for 7 series devices does not permit dynamic reconfiguration of the user application. Using DFX, the dynamic region can be reloaded with different user applications or field updates. Tandem Configuration for UltraScale and UltraScale+ devices does permit Field Updates and compatibility with DFX in general. The overall flow is Tandem Configuration for a two-stage initial load, followed by partial reconfiguration to dynamically modify the user application.

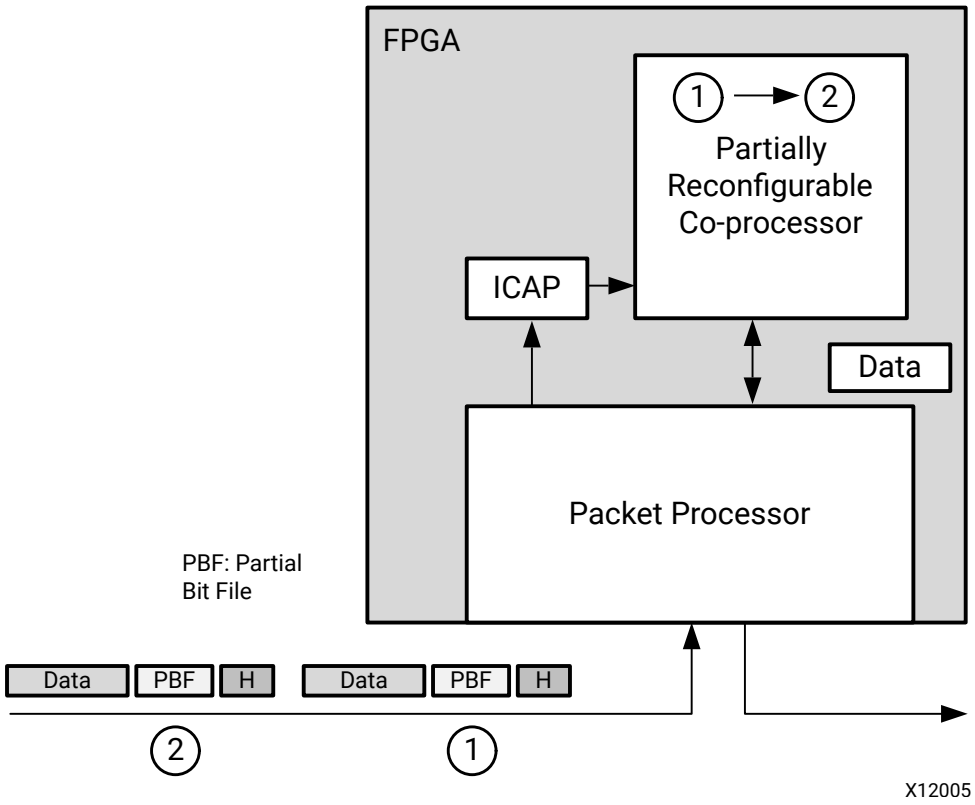
Tandem Configuration is designed to be a specific solution for a specific goal: fast configuration of a PCIe endpoint to meet enumeration requirements. For more information, see the following manuals:

- *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054)
- *Virtex 7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023)
- *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156)
- *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)

Dynamically Reconfigurable Packet Processor

A packet processor can use Dynamic Function eXchange to change its processing functions quickly, based on the packet types received. In the following figure, a packet has a header that contains the partial BIT file, or a special packet contains the partial BIT file. After the partial BIT file is processed, it is used to reconfigure a co-processor in the FPGA. This is an example of the FPGA reconfiguring itself based on the data packet received instead of relying on a predefined library of partial BIT files.

Figure 5: **Dynamically Reconfigurable Packet Processor**

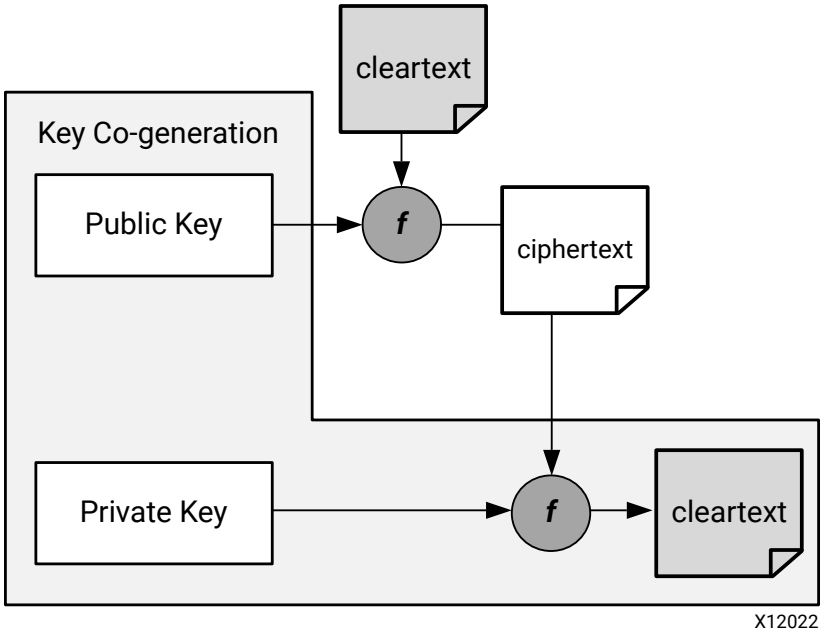


Asymmetric Key Encryption

There are some new applications that are not possible without Dynamic Function eXchange. A very secure method for protecting the FPGA configuration file can be architected when Dynamic Function eXchange and asymmetric cryptography are combined. (See Public-key cryptography for asymmetric cryptography details.)

In the following figure, the group of functions in the shaded box can be implemented within the physical package of the FPGA. The cleartext information and the private key never leave a well-protected container.

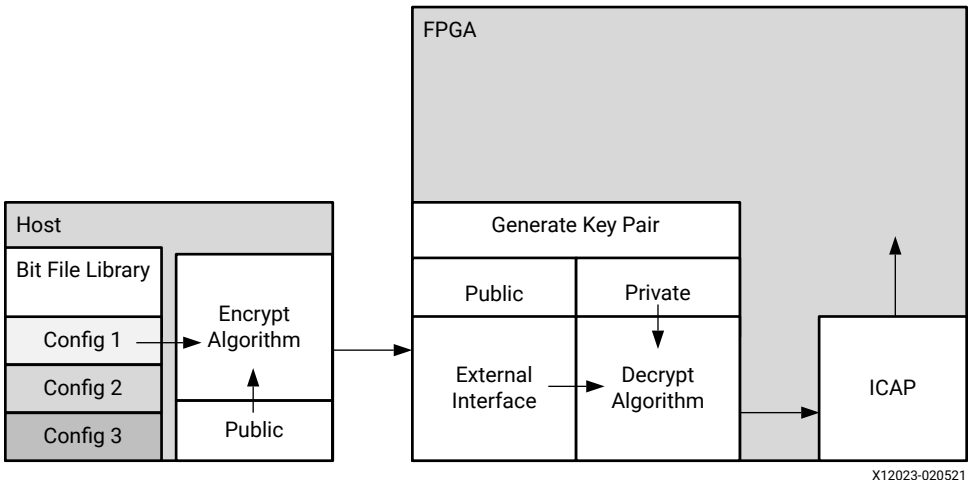
Figure 6: **Asymmetric Key Encryption**



In a real implementation of this design, the initial BIT file is an unencrypted design that does not contain any proprietary information. The initial design only contains the algorithm to generate the public-private key pair and the interface connections between the host, FPGA and ICAP.

After the initial BIT file is loaded, the FPGA generates the public-private key pair. The public key is sent to the host which uses it to encrypt a partial BIT file. The encrypted partial BIT file is downloaded to the FPGA where it is decrypted and sent to the ICAP to partially reconfigure the FPGA, as shown in the following figure.

Figure 7: **Loading an Encrypted Partial Bit File**



The partial BIT file could be the vast majority of the FPGA design with the logic in the static design consuming a very small percentage of the overall FPGA resources.

This scheme has several advantages:

- The public-private key pair can be regenerated at any time. If a new configuration is downloaded from the host it can be encrypted with a different public key. If the FPGA is configured with the same partial BIT file, such as after a power-on reset, a different public key pair is used even though it is the same BIT file.
- The private key is stored in SRAM. If the FPGA ever loses power the private key no longer exists.
- Even if the system is stolen and the FPGA remains powered, it is extremely difficult to find the private key because it is stored in the general purpose FPGA programmable logic. It is not stored in a special register. You could manually locate each register bit that stores the private key in physically remote and unrelated regions.

Summary

In addition to reducing size, weight, power and cost, Dynamic Function eXchange enables new types of FPGA designs that would otherwise be impossible to implement.

Vivado Software Flow

The AMD Vivado™ Dynamic Function eXchange (DFX) design flow is similar to a standard design flow, with some notable differences. The implementation software automatically manages the low-level details to meet silicon requirements. You must provide guidance to define the design structure and floorplan. The following steps summarize processing a DFX design:

1. Synthesize the static and reconfigurable modules (RM) separately. See Synthesis for more information.
2. Create physical constraints (Pblocks) to define the reconfigurable regions. See Create a Floorplan for the Reconfigurable Region for more information.
3. Set the `HD.RECONFIGURABLE` property on each reconfigurable partition (RP). See Define a Module as Reconfigurable for more information.
4. Implement a complete design (static and one RM per RP) in context. See Implementation for more information.
5. Save a design checkpoint for the full routed design. See Implementation for more information.
6. Remove RMs from this design and save a static-only design checkpoint. See Implementation for more information.
7. Lock the static placement and routing. See Preserving Implementation Data for more information.
8. Add new RMs to the static only design and implement this new configuration, saving a checkpoint for the full routed design.
9. Repeat Step 8 until all RMs are implemented.
10. Run a verification utility (`pr_verify`) on all configurations. See Verifying Configurations for more information.
11. Create bitstreams for each configuration. See Bitstream Generation for more information.

Related Information

Synthesis
Create a Floorplan for the Reconfigurable Region
Define a Module as Reconfigurable
Implementation
Preserving Implementation Data
Verifying Configurations
Bitstream Generation

Dynamic Function eXchange Commands

The DFX flows support project flows and non-project batch flows. The next chapter describes project flows in detail. This chapter covers the underlying foundations that the project flow uses.

The *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947) provides example scripts for the non-project flow. It also provides step-by-step instructions for setting up the flows.

The following sections describe a few specialized commands and options needed for the DFX flows. The sections also describe how to use these commands to run a DFX flow. For more information on individual commands, refer to the *Vivado Design Suite Tcl Command Reference Guide* (UG835).

Synthesis

Synthesizing a partially reconfigurable design does not require any special commands, but does require bottom-up synthesis. There are currently no unsupported commands for synthesis, optimization, or implementation.

These synthesis tools are supported:

- XST (supported for 7 series only)
- Synplify
- Vivado Synthesis

IMPORTANT! *Bottom-up synthesis refers to a synthesis flow in which each module has its own synthesis project. This generally involves turning off automatic I/O buffer insertion for the lower level modules.*

This document only covers the Vivado synthesis flow.

Synthesizing the Top Level

You must have a top-level netlist with a black box for each reconfigurable partition (RP). This requires the top-level synthesis to have module or entity declarations for the partitioned instances, but no logic; the module is empty.

The top-level synthesis infers or instantiates I/O buffers on all top level ports. For more information on controlling buffer insertion, see the Setting a Bottom-Up Out-of-Context Flow topic in the *Vivado Design Suite User Guide: Synthesis* (UG901).

```
synth_design -flatten_hierarchy rebuilt -top <top_module_name> -part <part>
```

Synthesizing Reconfigurable Modules

Because each RM must be instantiated in the same black box in the static design, the different versions must have identical interfaces. The name of the block must be the same in each instance, and all the properties of the interfaces (names, widths, direction) must also be identical. Each configuration of the design is assembled like a flat design.

To synthesize a RM, turn off all buffer insertions. You can do so in Vivado Synthesis using the `synth_design` command with the `-mode out_of_context` switch:

```
synth_design -mode out_of_context -flatten_hierarchy rebuilt -top
<reconfig_module_name> -part <part>
```

Table 1: **synth_design Options**

Command Option	Description
`-mode out_of_context`	Prevents I/O insertion for synthesis and downstream tools. The `out_of_context` mode is saved in the checkpoint if `write_checkpoint` is issued.
`-flatten_hierarchy rebuilt`	There are several values allowed for `-flatten_hierarchy`, but `rebuilt` is the recommended setting for DFX flows.
`-top`	This is the module/entity name of the module being synthesized.
`-part`	This is the AMD part being targeted (for example, `xc7k325tffg900-3`)

The `synth_design` command synthesizes the design and stores the results in memory. In order to write the results out to a file, use:

```
write_checkpoint <file_name>.dcp
```

It is recommended to close the design in memory after synthesis, and run implementation separately from synthesis.

Reading Design Modules

If there is currently no design in memory, you must load a design. This can be done in a variety of ways, for either the static design or for RM. After the configurations are implemented, checkpoints are exclusively used to read in placed and routed module databases.

Method 1: Add and Link Files

This is the recommended method to load and link all design sources in the most explicit and thorough manner. The following steps pull in all necessary design sources and define the RP boundaries.

1. Create a new project in memory. While this allows you to select a target device, the project is not saved.

```
create_project -part <part> -in_memory
```

2. Add all the design sources. This can include multiple checkpoints for static or reconfigurable logic, including lower-level RM sources.

```
add_files <top>.dcp
add_files <rp1_rmA_top>.dcp
add_files <rp1_rmA_lower>.dcp
add_files <rp2_rmA_top>.dcp
```

3. Use the `SCOPED_TO_CELLS` property to define relationships between levels of hierarchy.

```
set_property SCOPED_TO_CELLS {<RP1_module_instance>} [get_files
<rp1_rmA_top>.dcp]
set_property SCOPED_TO_CELLS {<RP1_lower_module_instance>} [get_files
<rp1_rmA_lower>.dcp]
set_property SCOPED_TO_CELLS {<RP2_module_instance>} [get_files
<rp2_rmA_top>.dcp]
```

4. Link the design together, defining all RPs.

```
link_design -top <top> -part <part> -reconfig_partitions
{<RP1_module_instance> <RP2_module_instance>}
```

Table 2: **link_design Options**

Command Option	Description
`-part`	This is the AMD part being targeted (for example, `xc7k325tffg900-3`)
`-top`	This is the module/entity name of the module being implemented. This switch can be omitted if `set_property top <top_module_name> [current_fileset]` is issued prior to `link_design`.
`-reconfig_partitions <args>`	Specify a list of RPs to load while opening the design. The specified RPs are then marked with the HD.RECONFIGURABLE property for proper handling in the design.
`-pr_config <arg>`	For the project-based design flow only. This option specifies the PR Configuration to apply while opening the design.

Method 2: Read Netlist Design

This approach should be used when modules have been synthesized by tools other than Vivado synthesis.

```
read_edif <top>.edf/edn/ngc
read_edif <rp1_a>.edf/edn/ngc
read_edif <rp2_a>.edf/edn/ngc
link_design -top <top_module_name> -part <part>
```

Method 3: Open/Read Checkpoint

If the static (top-level) design has synthesis or implementation results stored as a checkpoint, it can be loaded using the `open_checkpoint` command. This command reads in the static design checkpoint and opens it in active memory:

```
open_checkpoint <file>
```

If the checkpoint is for the complete netlist of a RM (that is, not for static), the instance name can be specified using `read_checkpoint -cell`. If the checkpoint is a post-implementation checkpoint, the additional `-strict` option must be used. This option can also be used with a post-synthesis checkpoint to ensure exact port matching. To read in a checkpoint in a RM, the top-level design must be open and have a black box for the specified cell. Then the following command can be specified:

```
read_checkpoint -cell <cellname> <file> [-strict]
```

Table 3: **`read_checkpoint` Switches**

Switch Name	Description
`-cell`	Specifies the full hierarchical name of the RM.
`-strict`	Requires exact ports match for replacing a cell, and checks that part, package, and speed grade values are identical. Should be used when restoring implementation data.
`<file>`	Specifies the full or relative path to the checkpoint (DCP) to be read in.

CAUTION! *Do not use this method if the synthesized checkpoint has underlying modules that are not included. The `read_checkpoint -cell` approach does not support nesting. Use the `link_design` approach instead in Method 1.*

CAUTION! *Any Tcl variable pointing to design objects becomes invalid after subsequent `read_checkpoint -cell` commands. The content of those variables needs to be rebuilt prior to a second call to `read_checkpoint -cell`. Failure to do so could result to unwanted behavior (or even crashes) due to referencing objects that no longer exist.*

Related Information

Method 1: Add and Link Files

Method 4: Open Checkpoint/Update Design

This is useful when the synthesis results are in the form of a netlist (EDF or EDN), but static has already been implemented. The following example shows the commands for the second configuration in which this is true.

```
open_checkpoint <top>.dcp
lock_design -level routing
update_design -cells <rp1> -from_file <rp1_b>.{edf/edn}
update_design -cells <rp2> -from_file <rp2_b>.{edf/edn}
```

Adding Reconfigurable Modules with Sub-Module Netlists

If a RM has sub-module netlists, it can be difficult for the Vivado tools to process the sub-module netlists. This is because in the DFX flow the RM netlists are added to a design that is already open in memory. This means the `update_design -cells` command must be used, which requires the cell name for every EDIF file, which can be troublesome to get.

There are two ways to make loading RM sub-module netlists easier in the Vivado Design Suite.

Method 1: Create a Single RM Checkpoint (DCP)

Create an RM checkpoint (DCP) that includes all netlists. Use `add_files` to add all of the EDIF (or NGC) files, and use `link_design` to resolve the EDIF files to their respective cells. Here is an example of the commands used in this process:

```
add_files [list rm.edf ip_1.edf  ip_n.edf]
# Run if RM XDC exists
add_files rm.xdc
link_design -mode out_of_context -top <rm_module> -part <part>
write_checkpoint rm_v#.dcp
close_project
```



IMPORTANT! *Using this methodology to combine/convert a netlist into a DCP is the recommended way to handle an RM that has one or more NGC source files as well.*

Then this newly-created RM checkpoint can be used in the DFX flow. In the commands below, the single `read_checkpoint -cell` command replaces what could be many `update_design -cell` commands.

```
add_files static.dcp
link_design -top <top> part <part>
lock_design -level routing
read_checkpoint -cell <rm_inst> rm_v#.dcp
```

Method 2: Place the Sub-Module Netlists in the Same Directory as the RM's Top-Level Netlist

When the top-level RM netlist is read into the DFX design using `update_design -cell`, make sure that all sub-module netlists are in the same directory as the RM top-level netlist. In this case, the lower-level netlists do not need to be specified, but they are picked up automatically by the `update_design -cells` command. This is less explicit than Method 1, but requires fewer steps. In this case the commands to load the RM netlist would look like the following:

```
add_files static.dcp
link_design -top <top> part <part>
lock_design -level routing
update_design -cells <rm_inst> -from_file rm_v#.edf
```

In the last (`update_design`) command above, the lower-level netlists are picked up automatically if they are in the same directory as `rm_v#.edf`.

Reading Design Constraints

New constraints can be applied for each configuration at various points in the flow. If an RM is read in as a DCP, then any constraints stored in the DCP are automatically applied. Additionally, the `read_xdc` command can be used to apply constraints scoped to the top-level, or to the specific cell (using `-cell` switch). If constraint are expected to directly or indirectly affect the RM, then the RM must be resolved (not a black box) prior to reading in the new constraints. Otherwise, the constraints may be dropped or not correctly propagated in the constraint system. Because Static is only placed and routed in the initial configuration, all constraints for subsequent configurations (where Static is locked) should be focused strictly on the RP regions being implemented.

Implementation

Because the DFX flow allows for various configurations in hardware, multiple implementation runs are required. Each implementation of a DFX design is referred to as a *configuration*. Each module of the design (static or RM) can be implemented or imported (if previously implemented). Implementation results for the static design must be consistent for each configuration, so that the design is implemented in one configuration, and then imported in subsequent configurations. Additional configurations can be constructed by importing static, and implementing or importing each RM.

There are no restrictions to the support of implementation commands or options for DFX, but certain optimizations and sub-routines are not done if they oppose the fundamental requirements of partial reconfiguration. The following list of commands can be run after the logical design is loaded (using `link_design` or `open_checkpoint`):

```
# Run if all constraints are not already loaded
read_xdc
# Optional command
opt_design
place_design
# Optional command
phys_opt_design
route_design
```

Preserving Implementation Data

In the DFX flow, it is a requirement to lock down the placement and routing results of the static logic from the first configuration for all subsequent configurations. The static implementation of the first configuration must be saved as a checkpoint. When the checkpoint is read for subsequent configurations, the placement and routing must be locked, to ensure that the static design remains completely identical from configuration to configuration. To lock the placement and routing of an imported checkpoint (static or reconfigurable), the `lock_design` command is used.

```
lock_design -level [logical|placement|routing] [cell_name]
```

When locking down the static logic with the above command, the optional `[cell_name]` can be omitted.

```
lock_design -level routing
```

To lock the results of an imported RM, the full hierarchical name should be specified within the post-implementation checkpoint:

```
lock_design -level routing u0_RM_instance
```

For Dynamic Function eXchange, the only supported preservation level is `routing`. Other preservation levels are available for this command, but they must only be used for other Hierarchical Design flows.

Dynamic Function eXchange Constraints and Properties

There are properties and constraints unique to the Dynamic Function eXchange flow. These initiate DFX-specific implementation processing and apply specific characteristics in the partial bitstreams. The four areas for constraints and properties for DFX are:

Table 4: **Constraints and Properties**

Constraints and Properties	Necessity
Defining a module as reconfigurable	Required
Creating a floorplan for the reconfigurable region	Required
Applying reset after reconfiguration	Optional, but highly recommended
Turn on visualization scripts	Optional

Define a Module as Reconfigurable

In order to implement a DFX design, it is required to specify each RM as such. To do this you must set a property on the top level of each hierarchical cell that is going to be reconfigurable. For example, take a design where one RP named `inst_count` exists, and it has two RMs, `count_up` and `count_down`. The following command must be issued prior to implementation of the first configuration.

```
set_property HD.RECONFIGURABLE TRUE [get_cells inst_count]
```

This initiates the Dynamic Function eXchange features in the software that are required to successfully implement a DFX design. The `HD.RECONFIGURABLE` property implies a number of underlying constraints and tasks:

- Sets `DONT_TOUCH` on the specified cell and its interface nets. This prevents optimization across the boundary of the module.
- Sets `EXCLUDE_PLACEMENT` on the cell's Pblock. This prevents static logic from being placed in the reconfigurable region.
- Sets `CONTAIN_ROUTING` on the cell's Pblock. This keeps all the routing for the RM within the bounding box.
- Enables special code for DRCs, clock routing, etc.

`IS_DFX` is a read-only property automatically set by the tool and associated with reconfigurable partition Pblocks in DFX designs. This property is set to `TRUE`, indicating that the Pblock is associated with a reconfigurable partition.

This property is TRUE on the reconfigurable Pblock that have `EXCLUDE_PLACEMENT` and `CONTAIN_ROUTING` enabled.

Create a Floorplan for the Reconfigurable Region

Each RP is required to have a Pblock to define the physical resources available for the RM. Because this Pblock is set on a RP, these restrictions and requirements apply:

- The Pblock must contain only valid reconfigurable element types. The region can overlap other site types, but these other sites must not be included in the `resize_pblock` commands.
- Multiple Pblock rectangles for each component type can be used to create the RP region, but for the greatest routability, they should be contiguous. Gaps to account for non-reconfigurable resources are permitted, but in general, the simpler the overall shape, the easier the design is to place and route.
- If using the `RESET_AFTER_RECONFIG` property for 7 series devices, the Pblock height must align to clock region boundaries. See Apply Reset After Reconfiguration for more details.
- The width and composition of the Pblock must not split interconnect columns for 7 series devices. See Floorplanning for 7 Series Devices for more details.
- The resource usage of the largest RM needs to be taken into consideration when defining the Pblock in certain parts. If the largest RM exceeds the documented maximum resource counts of the target device, `write_bitstream` generates an error.
- The Pblock must not overlap any other Pblock in the design.
- Standard Pblocks for floorplanning logic within a RP are supported, as are nested Pblocks.
  - You can set a nested Pblock under a reconfigurable Pblock with `EXCLUDE_PLACEMENT=TRUE` to further constrain a specific module to a defined physical location.
- The IS_SOFT property of a reconfigurable Pblock is automatically set to `FALSE`, as the Pblock size and boundaries must remain fixed. Setting this property to `TRUE` results in an error.
- For any pre-Versal devices, the nested Pblocks under the reconfigurable Pblock inherit `IS_SOFT = FALSE` property. For Versal devices, the nested Pblocks under reconfigurable Pblock the default property of `IS_SOFT` is set to `TRUE`. For disjoint Pblock with guided floorplan user should manually set `IS_SOFT = FALSE` on the guided child Pblock (primary region) to avoid routing issues. For more information, see Create Disjoint Pblocks for a Reconfigurable Partition.

Table 5: **Pblock Commands and Properties**

Command/Property Name	Description
`create_pblock`	Command used to create the initial Pblock for each RP instance.
`add_cells_to_pblock`	Command used to specify the instances that belong to the Pblock. This is typically a level of hierarchy as defined by the bottom-up synthesis processing.

Table 5: **Pblock Commands and Properties** *(cont'd)*

Command/Property Name	Description
resize_pblock	Command used to define the site types (such as SLICE or RAMB36) and site locations that are owned by the Pblock.
RESET_AFTER_RECONFIG	Pblock property used to control the use of the dedicated GSR event on the reconfigurable region. Use of this property is highly recommended and, for 7 series and Zynq devices, requires clock region alignment in the vertical direction.
CONTAIN_ROUTING	Pblock property used to control the routing to prevent usage of routing resources not owned by the Pblock. This property is mandatory for PR and is set to True automatically for RPs. Static routing is still allowed to use resources inside of the Pblock.
EXCLUDE_PLACEMENT	Pblock property used to prevent the placement of any logic, not belonging to the Pblock, inside the defined Pblock RANGE. This property is mandatory for PR and set to true automatically for RPs.
IS_DFX	Pblock property used to indicate the Pblock is the primary Pblock for a reconfigurable partition. This property is set to TRUE automatically for RPs.
PARTPIN_SPREADING	Pblock property used to control the maximum number of PartPins per INT tile. The default is 5. Setting a lower value (3) increases the spreading between partition pin placements. This typically eases routing congestion in areas with dense PartPin placement, but can negatively affect RP interface timing.

The following is an example of a set of constraints for a RP:

```
#define a new pblock
create_pblock pblock_count
#add a hierarchical module to the pblock
add_cells_to_pblock [get_pblocks pblock_count] [get_cells [list inst_count]]
#define the size and components within the pblock
resize_pblock [get_pblocks pblock_count] -add {SLICE_X136Y50:SLICE_X145Y99}
resize_pblock [get_pblocks pblock_count] -add {RAMB18_X6Y20:RAMB18_X6Y39}
resize_pblock [get_pblocks pblock_count] -add {RAMB36_X6Y10:RAMB36_X6Y19}
```

Related Information

Apply Reset After Reconfiguration
Floorplanning for 7 Series Devices

Floorplan in the Vivado IDE

The Vivado IDE can be used for planning and visualization tasks. The best example of this is using the Device view to create and modify Pblock constraints for floorplanning.

1. Open the synthesized static design and the largest of each RM. Here are the commands, using the tutorial design (found in the *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947) as an example:

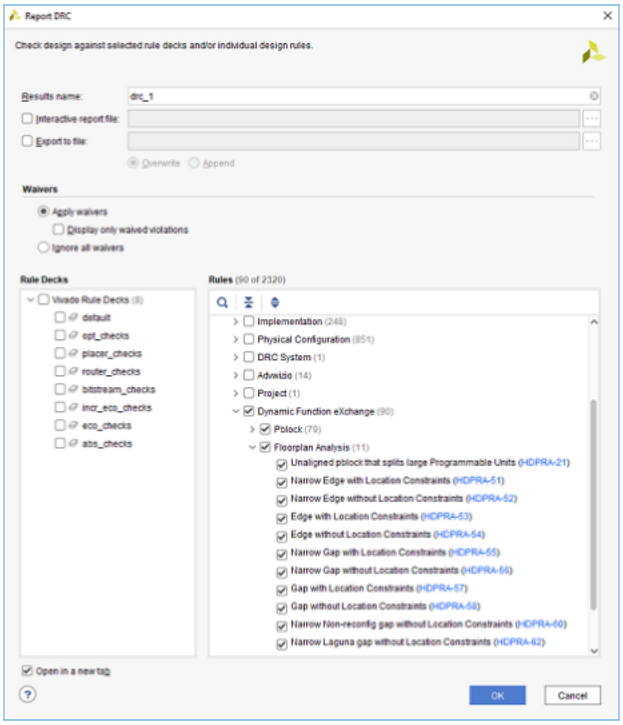
```
open_checkpoint synth/Static/top_synth.dcp
set_property HD.RECONFIGURABLE true [get_cells inst_count]
read_checkpoint -cell [get_cells inst_count] synth/count_up/
count_synth.dcp
set_property HD.RECONFIGURABLE true [get_cells inst_shift]
read_checkpoint -cell [get_cells inst_shift] synth/shift_right/
shift_synth.dcp
```

   At this point, a full configuration has been loaded into memory, and the RPs have been defined.

2. To create Pblock constraints for the RPs, right-click on an instance in the Netlist window (in this case, inst_count or inst_shift) and select **Draw Pblock**. Create a rectangle in the Device view to select resources for this RP.

3. With this Pblock selected, note that the Pblock Properties pane shows the number of available and required resources. The number required is based on the currently loaded RM, so keep in mind that other modules may have different requirements. If additional rectangles are required to build the appropriate shape (an "L", for example), right-click the Pblock in the Device view and select **Add Pblock Rectangle**.

4. Design rule checks (DRCs) can be issued to validate the floorplan and other design considerations for the in-memory configuration. To run, select **Reports → Report DRC** and ensure the DFX checks are present (see the following figure). If `HD.RECONFIGURABLE` has not been set on a Pblock, only a single DRC is available, instead of the full complement shown below.



This set of DRCs can be run from the Tcl Console or within a script, by using the `report_drc` command. To limit the checks to the ones shown here for DFX, use this syntax:

```
report_drc -checks [get_drc_checks HDPR*]
```

To extend the DRCs to those checked during specific phases of design processing the `-ruledeck` option can be used. For example, the following command can be issued on a placed and routed design:

```
report_drc -ruledeck bitstream_checks
```

To save these floorplanning constraints, enter the following command in the Tcl Console:

```
write_xdc top_fplan.xdc
```

The Pblock constraints stored in this constraints file can be used directly or can be copied to another top-level design constraints file. This XDC file contains all the constraints in the current design in memory not just the constraints recently added.

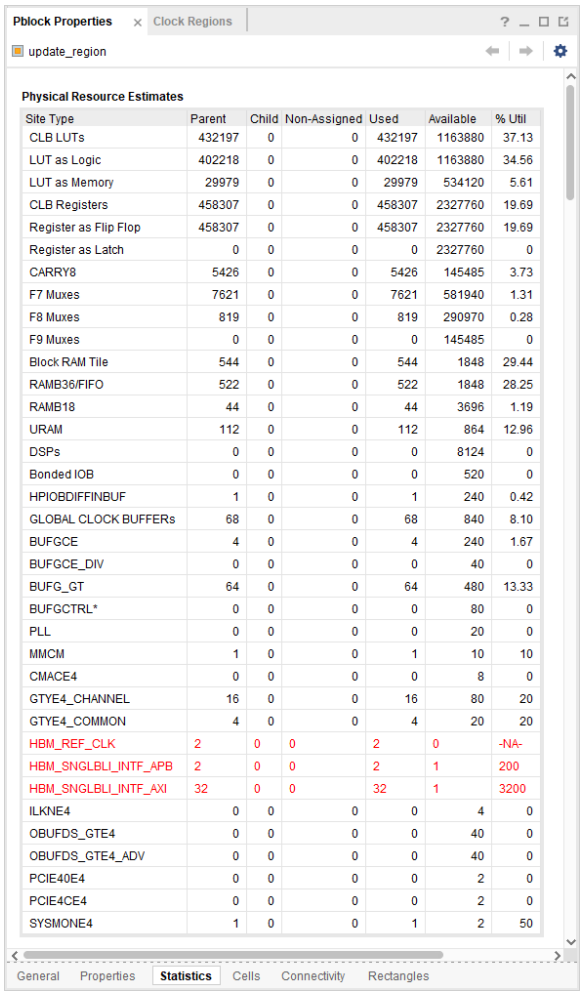
CAUTION! *Do NOT save the overall design from the Vivado IDE using **File → Checkpoint → Save** or the equivalent button. If you save the currently loaded design in this way, you will overwrite your synthesized static design checkpoint with a new version that includes RMs and additional constraints.*

Manage the Reconfigurable Partition Floorplan

With the floorplan for the DFX design created, Pblocks can be analyzed and modified as needed to suit the requirements of the design. The reconfigurable Pblock is a collection of `resize_pblock` commands, one for each resource type. These can be modified directly in the design constraint file (.xdc) or interactively within the IDE. With an existing Pblock, select and pull edges in or out to include more or fewer resources. If new resource types are enclosed, a dialog box asks which new types should be added to the Pblock. Modifications in the IDE can be saved to the target .xdc file in a project, or explicitly written to a constraint file using the `write_xdc` command.

With a floorplan open in the AMD Vivado™ IDE, select the Pblock to see the Pblock properties. Each resource type covered by this Pblock is shown within the Statistics tab. The number of each resource available and required for the current reconfigurable module is listed. Any deficiencies in number or type are shown in red, alerting you to expand the scope of the Pblock.

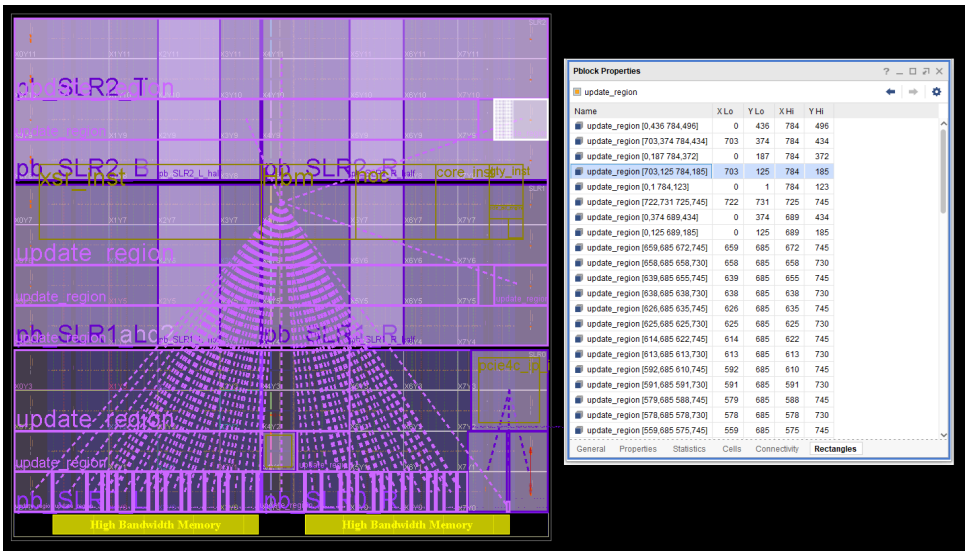
Figure 8: **Pblock Statistics**



Site Type	Parent	Child	Non-Assigned	Used	Available	% Util
CLB LUTs	432197	0	0	432197	1163880	37.13
LUT as Logic	402218	0	0	402218	1163880	34.56
LUT as Memory	29979	0	0	29979	534120	5.61
CLB Registers	458307	0	0	458307	2327760	19.69
Register as Flip Flop	458307	0	0	458307	2327760	19.69
Register as Latch	0	0	0	0	2327760	0
CARRY8	5426	0	0	5426	145485	3.73
F7 Muxes	7621	0	0	7621	581940	1.31
F8 Muxes	819	0	0	819	290970	0.28
F9 Muxes	0	0	0	0	145485	0
Block RAM Tile	544	0	0	544	1848	29.44
RAMB36/FIFO	522	0	0	522	1848	28.25
RAMB18	44	0	0	44	3696	1.19
URAM	112	0	0	112	864	12.96
DSPs	0	0	0	0	8124	0
Bonded IOB	0	0	0	0	520	0
HPIOBDIFFINBUF	1	0	0	1	240	0.42
GLOBAL CLOCK BUFFERs	68	0	0	68	840	8.10
BUFGCE	4	0	0	4	240	1.67
BUFGCE_DIV	0	0	0	0	40	0
BUFG_GT	64	0	0	64	480	13.33
BUFGCTRL*	0	0	0	0	80	0
PLL	0	0	0	0	20	0
MMCM	1	0	0	1	10	10
CMACE4	0	0	0	0	8	0
GTYE4_CHANNEL	16	0	0	16	80	20
GTYE4_COMMON	4	0	0	4	20	20
HBM_REF_CLK	2	0	0	2	0	-NA-
HBM_SNGLBLI_INTF_APB	2	0	0	2	1	200
HBM_SNGLBLI_INTF_AXI	32	0	0	32	1	3200
ILKNE4	0	0	0	0	4	0
OBUFDS_GTE4	0	0	0	0	40	0
OBUFDS_GTE4_ADV	0	0	0	0	40	0
PCIE40E4	0	0	0	0	2	0
PCIE4CE4	0	0	0	0	2	0
SYSMONE4	1	0	0	1	2	50

Pblocks can also be seen as a collection of rectangles, independent of resource type. The Rectangles tab shows a full list of rectangles that comprise the entire Pblock. You can select a rectangle to highlight the subsection of the Pblock it represents. To remove a rectangle, right-click the rectangle, and select **Clear Rectangle**. This is useful when pruning an oddly shaped Pblock, returning a specific section back to the static design.

Figure 9: **Pblock Rectangles**



Programmable Units

Programmable Units (PU) are the minimum required resource set for partial reconfiguration. The granularity varies based on the resource type and architecture family. In 7 series, this recommended minimum size is a clock-region high column of a single resource – aligning to a clock region allows dedicated initialization to be done. In AMD UltraScale™ and newer architectures, adjacent sites share a routing resource (or Interconnect Tile), a PU is defined in terms of pairs. Specific details are shown in the Design Considerations chapters for UltraScale and AMD Versal™ devices.

One technique that can be used in floor planning is to build out to the fundamental programmable unit from any site that is needed within a Pblock. You can identify a fundamental PU by working outwards from any site like so:

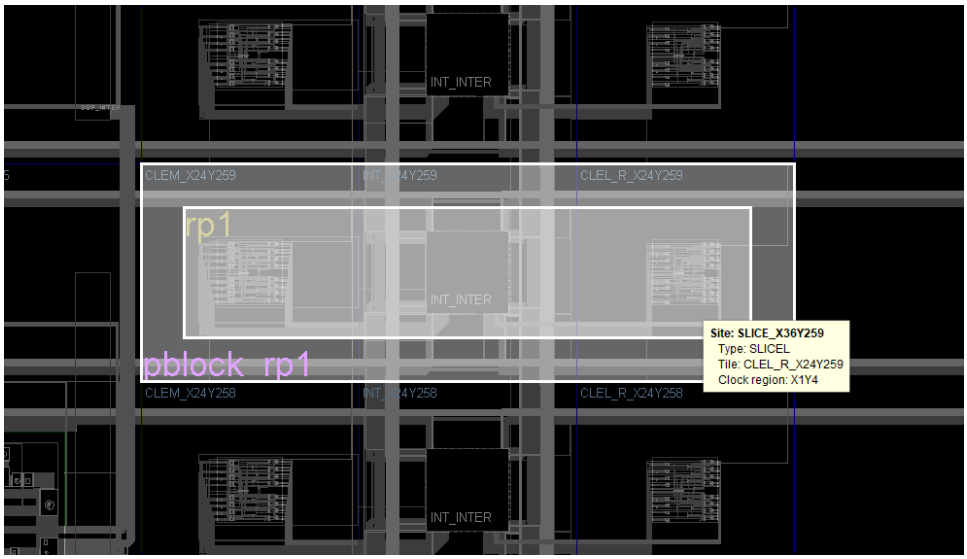
```
get_sites -of_objects [get_tiles -pu -of_objects [get_tiles -of_objects
[get_sites <site>]]]
```

For example, the programmable unit for side-by-side SLICE sites can be found in this manner:

```
resize_pblock pblock_rp1 -add [get_sites -of_objects [get_tiles -pu
-of_objects [get_tiles -of_objects [get_sites SLICE_X36Y259]]]]
```

This returns the base PU showing the pair of CLBs within a SLICE with the shared interconnect between them.

Figure 10: **CLB PU**

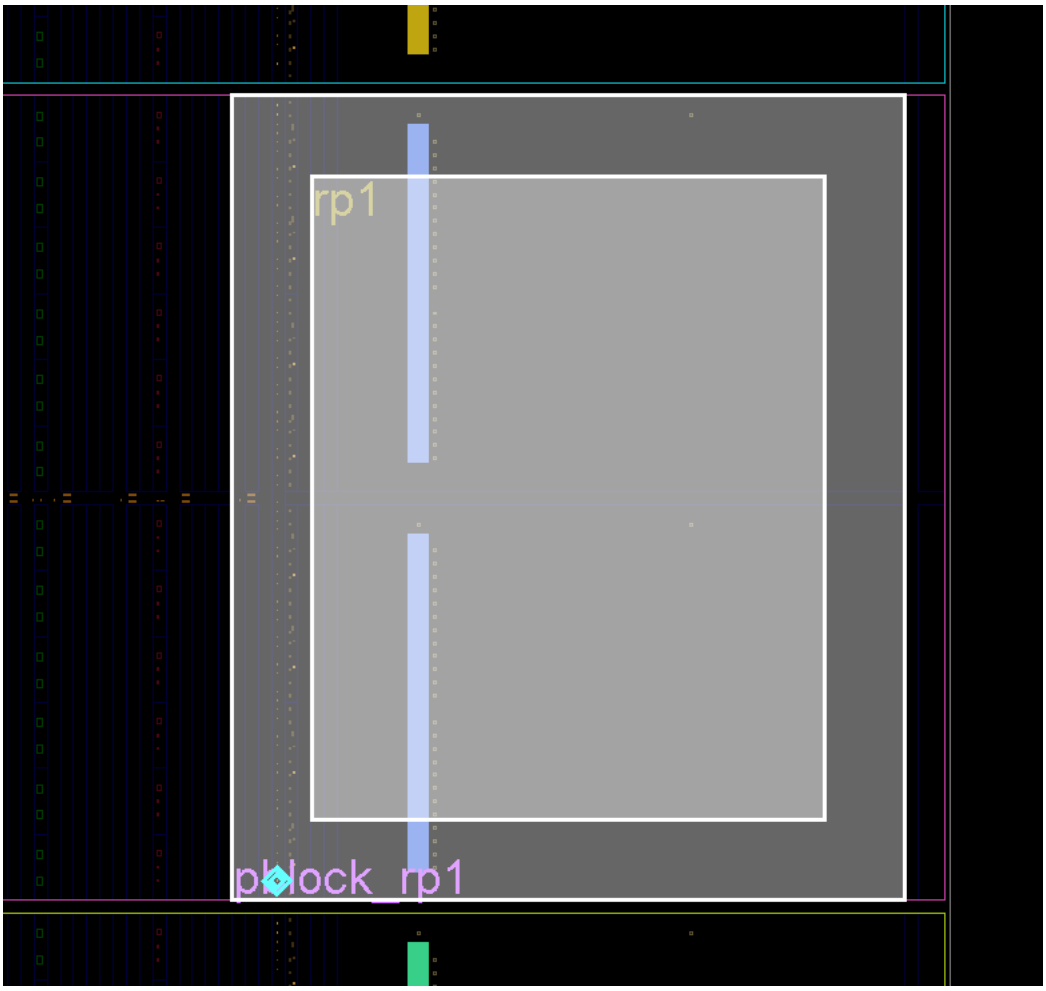


This technique is more typically useful when the site in question is less frequently used, such as clocking or IO resources, or Interlaken or PCIe®. If a clock modifying block such as an MMCM is desired, the resulting PU is a full IO bank.

```
resize_pblock pblock_rp1 -add [get_sites -of_objects [get_tiles -pu
-of_objects [get_tiles -of_objects [get_sites MMCM_X0Y1]]]]
```

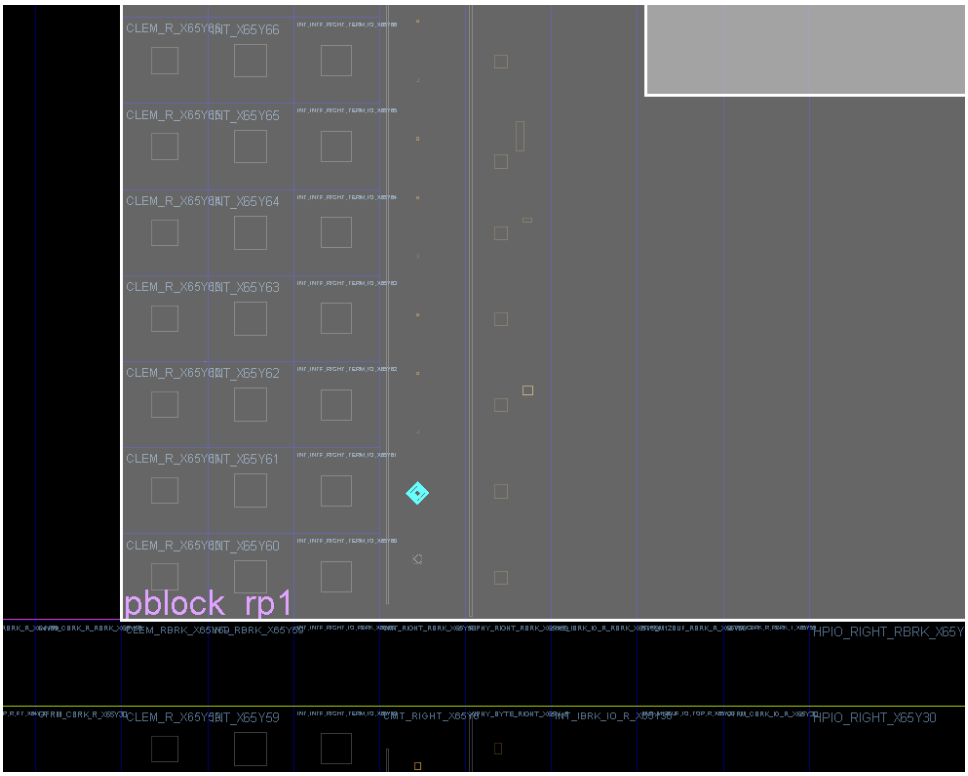
In the following figure, the MMCM is marked in blue.

Figure 11: **IO PU**



Looking more closely, you can see that a column of CLBs is part of this PU.

Figure 12: **IO PU Close**



Using Visualization Scripts

For each RP, scripts are automatically created to confirm the site ownership for each part of a DFX design. The visualization scripts generated can vary based on architecture and need.

Scripts are automatically created for all RP Pblock in an `hd_visual` directory, which is created in the directory where the run script is launched. To use these scripts, read a routed design checkpoint into the Vivado IDE, then source one of the scripts. These design-specific scripts highlight configuration tiles as you have defined them, show configuration frames used to create the partial bit file, or show sites excluded by the DFX floorplan. Additional scripts are created for other flows, such as Module Analysis or Tandem Configuration, and are not used for DFX.

For 7 series devices, the main script is named `<rp_pblock>_AllTiles.tcl` and shows all the sites owned by the RP, for both placement and routing of any implemented RMs. Three additional scripts might be created per design when necessary: `blockedBelsRouteThrus.tcl`, `blockedPins.tcl`, and `blockedSitesInputs.tcl`. When designs encounter higher levels of congestion, these scripts are created to show restricted sites. This information can be used to adjust the size and shape of the RP Pblock, and can also be shared with support for troubleshooting purposes. Other scripts are created for very specific goals and are not needed in most cases.

For UltraScale, AMD UltraScale+™ and Versal devices, hd_visual scripts are still generated but are not recommended for use. Instead, use the `get_dfx_footprint` Tcl utility to collect information about reconfigurable pblocks or the target device. See Floorplanning Visualization for more details.

Timing Constraints

Timing constraints for a partially reconfigurable design are similar to timing constraints for a traditional flat design. The primary clocks and I/Os must be constrained with the corresponding constraints. For more information on these constraints, see Defining Clocks and Constraining I/O Delays in the *Vivado Design Suite User Guide: Using Constraints* (UG903).

After the correct constraints are applied to the design, run static timing analysis to verify the performance of the design. This verification must be run for each RM in the overall static design. For more information on how to analyze the design, see the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906).

The Vivado Design Suite includes the capability to run cell level timing reports. Use the `-cell` option for `report_timing` or `report_timing_summary` to focus timing analysis on a specific RM. This is especially useful on configurations where the static design has been imported and locked from a prior configuration.

There is a **Partition** column added to the timing reports generated by `report_timing` and `report_timing_summary`. It helps identify if failing paths are within static, an RM, or crosses an RP boundary. Both of these commands have a new `-no_pr_attribute` switch to turn this new functionality off. This can be useful if, for example, scripts are being used to parse the timing reports and are negatively affected by this new column.

Partition Pins

Interface points called partition pins are automatically created within the Pblock ranges defined for the RP. These virtual I/O are established within interconnect tiles as the anchor points that remain consistent from one module to the next. No physical resources such as LUTs or flip-flops are required to establish these anchor points, and no additional delay is incurred at these points.

The placer chooses locations based on source and loads and timing requirements, but you can specify these locations as well. The following constraints can be applied to influence partition pin placement.

Table 6: **Context Properties**

Command/Property Name	Description
HD.PARTPIN_LOCS	Used to define a specific interconnect tile (INT) for the specified port to be routed. Overrides an HD.PARTPIN_RANGE value. Affects placement and routing of logic on both sides of the RP boundary. Do not use this property on clock ports, as this assumes local routing for the clock. Do not use this property on dedicated connections.
HD.PARTPIN_RANGE	Used to define a range of component sites (SLICE, DSP, block RAM) or interconnect tiles (INT) that can be used to route the specified port(s). The value is automatically calculated based on Pblock range if no user-defined HD.PARTPIN_RANGE value exists.

Context Property Examples

- `set_property HD.PARTPIN_LOCS INT_R_X4Y153 [get_pins <hier/pin>]`
- `set_property HD.PARTPIN_RANGE SLICE_X4Y153:SLICE_X5Y157 [get_pins <hier/pins>]`

`get_pins` should use the full hierarchical path to the pin or pins to be constrained on the partition interface. `get_ports` can also be used, but the reference must be scoped to the proper level of hierarchy. Instance names for interconnect tile sites can be seen in the Device View with the Routing Resources enabled.

***Note*:** The `HD.PARTPIN_RANGE` is automatically set during `place_design` if no user-defined value is found. Once the value is set, it will not be reset during interactive place and route, such as making experimental changes to the RP Pblocks and running `place_design -unplace`. In this case, the `HD.PARTPIN_RANGE` and `HD.PARTPIN_LOCS` need to be reset manually if Pblock adjustments are made. The properties can be reset like most properties.

The following Tcl proc can be useful when doing this kind of interactive floorplanning on DFX designs:

```
################################################
Proc to unroute, uplace, and reset HD.PARTPIN_*
################################################
proc pr_unplace {} {
route_design -unroute
place_design -unplace
set cells [get_cells -quiet -hier -filter HD.RECONFIGURABLE]
foreach cell $cells {
reset_property HD.PARTPIN_LOCS [get_pins $cell/*]
reset_property HD.PARTPIN_RANGE [get_pins $cell/*]
}
}
```

Partition pin information can be obtained from placed or routed designs by using the `get_pplocs` command. Use either the `-nets` or `-pins` option to focus the response to a particular RP or interface pin.

```
get_pplocs -nets <args> -pins <args> [-count] [-unlocked] [-locked] [-level
<arg>] [-quiet] [-verbose]
```

Table 7: **get_pplocs Options**

Name	Description
`-nets`	List of nets to report its PPLOCs.
`-pins`	List of pins to report its PPLOCs.
`[-count]`	Count number of PPLOCs; Do not report PPLOC or node names.
`[-unlocked]`	Report unlocked/unfixed PPLOCs only.
`[-locked]`	Report locked/fixed PPLOCs only; use `-level` to specify locked level.
`[-level]`	Specify locked level.
`[-quiet]`	Ignore command errors.
`[-verbose]`	Suspend message limits during command execution.

Example:

```
get_pplocs -pins [get_pins u_count/*]
```

In UltraScale or UltraScale+ designs, not all interface ports receive a partition pin. With the routing expansion feature, as explained in Expansion of CONTAIN_ROUTING Area, some interface nets are completely contained within the expanded region. When this happens, no partition pin is inserted; the entire net, including the source and all loads, is contained within the area captured by the partial bit file. Rather than pick an unnecessary intermediate point for the route, the entire net is rerouted, giving the Vivado tools the flexibility to pick an optimal solution.

***Note*:** The `PARTPIN_SPREADING` property in Table 5: Pblock Commands and Properties, can also be used to affect Partition Pins, but is applied at the Pblock level.

Related Information

Expansion of CONTAIN_ROUTING Area

Apply Reset After Reconfiguration

With the Reset After Reconfiguration feature, you hold the reconfiguring region in a steady state during partial reconfiguration. Next, you initialize all logic in the new RM to its starting values. Static routes can still freely pass unaffected through the region. Static logic and all other dynamic regions in the device continue to operate normally during partial reconfiguration. With this feature, DFX behaves like the FPGA initial configuration. You release synchronous elements in a known, initialized state.

IMPORTANT! *The release of global signals such as global set reset (GSR) global write enable (GWE) is not guaranteed to stay synchronized across the chip. If functionality in an RM relies on synchronized startup of initialized sequential elements, disable the clocks that drive that during reconfiguration. You can also disable* ***Clock Enables*** *on those elements during reconfiguration. After reconfiguration completes, re-enable the clocks or Clock Enables. For more information, refer to the Answer Record 44174.*

This is the `RESET_AFTER_RECONFIG` property syntax:

```
set_property RESET_AFTER_RECONFIG true [get_pblocks <reconfig_pblock_name>]
```

If the design uses the DRP interface of the 7 series XADC component, the interface is blocked (held in reset) during partial reconfiguration when RESET_AFTER_RECONFIG is enabled. The interface is non-responsive (busy), and there is no access during the length of the reconfiguration period. The interface becomes accessible again after partial reconfiguration is complete.

To apply the Reset After Reconfiguration methodology for 7 series and Zynq 7000 SoC devices, Pblock constraints must align to reconfigurable frames. Because the GSR affects every synchronous element within the region, exclusive use of reconfiguration frames is required; static logic is not permitted within these reconfigurable frames (static routing is permitted). Pblocks must align vertically to clock regions, because that matches the base region for a reconfigurable frame. The width of a Pblock does not matter when using `RESET_AFTER_RECONFIG`.

UltraScale and UltraScale+ devices do not have this clock region alignment requirement, and GSR can be applied at a fine granularity. Because of this, `RESET_AFTER_RECONFIG` is automatically applied for all RPs in the UltraScale and UltraScale+ architecture. This capability cannot be disabled.

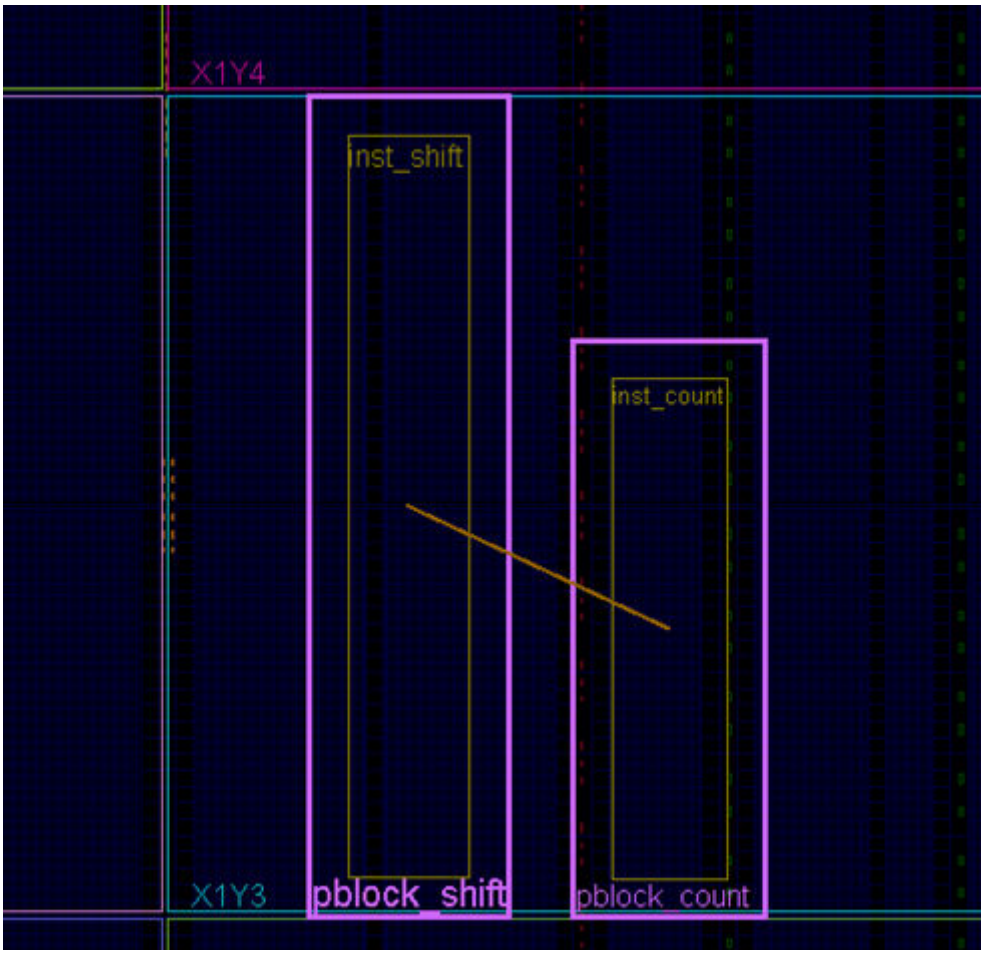
In the following figure, the Pblock on the left (`pblock_shift`) is frame-aligned because the top and bottom of the Pblock align to the height of clock region X1Y3. The Pblock on the right (`pblock_count`) is not frame-aligned.

- For 7 series devices: Pblocks that are not frame-aligned (such as `pblock_count` in the figure below) cannot have `RESET_AFTER_RECONFIG` set because any static logic placed between it and the clock region boundary above it would be affected by GSR after that module was partially reconfigured.

- For UltraScale and UltraScale+ devices: because of the improved GSR controls, both Pblocks automatically use `RESET_AFTER_RECONFIG`.

Using the `SNAPPING_MODE` constraint automatically creates legal, reconfigurable Pblocks. For more information, see Automatic Adjustments for Reconfigurable Partition Pblocks for 7 series devices or Automatic Adjustments for PU on Pblocks for UltraScale and UltraScale+ devices.

Figure 13: **RESET_AFTER_RECONFIG Compatible (Left) and Incompatible (Right) Pblocks**



The GSR capabilities are embedded within the partial bitstreams, so nothing extra must be done to include this feature during reconfiguration. However, because this process uses the `SHUTDOWN` sequence (masked to the reconfiguring region only), the external `DONE` pin are pulled `LOW` when reconfiguration starts, then pull `HIGH` when it successfully completes. This behavior must be considered when setting up the board. Using the `STARTUP` block `USRDONEO` is not an option to prevent the `DONE` pin from changing state, because this block is disabled during shutdown. Nor can `STARTUP` be used for other purposes, such as generating a configuration clock for partial reconfiguration if `RESET_AFTER_RECONFIG` is used.

To open the GSR mask for only the dynamic region when reconfiguration occurs, the mask for the entire design begins as closed after the initial configuration. Each partial bitstream opens the mask for the target region, loads new configuration data, issues a GSR event for this region, then closes the mask. For UltraScale devices only, this process is split between two bitstreams. See Clearning Bitstreams for more information. Because the mask is closed when reconfiguration is not occurring, full-device access to GSR is not permitted.

For 7 series devices only, an alternative approach would be to forgo this property and apply a local reset to any reconfigured logic that requires initialization to function properly. This approach does not require vertical alignment to clock region boundaries. Without GSR or a local reset, the initial starting value of a synchronous element within a reconfigured module cannot be guaranteed.

Related Information

Automatic Adjustments for Reconfigurable Partition Pblocks
Automatic Adjustments for PU on Pblocks
Clearing Bitstreams

Software Flow

This section describes the basic flow, and gives sample commands to execute this flow.

Synthesis

Each module, including the static module, must be synthesized bottom-up so that a netlist or checkpoint exists for the static module and each RM, including other HDL associated with the static design, such as black box module definitions for RMs.

1. Synthesize the top level:

```
read_verilog top.v
read_xdc top_synth.xdc
synth_design -top top -part xc7k70tfbg676-2
write_checkpoint top_synth.dcp
```

2. Synthesize an RM:

```
read_verilog rp1_a.v
synth_design -top rp1 -part xc7k70tfbg676-2 -mode out_of_context
write_checkpoint rp1_a_synth.dcp
```

3. Repeat for each remaining RM:

```
read_verilog rp1_b.v
synth_design -top rp1 -part xc7k70tfbg676-2 -mode out_of_context
write_checkpoint rp1_b_synth.dcp
```

Implementation

Create as many configurations as necessary to implement all RMs at least once. The first configuration loads in synthesis results for top and the first RM. You must then mark the module as being reconfigurable, then run implementation. Write out a checkpoint for the complete routed configuration, and optionally for the RM so it can be reused later if desired. Finally, remove the RM from the design (`update_design -cell -black_box`) and write out a checkpoint for the locked static design alone.

Configuration 1:

```
open_checkpoint top_synth.dcp
read_xdc top_impl.xdc
set_property HD.RECONFIGURABLE true [get_cells rp1]
read_checkpoint -cell rp1 rp1_a_synth.dcp
opt_design
place_design
route_design
write_checkpoint config1_routed.dcp
write_checkpoint -cell rp1 rp1_a_route_design.dcp
update_design -cell rp1 -black_box
lock_design -level routing
write_checkpoint static_routed.dcp
```

For the second configuration, load the placed and routed checkpoint for static (if it was closed), which currently has a black box for the RM. Then load in the synthesis results for the second RM and implement the design. Finally write out an implementation checkpoint for the second version of the RM.

Configuration 2:

```
open_checkpoint static_routed.dcp
read_checkpoint -cell rp1 rp1_b_synth.dcp
opt_design
place_design
route_design
write_checkpoint config2_routed.dcp
write_checkpoint -cell rp1 rp1_b_route_design.dcp
```



TIP: *Keep each configuration in a separate folder so that all intermediate checkpoints, log and report files, bit files, and other design outputs are kept unique.*

If multiple RPs exist, other configurations might be required. Additional configurations can also be created by importing previously implemented RM to create full designs that exist in hardware. This can be useful for creating full bitstreams with a desired combination for power-up, or for performing static timing analysis, power analysis, or simulation.

Full place and route results for each RM checkpoint is preserved completely, so creating new configurations is easily done by loading a collection of routed checkpoints. However, there are limitations to be aware of when using the flow. Using `write_checkpoint -cell` to save the RM implementation results does not preserve constraints local to this module. For RMs with internal clock constraints or timing exceptions starting and/or ending within the RM, these constraints need to be reapplied for timing analysis after creating the new configuration. RMs with AMD or third party IP are good examples of modules that might be exposed to this limitation.

Incremental Compile

Dynamic Function eXchange designs can use the Vivado Incremental Compile feature for any parent or child configuration run. The flow is documented in *Vivado Design Suite User Guide: Implementation* (UG904), and best results are seen when 95% of design instances match.

***Note*:** Support is limited to UltraScale and UltraScale+ devices only.

For any Incremental Compile usage, be sure to select a prior checkpoint for that specific configuration to match as much of the design as possible. Do *not* modify the design floorplan or boundary pins for any reconfigurable partition during an incremental run. It is recommended to recompile a parent configuration run when the layout or structure of a DFX design is required. When targeting a child configuration run, all static logic and routing is locked, so the Incremental Compile effort will be focused within RMs.

Reporting

Each step of the implementation flow performs design rule checks (DRCs) unique to partial reconfiguration. Keep a close eye on the messages given by the implementation steps to ensure no critical warnings are issued. These messages provide guidance to optimize module interfaces, floorplans, and other key aspects of DFX designs.

Most reports that can be generated do not have DFX-specific sections, but useful information can be extracted nonetheless. For example, utilization information can be obtained by using the `-pblocks` switch for the `report_utilization` command. This shows the used and available resources within a given RM. Here is an example using the design from the *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947):

```
report_utilization -pblocks [get_pblocks pblock_count]
```

For clock reporting, however, `report_clock_utilization` shows the clocks reserved for partial reconfiguration implementation.

The Dynamic Function eXchange flow can be used with the IEEE-1735 v2 encryption capability available within Vivado. Static design checkpoints can be encrypted and shared with other users without exposing details of the design. Rights management can be set such that details such as LUT contents and schematic details can be hidden, and netlist export and design modification can be disabled. Developers of dynamic regions can still insert their reconfigurable logic and implement within this locked static context. If permission is given, these developers can generate partial bitstreams from within this encrypted context for their dynamic function.

A license is required to use this feature, and any licensed IP within the static region still requires a valid license to open that checkpoint even if it is encrypted.

For more information on creating encrypted design checkpoints and the options available, see the Encrypting IP in Vivado chapter in *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118).

Verifying Configurations

Once all configurations have been completely placed and routed, a final verification check can be done to validate consistency between these configurations using `pr_verify`. This command takes in multiple routed checkpoints (DCPs) as arguments, and outputs a log of any differences found in the static implementation and Partition Pin placement between them. Placement and routing within any RMs is ignored during the comparison.

The `pr_verify` utility must be run on the same types of design checkpoints. If the full standard DFX design flow is used, `pr_verify` compares full design configuration checkpoints that have completed routing to confirm compatibility. If the Abstract Shell flow is used, use `pr_verify` to compare the initial abstract shell checkpoint that still contains a black box for the RP to a checkpoint that has an RM implemented in that partition, or to compare two RM implementations within the same abstract shell. Any `pr_verify` comparisons between full and abstract shells or different abstract shells are expected to fail, as the static design contents are different.

When only two configurations are to be compared, list the two routed checkpoints as `<file1>` and `<file2>`. The `pr_verify` command loads both in memory and makes the comparison.

When more than two configurations are to be compared, provide a master configuration using the `-initial` switch, then list the remaining configurations by using the `-additional` switch, listing configurations in braces (`{` and `}`). The initial configuration is kept in memory and the remaining configurations are compared against the initial one. Bitstreams should not be generated for any configurations if any pair of configurations do not pass the PR Verify check.

```
pr_verify [-full_check] [-file <arg>] [-initial <arg>] [-additional <arg>]
[-quiet] [-verbose] [<file1>] [<file2>]
```

Table 8: `pr_verify` **Options**

Command Option	Description
`-full_check`	Default behavior is to report the first difference only; if this option is selected, `pr_verify` reports all differences in placement or routing.
`-file`	Filename to output results to. Send output to console if `-file` is not used.
`-initial`	Select one routed design checkpoint against which all others will be compared.
`-additional`	Select one or more routed design checkpoints to compare against the initial one. List multiple checkpoints within braces, separated by a space, as in this example: `{config2.dcp config3.dcp config4.dcp}`
`-quiet`	Ignore command errors.
`-verbose`	Suspend message limits during command execution.

The following is a sample command line comparing two configurations:

```
pr_verify -full_check config1_routed.dcp config2_routed.dcp -file
pr_verify_c1_c2.log
```

The following is a second example verifying three configurations:

```
pr_verify -full_check -initial config1.dcp -additional {config2.dcp
config3.dcp} -file
three_config.log
```

The scripts provided with the *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947) have a Tcl Proc called `verify_configs` that automatically runs all existing configurations through `pr_verify`, and reports if the DCPs are compatible or not.

Bitstream Generation

As in a flat flow, bitstreams are created with the `write_bitstream` command. For each design configuration, simply issue `write_bitstream` to create a full standard configuration file plus one partial bit file for each RM within that configuration.

AMD recommends providing the configuration name and RM names in the `-file` option specified with `write_bitstream`. Only the base bit file name can be modified, so it is important to record which RMs were selected for each configuration.

Using the previous design, the following is an example of reading routed checkpoints (configurations) and creating bitstreams for all implemented RMs.

```
open_checkpoint config1_routed.dcp
write_bitstream config1
```

This command generates all possible bitstreams for this particular configuration. It creates a full design bitstream called `config1.bit`. This bitstream should be used to program the device from power-up and includes the functionality of any RMs contained within. It also creates partial bit files `config1_pblock_rp1_partial.bit` and `config1_pblock_rp2_partial.bit` that can be used to reconfigure these modules while the FPGA continues to operate. For UltraScale devices, it creates clearing bitstreams that pair with each partial bitstream, allowing you to prepare the partition for the next partial image. Repeat these steps for each configuration.

TIP: *Rename each partial bit file to match the RM instance from which it was built to uniquely identify these modules. The current solution names partial bit files only on the configuration base name and Pblock name: `<base_name>_<pblock_name>_partial.bit`*

The size of each partial bitstream is reported in the output from `write_bitstream`. As this command is run, these messages are reported for each partial and clearing bit file.

```
Creating bitmap...
Creating bitstream...
Partial bitstream contains 3441952 bits.
Writing bitstream ./Bitstreams/right_up_pblock_inst_shift_partial.bit...
```

Bitstream compression, encryption, and other advanced features can be used. See Known Limitations for specific unsupported use cases for UltraScale devices.

Related Information

Known Limitations

Generating Partial Bitstreams Only

If the full design configuration file is not required, then a single partial bitstream can be created on its own. With a full design configuration checkpoint loaded in memory, use the `-cell` option to identify the instance for which a partial bitstream is needed. The name of this partial bitstream can be given, as it is not automatically derived from the Pblock name.

```
write_bitstream -cell rp1 RM_count_down_partial.bit
```

This creates *only* a partial bitstream for the RP identified.

CAUTION! *Do not run `write_bitstream` directly on RM checkpoints; only use full design checkpoints. RM checkpoints, while they are placed and routed submodules, have no information regarding the top level design implementation, and therefore would create unsuitable partial bit files.*

Generating Full Configuration Bitstreams Only

If only power-on design bitstream is desired, the `-no_partial_bitfile` option can be used to bypass creation of partial bitstreams.

```
write_bitstream -no_partial_bitfile config3
```

Using this option skips the stage that creates partial and clearing bitstreams. It saves `write_bitstream` runtime for scenarios where either you are looking to test only the full design without DFX, or if the partial bitstreams already exist.

Generating Static-Only Bitstreams

If a power-on configuration of the static design only is desired, run `write_bitstream` on the checkpoint that has empty RPs (after `update_design -black_box` and `update_design -buffer_ports` have run). This greybox configuration can be compressed to reduce the bit file size and configuration time.

Tcl Scripts

Scripts are provided to run this flow in the *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947). The details of these sample scripts are documented in the tutorial itself and in the `readme.txt` contained in the sample design archive.

Nested Dynamic Function eXchange

Nested Dynamic Function eXchange (DFX) is the concept of placing one or more dynamic regions within a dynamic region, subdividing a device to permit more granular reconfiguration. With this feature, you can segment a RP into smaller regions, each of which is partially reconfigurable. This greater depth of flexibility allows for RM of different sizes, shapes, and resource sets to be swapped on the fly. For example, a data center application could load one large RM in a region in a device, or two smaller independent functions in that same region; these two smaller functions could then be individually reconfigured as needed, resulting more efficient use of silicon resources.

Although there is no formal limit to the number of levels into which a device can be subdivided, the further you subdivide, the more difficult it is to place and route. Also, the more complex the levels become, the more complex the management of partial bitstreams become. Realistically, most designs on even the largest devices should not exceed three levels of reconfiguration.

Nested DFX is supported as follows. Only the Tcl-based non-project flow is supported in this release.

- UltraScale and UltraScale+ devices, including Zynq UltraScale+ MPSoC and RFSoC devices are supported.
- Versal devices are not currently supported.
- 7 series are not supported.

Nested DFX Structure and Commands

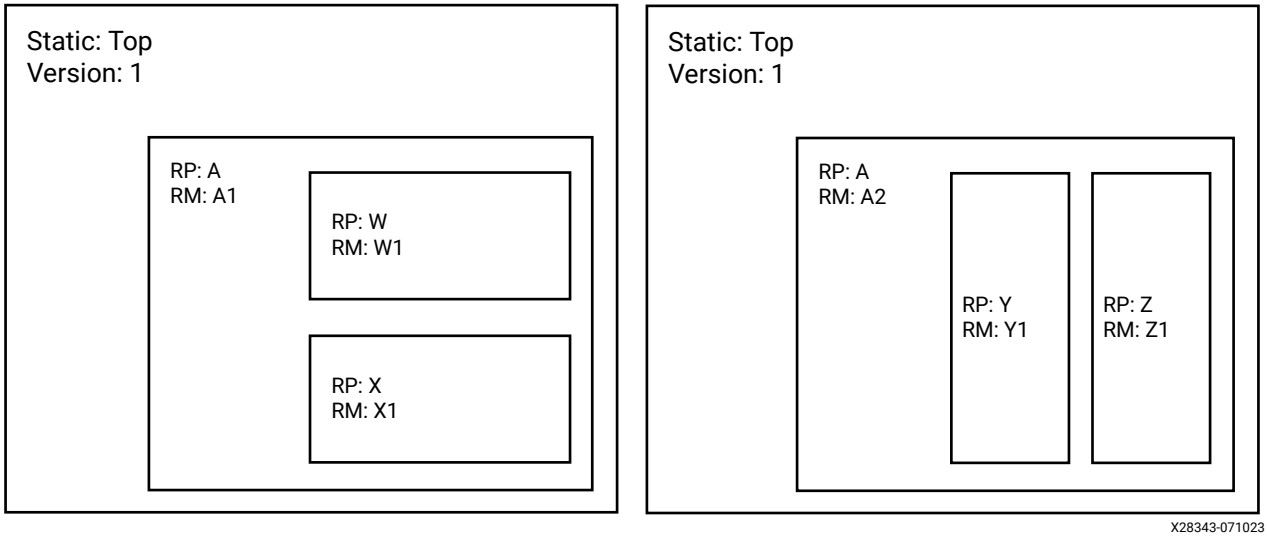
The Nested Dynamic Function eXchange flow is essentially a step and repeat through hierarchy, moving the context of static and reconfigurable boundaries with each pass through the Vivado tools. The first pass (configuration) establishes the implementation results of the static design, just as is done for the standard DFX flow. Subsequent runs establish and implement lower-order RPs, each with its own relative static level above it.

Throughout the flow, the general requirements and recommendations for a standard Dynamic Function eXchange design will still apply. For example, each module set to be reconfigurable must be synthesized out-of-context, each partition must be floor planned, and proper decoupling should be inserted on the static side of all reconfigurable boundaries.

Essentially Nested DFX should be viewed from the context of what is static and reconfigurable at a specific hierarchical boundary. See the following figure that describes the Vivado tool flow and design considerations.

Design Structure

Figure 14: **Two Configurations of a Nested DFX Design**



This image shows two configurations of a Nested DFX design. Both designs have the same static logic (Top) and floorplan for RP A. RP A is further divided into two more RPs, W and X for RM A1, or Y and Z for RM A2. RP A could have more RM versions (A3, A4, etc.) and these could have any number, size or shape sub-RPs, even none at all. The lower level RPs W, X, Y, and Z will each have their own collection of RMs (W1, W2, W3, etc.), and must each be implemented with all implementation results above them locked.

Note: In the image above, RPs W and X must be in different clock regions. RPs cannot occupy the same vertical column in any single clock region given the composition requirements of partial bitstreams.

In the Nested DFX implementation flow, users implement each RM in the context of the static design above it. The first RM for any RP establishes the static implementation results for the RM level immediately above it (A1 or A2 in this case), then all remaining RMs are implemented into this context. This is the flow regardless of where in the hierarchy the current target RP is.

Tcl Commands to Manage Nested DFX

Two new Tcl commands are used to subdivide and recombine the RPs for Vivado processing. Unsurprisingly, the command names are `pr_subdivide` and `pr_recombine`. These commands shift the perspective for Vivado tools, moving the HD.RECONFIGURABLE property lower (subdivide) or higher (recombine) to define the logical boundary of static versus reconfigurable.

pr_subdivide

The first new Tcl command is `pr_subdivide`. As the name implies, this command breaks up a RP into one or more lower level RPs.

```
pr_subdivide
Description: Subdivide an RP into one or more lower-level
RPs when using the Nested Dynamic Function eXchange solution.
Syntax:
pr_subdivide [-cell <arg>] [-subcells <arg>] [-quiet] [-verbose]
[<from_dcp>]
Usage:
Name      Description
-------------------------
[-cell]     (Required) Specify parent RP module name
[-subcells]  (Required) Specify child RP module names
[-quiet]    Ignore command errors
[-verbose]  Suspend message limits during command execution
[<from_dcp>] (Required) Specify OOC synthesized checkpoint path for the RM
specified by option -cell
```

`pr_subdivide` is used on the first implementation of a DFX design, the run that establishes the results of a static portion design. This is the case whether static is the very top level, or includes an RP that has just been subdivided. With a fully routed initial design checkpoint open in Vivado, running `pr_subdivide` automatically perform these tasks:

- Run `update_design -black_box` on the target RP, if it is not already a black box.
- Run `lock_design -level routing` on the remaining design if the target RP was not already a black box.
- Load a post-synthesis RM checkpoint for this RP, identified by the `<from_dcp>` argument. This RM must have one or more instances of hierarchy (filled with logic or black boxes) that becomes RPs themselves.
- Push the HD.RECONFIGURABLE property from the original partition (the `-cell` target) to one or more lower-level partitions (defined by the `-subcells` option).

- Place the HD.RECONFIGURABLE_CONTAINER property on the original partition as a placeholder. `pr_recombine` needs to see this property to push the context back up to this level.

If the target RP is already a black box, you must run `lock_design -level routing` BEFORE `pr_subdivide` has been run to lock down everything that is currently placed and routed. If `lock_design` is run after `pr_subdivide`, `DONT_TOUCH` properties added to the newly loaded RM netlist prevents logical optimization, inhibiting performance.

Because `pr_subdivide` must be run on a fully routed design, only one RP can be subdivided at a time. If a design has more than one RP - for example, in the design above it there was an RP B at the same level as RP A - the first subdivided RP must be implemented before the second RP can be subdivided. Any number of RPs can be subdivided, but they can only be created when all other RPs in the design have placed and routed RMs residing within them.

Similarly, you cannot subdivide an RP, then immediately subdivide a new child RP without first implementing the new static area for the lower-level RPs. In the figure shown in Design Structure, when RP A is subdivided into RPs W and X, module A1 must be implemented before W or X are themselves subdivided, as that process requires static results for A1 to be locked.

Related Information

Design Structure

pr_recombine

The second new Tcl command is `pr_recombine`. This command is used to remove all lower level RPs, restoring the RP definition to the parent cell. This command is used less frequently than `pr_subdivide`, as it is only needed for bitstream generation for a parent-level RM, or to return to a specific design structure for analysis of that specific configuration.

```
pr_recombine
Description: Re-establish a parent cell as a RP while removing
lower-level RPs when using the Nested Dynamic Function
eXchange solution.
Syntax:
pr_recombine [-cell <arg>] [-quiet] [-verbose]
Usage:
Name    Description
-----------------------
[-cell]   (Required) Specify reconfigurable container module name
[-quiet]  Ignore command errors
[-verbose] Suspend message limits during command execution
```

`pr_recombine` moves the HD.RECONFIGURABLE property to the target cell, removing it from any cells below it.

The target cell must currently have an HD.RECONFIGURABLE_CONTAINER property, deposited there by `pr_subdivide`, identifying it as a viable target for `pr_recombine`.

Implementation Design Flow

This section describes the implementation flow using the example from the Design Structure section. The first design pass contains logic for Top and module A, but information about submodules W and X is not needed at this point. The goal of the first run is to establish the implementation result for Top, down to the partition pin interfaces to RP A. The results for module A may even be discarded, but A should be a representative module to help achieve the highest quality results for Top. In the following figure, the design is completely routed and A is defined as a RP. This is a standard DFX configuration at this point.

Figure 15: **Implemented Baseline Design Prior to pr_subdivide**

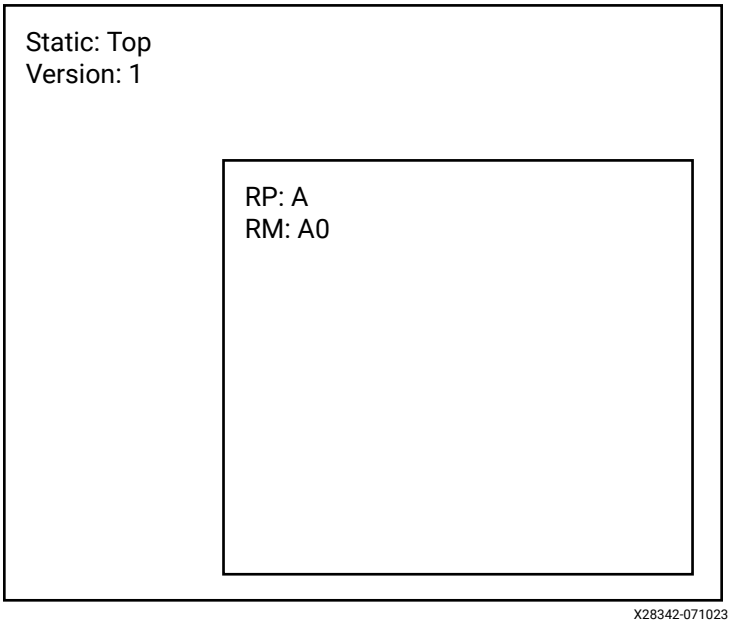
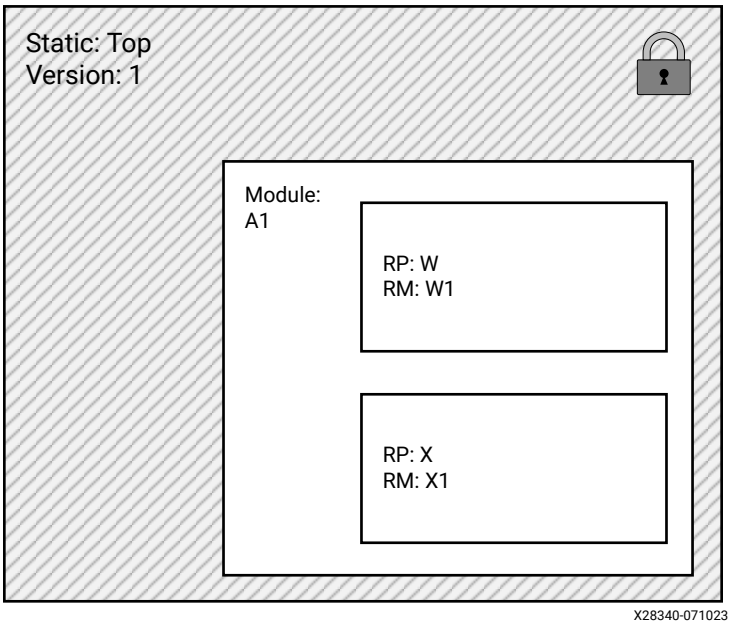


Figure 16: **Design After pr_subdivide**



Three key details are shown in the previous figure, which represents the design state after `pr_subdivide`, and after netlists and constraints have been added for W and X:

1. Top is implemented and locked.
2. A1 is not yet implemented, and no longer defined as an RP.
3. W and X are not yet implemented and are defined as RPs.

Related Information

Design Structure

Create Reconfigurable Module Results Under A1

Next, place and route this version of the design, implementing modules A1, W1 and X1. Then follow a normal DFX flow to create more RM results for different versions of W and X, implemented in the context of a locked A1 result, saving checkpoints along the way.

Starting with the design immediately after `pr_subdivide`, the first thing to do is to complete the full design (if post-synthesis design data for the new submodules were not included in the <from_dcp> checkpoint) and make sure the floorplan has pblocks for all new RPs. In this example code, `A1_pblocks.dcp` contains Pblock information for RPs W and X, and could contain timing or other constraints for any logic from A1 down.

```
read_checkpoint -cell A/W W1.dcp
read_checkpoint -cell A/X X1.dcp
read_xdc A1_pblocks.dcp
opt_design
place_design
route_design
write_checkpoint top_A1_W1_X1_routed.dcp
write_checkpoint -cell A/W W1_routed.dcp
write_checkpoint -cell A/X X1_routed.dcp
```

At this point, a normal DFX flow continues, with Top (already locked) and A1 (ready to be locked) representing the static design. Lock the static design and swap in new RMs for W and X and implement this second configuration using A1 in RP A.

```
update_design -black_box -cell A/W
update_design -black_box -cell A/X
lock_design -level routing
write_checkpoint top_a1_static.dcp
read_checkpoint -cell A/W W2.dcp
read_checkpoint -cell A/X X2.dcp
opt_design
place_design
route_design
write_checkpoint top_A1_W2_X2_routed.dcp
write_checkpoint -cell A/W W2_routed.dcp
write_checkpoint -cell A/X X2_routed.dcp
```

and so on. Using this standard step-and-repeat DFX flow, you should create a collection of routed checkpoints for W and X that are compatible with this version of Top and this version (A1) of partition A.

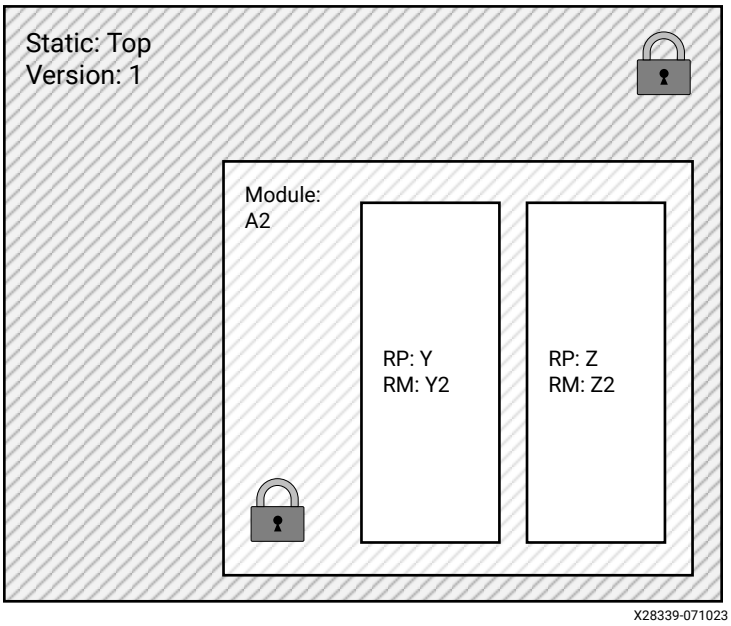
Create Reconfigurable Module Results Under A2

If other layouts or post-synthesis designs for module A are desired, the same fundamental process is followed. Starting with the initial routed design (either the full design with the initial implementation of A, or the locked static Top only and a black box for A) use `pr_subdivide` to insert a new module for A, and new RPs within.

```
open_checkpoint top_A0_routed.dcp
pr_subdivide -cell A -subcells {A/Y A/Z} A2.dcp
```

As seen in the Design Structure example image on the right, RM A2 is loaded with submodules Y and Z. From there, the implementation flow follows the same path as for A1, routing then locking module A2 while creating a set of RMs for Y and Z.

Figure 17: **Design with Both Top and A2 Locked, Implementing RMs Y2 and Z2**



Save checkpoints for full designs and RM with appropriate names. Use `update_design -black_box` to remove design information from RPs, leaving only the design above. These checkpoints will be used to assemble any combination of design modules for the purpose of running design analysis tools and generating full and partial bitstreams.

For example, for the design in the previous figure, use `write_checkpoint -cell A/Y` to save the routed result for module Y2, and `write_checkpoint -cell A/Z` to do the same for Z2. Then, after running `update_design -black_box` for by Y and Z, you can save just the results for A2. At that point, you can assemble a design image of (for example) Top+A2+Y1+Z2 which could be the default configuration the system will boot to.

Related Information

Design Structure

Checking Current Status

At any point, with a design checkpoint open, you can check the current status of any partition by reporting the properties on that cell. Calling `report_property` on a target cell will return one of three results as far as Nested DFX is concerned. An HD.RECONFIGURABLE* property will appear in the list if it is not set to its default value of false.

```
report_property [get_cells A]
```

1. `HD.RECONFIGURABLE bool false 1`

This cell is currently set as an RP. It can currently be implemented with one or more RMs and `pr_subdivide` can be called on it.

2. `HD.RECONFIGURABLE_CONTAINER bool false 1`

   This cell is currently set as a parent above one or more subdivided RPs. `pr_recombine` can be called on it.

3. <none>

   This cell is not and has never been a RP.

To get a listing of all cells in the design that are currently RPs or RP Containers, use a filtered `get_cells` command:

```
get_cells -hier -filter HD.RECONFIGURABLE
get_cells -hier -filter HD.RECONFIGURABLE_CONTAINER
```

Static Design Updates

Just as with the standard DFX design flow, implementation results are created in-context from the top down. If any part of the design that is considered static at any point must be updated, all results for RMs below that static must be reimplemented to ensure everything stays in sync.

For example, if there is a design change for Top, all existing results must be considered out-of-date and everything must be recompiled. Clearly AMD recommends creating modular design scripts to automate the process of rebuilding implementation results. If Top remains locked and there is an update only to module A1, all results dependent on A1 (all versions of W and X, as well as A1 itself) must be recompiled, but A2 and its lower level modules Y and Z, as well as any other versions of A, are still valid.

Verification Passes

Just as with a standard DFX design flow, Nested DFX design images should be checked using pr_verify to confirm all images are in sync. Like the core implementation tools (`opt_design`, etc.), `pr_verify` will act upon the design based on the current cells marked reconfigurable. With this in mind, perform apples-to-apples comparisons with the same current static design present.

In the examples shown here, run `pr_verify` to compare versions of A on a collection of checkpoints with only Top locked and cell A marked as reconfigurable - `pr_recombine` may be needed to create specific design images with this status. Run `pr_verify` on a collection of checkpoints with both Top and A1 locked to compare all images with different RMs for W and X, and then do the same with both Top and A2 locked to compare all images for Y and Z.

Bitstream Creation

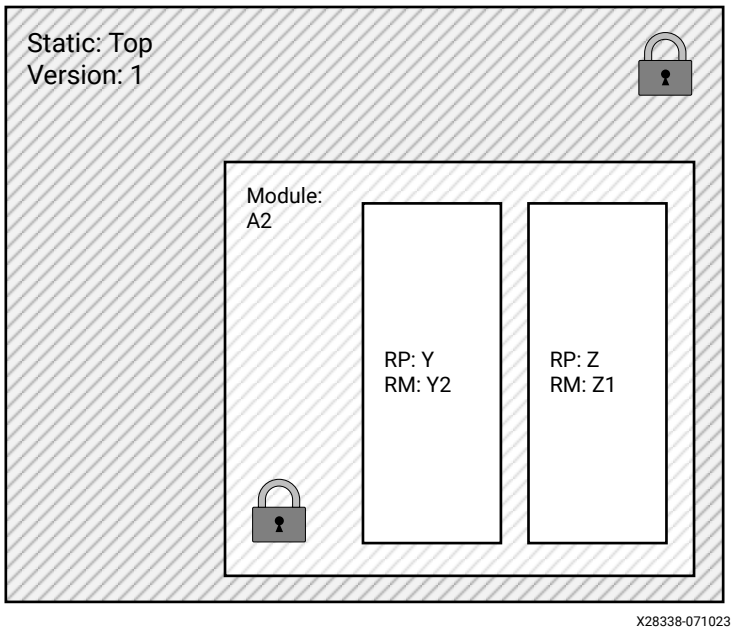
The Nested DFX design methodology moves the HD.RECONFIGURABLE property down and up through the hierarchy. Implementation tools follow standard DFX design rules based on what cells are currently defined as reconfigurable. This holds true for `write_bitstream` as well; partial bitstreams are only created for cells currently holding the HD.RECONFIGURABLE property.

With any fully routed design checkpoint open in Vivado, use `write_bitstream` to generate full and partial bitstreams. Remember, by default this command generates a standard full bitstream for the entire device and a partial bitstream for each cell defined as reconfigurable. Two options can limit results to one or the other:

1. The `-cell` option generates ONLY a partial bitstream for the requested cell.
2. The `-no_partial_bitfile` option generates ONLY a standard full device bitstream.

If the full design image you need a full device bitstream for, use a combination of `open_checkpoint` and `read_checkpoint -cell` (and `update_design -black_box` if necessary) to assemble a complete routed design, filling in each module one at a time with routed RM checkpoints. Use `report_route_status` to confirm that the design is complete.

Figure 18: **Assembled Design for Bitstream Generation**



For example, to create all bitstreams possible for the above design configuration, follow these steps. These commands assume that checkpoints for each module alone (each routed, and some locked) have been created using the same naming conventions as the A1 variants.

```
open_checkpoint top_A2_Y1_Z1_routed.dcp
update_design -black_box -cell A/Y
read_checkpoint -cell A/Y Y2_routed.dcp
write_bitstream top_A2_Y2_Z1.bit
```

This last command creates three bitstreams:

1. top_A2_Y2_Z1.bit, which is the full design bitstream for the entire device
2. top_A2_Y2_Z1_pblock_Y_partial.bit, which is the partial bitstream for Y2 only
3. top_A2_Y2_Z1_pblock_Z_partial.bit, which is the partial bitstream for Z1 only

Note that the names for the partial bitstreams are automatically generated. The name always starts with the base name you provided, followed by the RP Pblock name, followed by partial. If you would like to have names that show the name of the current RM (for example Y2), then call `write_bitstream` on the target RP directly:

```
write_bitstream -cell A/Y top_A2_Y2_partial.bit
write_bitstream -cell A/Z top_A2_Z1_partial.bit
```

Partial bitstreams can only be generated for cells currently marked as HD.RECONFIGURABLE. In order to create a partial bitstream for RP A, `pr_recombine` must be called before `write_bitstream`.

```
pr_recombine -cell A
write_bitstream -cell A A2_Y2_Z1_partial.bit
```

A full device bitstream for this image would be identical to one generated prior to `pr_recombine`, as the full device bitstream does not have any special programming indicating that later that device will be partially reconfigured.

Nested DFX Design Considerations

Nested Dynamic Function eXchange designs must follow the same rules and recommendations as standard Dynamic Function eXchange designs. Treat the parent RP (and above) as the static design. All design requirements for DFX are the same from the perspective of the reconfigurable boundary.

The Nested DFX solution does not allow more than one RP in a design to be subdivided until the first RP has a placed and routed implementation. Focus attention on a specific RP to create lower-level results. You can launch implementation runs in parallel after the design structure and floorplan is established.

Floorplanning

As you would expect, modules that are nested hierarchically must also have a nested floorplan. A child RP must have a Pblock that is completely contained within its parent RP Pblock, for all resource types. Expanded routing regions are supported and on by default, but they are created slightly differently in Nested DFX - routing expansion for lower-order RPs will fill the parent Pblock, but will still never overlap another RP at the same level. Use the `hd_visual` scripts to see the placement and routing regions for any RP in the design.

Partition pins are automatically managed by the Vivado tools and are established no differently for lower level RPs. Location ranges and constraints can be used if desired. Partition pins may be eliminated if the source and all loads are within the expanded routing region of the target RP.

Decoupling

Even though a submodule RP may be within a parent RP, from its perspective, everything above it is static. Strategies such as logical decoupling are still critical as one RM is loaded in to replace another. The DFX Decoupler, DFX AXI Shutdown Manager, or any user logic can be used to accomplish this task. Decouple only the RP that is about to be reconfigured.

Dynamic Function eXchange Controller IP

The DFX Controller does not (yet) know about Nested Dynamic Function eXchange, but it can still be used in this environment. Additional circuitry is needed to safeguard against loading partial bitstreams that would be incompatible with the currently operating design.

In the example design, a DFX Controller IP could be created to manage partial bitstreams for the entire design. It must be customized to understand five RPs: A, W, X, Y, and Z.

However, it does not know about the dependencies they have on each other, so the designer must build this part of the solution external to the IP.

For example, if RM A1 is currently loaded, the designer must not allow any trigger events that would reconfigure RPs Y or Z, only W or X. Each RP (Virtual Socket) has its own request, acknowledge and decouple controls, and these can be sent into a parent RP to access user logic that can manage child RPs - only acknowledge a request for reconfiguration if the target RP currently exists. Alternately, an AXI4-Lite interface can be used to read the current status of a parent RP managed by the DFX Controller to understand what child RPs can be reconfigured.

Bitstream Version and Usage Compatibility

The most critical aspect of Nested DFX is the additional importance of bitstream management. Not only must all full and partial bitstreams be generated from the same design version (using a consistent locked static image), but lower-level partial bitstreams must have consistent locked logic above them. `pr_verify` is used to confirm this consistency prior to bitstream generation, then users must manage bitstreams appropriately once they have been created. Tools such as the DFX Bitstream Monitor IP can be used to track bitstream versions during device operation.

Moreover, lower level partial bitstreams must be delivered only when their RP is active in the design. It is the responsibility of the system designer to build their controller solution such that it permits appropriate delivery of a lower-level partial bitstream. Nothing in the silicon will natively stop an incorrect partial bitstream from programming the device - as long as the Device ID is correct, the configuration engine will let it in.

Abstract Shell for Dynamic Function eXchange

AMD devices support Dynamic Function eXchange (DFX), which provides the capability to dynamically change the configuration of a portion of the device, while the rest of the device continues to operate normally. The Vivado tool flow lets you compile designs using an in-context methodology. The solution requires multiple passes through place and route.

The first pass establishes the static design implementation result along with the first RM for each RP. Then all subsequent place and route runs are done in context with that initial static image. A fully routed and locked static design database that contains netlist and placement and routing information for the entire static region must be loaded into Vivado before implementing any RM beyond the first.

The Abstract Shell solution reduces the requirements for this in-context flow. Because the static design is locked, it cannot (and must not) be modified when new RMs are implemented. The context is still critical, and the path through the tools does not change. However, instead of loading a full static design image, you can use an Abstract Shell checkpoint. This Abstract Shell contains only the minimal logical and physical database necessary to:

- Implement a new RM within a specific RP
- Validate timing and pass PR Verify
- Generate a partial bitstream for the RM

By using this approach, you can:

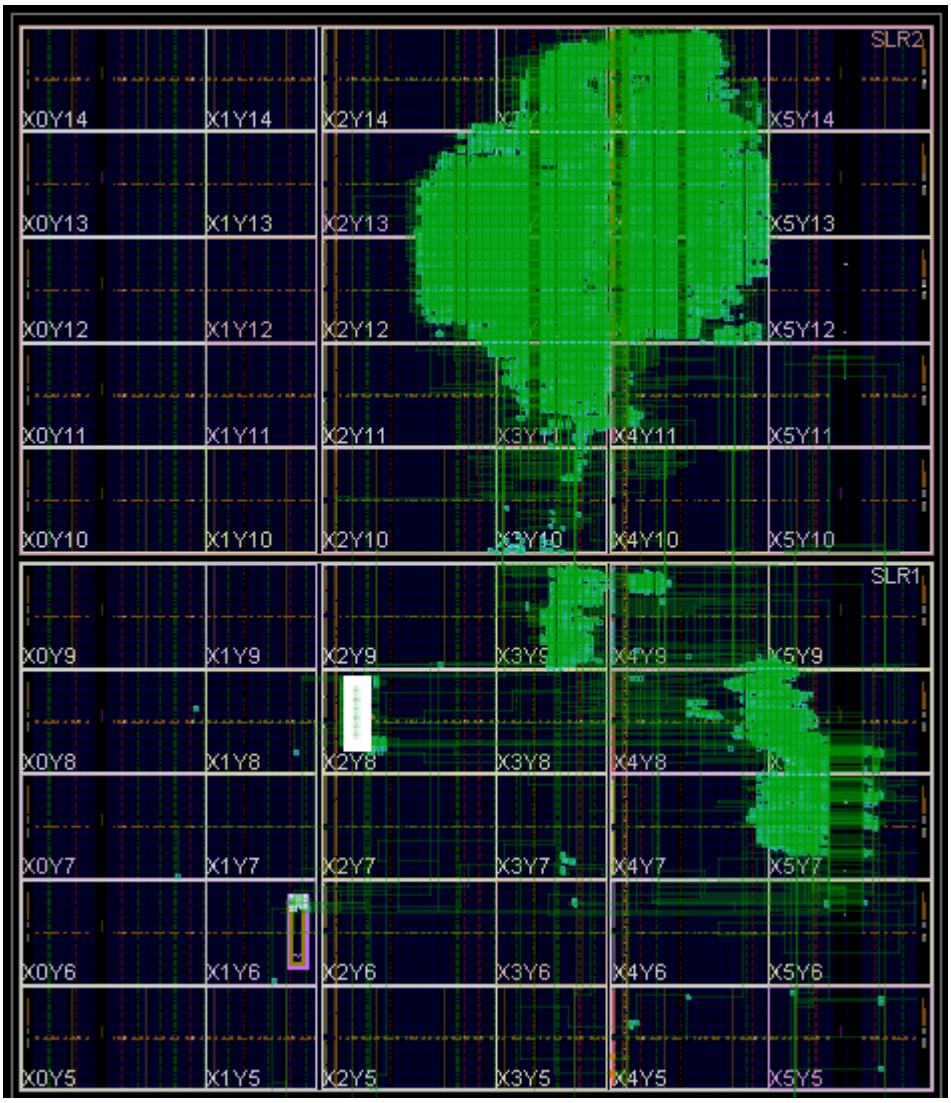
- Reduce compile time and memory usage for each RM compilation (beyond the first)
- Reduce the file size of the “static” design checkpoint for each RP

- For designs with multiple RPs, implement all RMs in parallel
- Generate partial bitstreams without the need to load the full static design
- Hide proprietary information that exists within the static design
- Avoid license checking for any IP in the static design

The Abstract Shell flow supports all UltraScale+ and Versal devices. Both project and non-project modes are supported, but the benefits of hiding proprietary design information or bypassing IP license checks are only possible in non-project mode. The Abstract Shell flow does not support UltraScale or 7 series devices.

As a point of comparison, here is the fully routed design checkpoint for the Lab 9 in *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947). This tutorial design targets a Virtex UltraScale+ VU9P and has two RPs for shift and count functions. u_shift is the Pblock in the lower left and u_count (selected) is above and to the right. This image is cropped to show only the top two SLRs in the device.

Figure 19: **Normal Full Static Design Targeting a VU9P**



The Abstract Shells for these two RPs strip away the vast majority of the static logic, which for this design includes the DFX Controller IP. Note that in this design with two RPs, only the target RP Pblock appears in the Abstract Shell checkpoint.

Figure 20: **Abstract Shell for the u_shift RP**

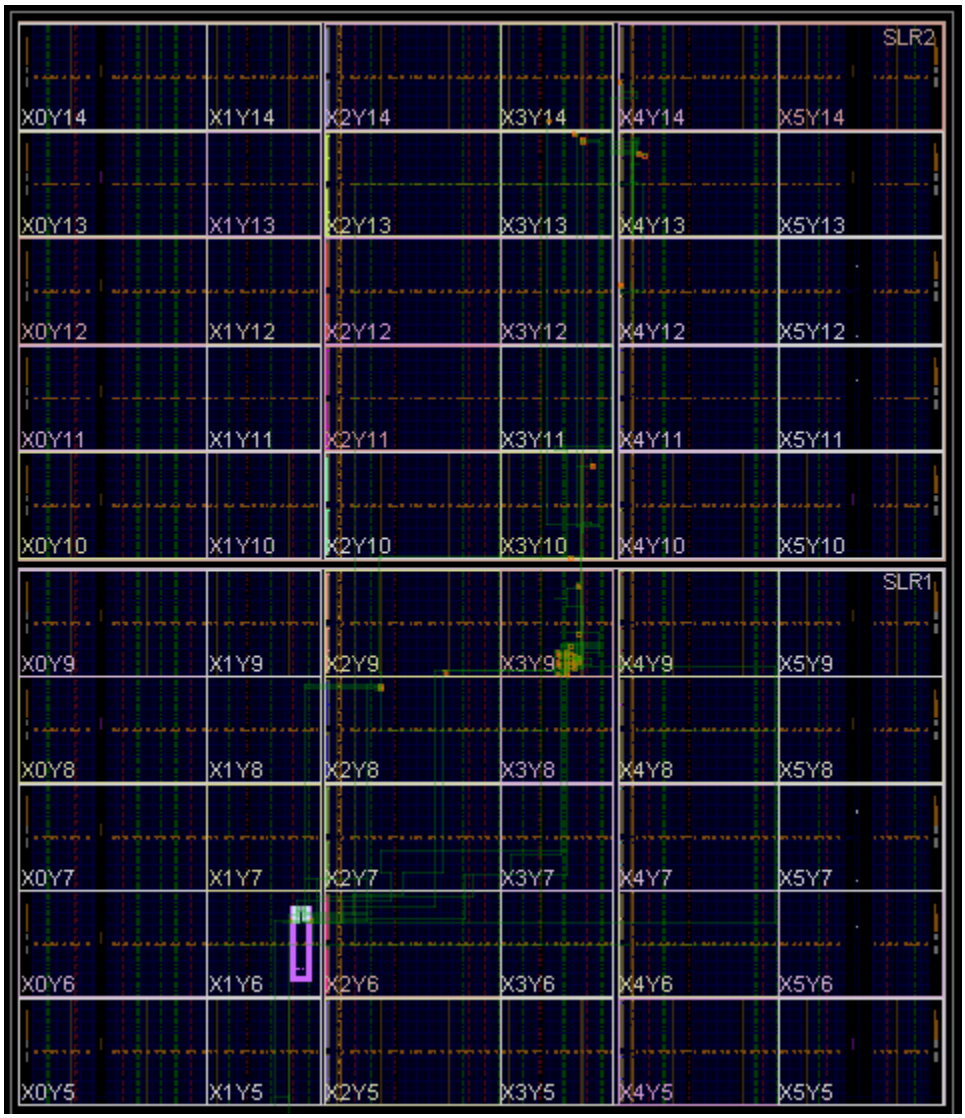
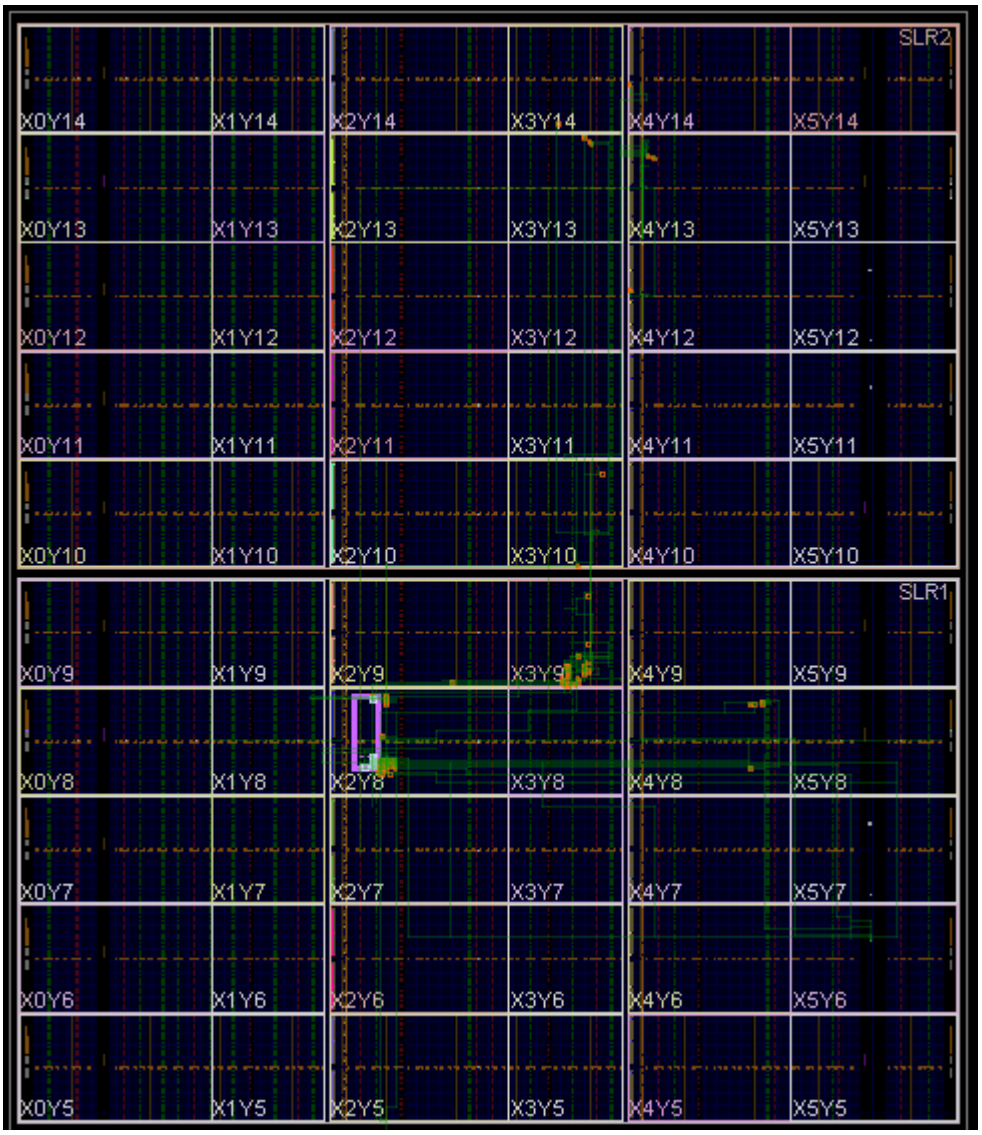


Figure 21: **Abstract Shell for the u_count RP**



For this design, the size of the full static-only design checkpoint is 58,816 KB. The checkpoint sizes for the Abstract Shells for u_shift and u_count are 1,712 KB and 1,873 KB, respectively. The size reduction is a function of both the size of the RP Pblock and the complexity of the static design. For a simple design with small RPs in a large device, the size difference can be very large. For designs with less static logic the improvement is more modest.

Abstract Shell Design Flow

The Abstract Shell design flow is nearly identical to the standard non-project Dynamic Function eXchange design flow. The differences are limited to the steps where the static design checkpoint is written from the initial (parent) implementation and where the static design is opened to begin implementation of the second RM (and beyond) for each RP.

Synthesis and implementation of the first configuration is no different with Abstract Shell than it is with the DFX standard flow. Once the initial full design checkpoint is implemented, an Abstract Shell for each RP can be created. If a design has more than one RP, each one has its own Abstract Shell. Any RMs for these RPs can be independently processed.

UltraScale+ and Versal devices are supported by the Abstract Shell flow, both in project and non-project modes. The commands in this section summarize the details about how the flow is run.

***Note*:** For Versal designs, in order to run a non-project flow, reconfigurable modules must only pass NoC connectivity through the interface using RTL-based modular NoC virtual interfaces. Block designs can be used within static (as is required for CIPS IP) and even within the RMs, but the concept of NoC INI ports is strictly limited to IPI project-based flows.

With the fully routed initial design image open in memory, with an RM present in each RP, use the `write_abstract_shell` command to create an Abstract Shell for a target RP. If more than one RP exists within a design, the command must be run separately for each RP. This command does the following:

- Carves out the target RP (using `update_design -black_box`)
- Locks the remaining design (including any other RMs, using `lock_design -level routing`)
- Writes the Abstract Shell for the target RP (using `write_checkpoint`)
- Runs `pr_verify` for this checkpoint compared to the original fully routed design

```
write_abstract_shell  -cell <arg> [-force] [-quiet] [-verbose] <file>
```

Table 9: **write_abstract_shell Switches**

Switch Name	Description
`-cell`	Specifies the full hierarchical name of the RP. Only one RP can be selected.
`-force`	Overwrites an existing checkpoint of the same name.
`-quiet`	Ignores command errors.
`-verbose`	Suspends message limits during command execution.
`<file>`	Specifies the full or relative path to the checkpoint (DCP) to be written.

When you look at the Abstract Shell checkpoints, you see they are smaller than the full design static checkpoint. For simple designs with large dynamic regions, they can be 80-90% of the size, depending on the floorplan and implementation. For larger designs, with more static logic, the size reduction is more considerable.

Abstract Shells contain only the logic and routing at the periphery of the target RP needed to implement new RMs into that RP. This includes any routing within not only the target RP Pblock but the expanded routing region as well. Much of the surrounding logic, including logic within the expanded region, is removed, and this configuration information is skipped in the resulting partial bitstream. Timing and boundary constraints are included in the shell as new modules implemented in this context inherit these goals from the static design.

Before continuing with Abstract Shells, you can confirm they have been constructed successfully by running two types of checks. First, consistency with the full static design it was created from is automatically confirmed when `write_abstract_shell` is called by running PR Verify. You can also perform this PR Verify check manually by comparing the Abstract Shell checkpoint to the static-only design after black boxes have been established and `lock_design` has been run.

The second check validates the routing status of the shell itself. This can be done by opening the Abstract Shell and calling `report_route_status` to check the status of the checkpoint.

***Note*:** Only the target RP can be a black box when an Abstract Shell is created. If any other RP is a black box, the `write_abstract_shell` command returns this error: `ERROR: [Common 17-69] Command failed: Failed to create design checkpoint`. In general, if tool errors or issues with place and route for new RMs in an Abstract Shell checkpoint are encountered, check the behavior using a full static design checkpoint first.

Related Information

Vivado Project Flow

Implementing Reconfigurable Modules in Abstract Shells

Any new RMs can be implemented within Abstract Shells. Each RM can be implemented in parallel in separate Vivado session as each RP is managed independently. The implementation flow is no different than the standard DFX flow starting with the full static design image.

IMPORTANT! *Use the same methodology in the Abstract Shell run as you used in the run that created the original implementation. For example, if the parent implementation uses the `add_files` / `link_design` approach (used in project mode), use the same approach for the Abstract Shell child implementation runs. If you used `open_checkpoint` and `read_checkpoint -cell` to build the initial design, continue that approach for the Abstract Shell implementation run.*

The flow through the implementation tools follows the same steps from `link_design` (or `read_checkpoint -cell`) through `place_design` and `route_design`, and like the standard flow, actions performed by the Vivado tools focus only on the target RM to be implemented. Any new constraints – such as placement directives, floorplanning, and timing goals – can be applied and scoped to the target RM.

When `route_design` is complete, call `write_checkpoint` to save the entire Abstract Shell with the implemented RM and call and `write_checkpoint -cell` to save just the implemented RM alone. The RM checkpoint alone can be read back into the full static design checkpoint (along with other RM checkpoints for other RPs if necessary) to assemble a complete design.

Generating Partial Bitstreams

Before considering partial bitstream generation, always use PR Verify. PR Verify compares multiple design images where RMs differ, but static is the same, to ensure all DFX rules have been followed. If full configuration assembly is done, you can run PR Verify in the standard way, comparing the entire static design for each checkpoint configuration. However, PR Verify can also run in the Abstract Shell context, comparing the initial Abstract Shell to the shell with the routed RM. If a checkpoint for an Abstract Shell with a routed RM is still open in Vivado, you can use the `-in_memory` option to compare it to the original shell. The comparison here is between the Abstract Shell for u_shift with a black box and the Abstract Shell with an RM implemented within it.

PR Verify fails if:

- A full static design checkpoint is compared to an Abstract Shell checkpoint
- An RM checkpoint is loaded without its Abstract Shell
- Abstract Shells for different RPs are compared

Generating Partial Bitstreams from Full Configurations

Bitstream generation can be done in two ways. The first is using the standard DFX approach, where a full design is open in Vivado and both full and partial bitstreams can be generated. Using the Abstract Shell approach, you do not create multiple configurations as the standard flow uses, as each RM is implemented on its own, independent of the full static top. However, any possible configuration can be created by linking the full static checkpoint with one RM checkpoint per RP. With a full configuration open in memory, you can call `write_bitstream` in the traditional manner. This by default produces all full and partial bitstreams for this design image. Use the `-no_partial_bitfile` or `-cell` options to create only full or only partial bit files, respectively.

Generating Partial Bitstreams from Abstract Shells

Alternatively, partial bitstreams can be generated directly from the Abstract Shell implementation for any RM. With this approach, the complete static design information is not required to generate partial bitstreams. The Abstract Shell contains all the information needed not only to implement any RM, but to create the bitstream for that function. The `-cell` option for `write_bitstream` is required.

```
write_bitstream -cell <cell_inst> <RM_partial>.bit
```

> **CAUTION!** *If the `-cell` option is omitted, `write_bitstream` flags an error, as this is interpreted as a request to create a full design bitstream. The full design is clearly not present here. Similarly, the Vivado tools return an error if you attempt to generate a partial bitstream from just the RM checkpoint alone (the one created from `write_checkpoint -cell`, with no static design data present above the target cell). This bitstream would only have the information for the RM, not for the static design that connects to it.*

Full device bitstreams can only be generated from checkpoints containing the full static design checkpoint plus one RM per RP. RMs can be greybox implementations, but even these must be fully placed and routed. Partial bitstreams from any full or Abstract Shell checkpoint are compatible with static as long as that version of static used to create each Abstract Shell has not been modified.

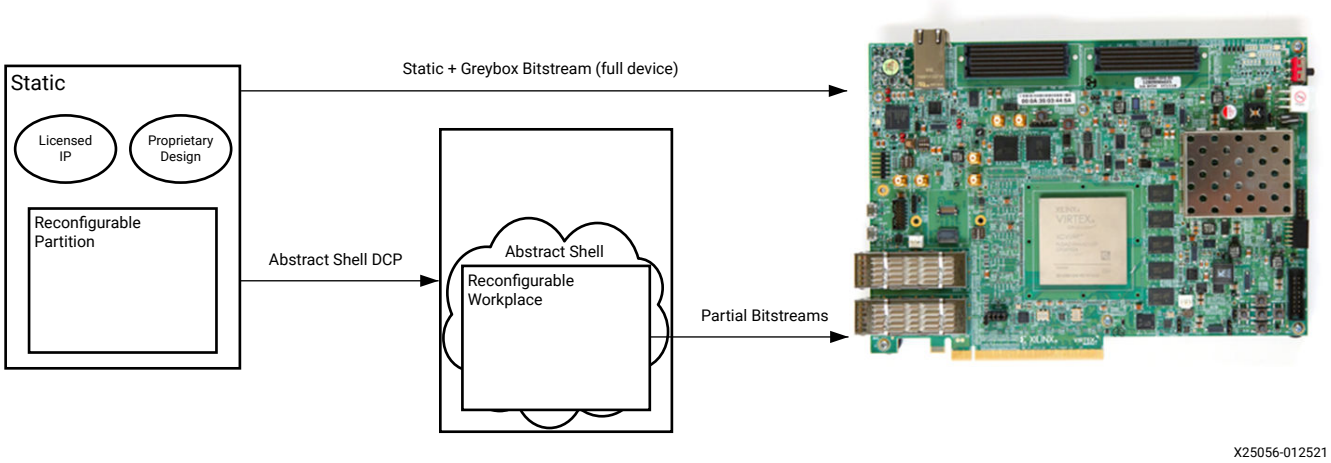
Device Programming Strategies

At the end of these implementation and bitstream generation processes, you have full and partial bitstreams as you do with the standard DFX flow. For single-user scenarios, follow the recommendations and considerations outlined in Design Considerations.

It is critical to keep all partial bitstreams in sync with the same implementation version as the full device bitstream. This must be part of the partial reconfiguration controller functionality, wherever and however this is managed. This is especially important when using Abstract Shell, as bitstreams can be created in different environments by different users. PR Verify confirms consistency, but once these bitstreams leave Vivado, it is up to the system design to maintain that consistency.

For multi-user environments, there are scenarios where the designer of the RMs does not have access to the static platform design. If you only have access to an Abstract Shell, you can only generate partial bitstreams. In such a scenario, you must also be given an initial device bitstream to program the static design. This full device bitstream can have a greybox (no functionality, only tie-offs) in the target RP, or any other initial “hello world” type of application. Configure the device with this initial image, then use Dynamic Function eXchange to load (and reload) their custom applications with the partially created bitstreams.

Figure 22: **Bitstream Delivery Flow for Multi-User Environments**



Any changes to the static platform design require updates to the full device bitstream and any Abstract Shell checkpoints to keep all bitstreams in sync.

Related Information

Design Considerations

Additional Solution Details

The contents of the Abstract Shell are determined by two main factors: the floorplan of the target RP and the connectivity to this module.

The floorplan for reconfigurable Pblocks in UltraScale+ devices by default creates an expanded routing region to improve routability and reduce congestion. This expansion creates the frameset to be captured as the programming contents for the partial bitstream. To implement any RM and generate the partial bitstream for this expanded region, the Abstract Shell contains some (but not all) of the placement and all of the routing information for this region. Any part of the static design included in a partial bitstream is reprogrammed while it continues to operate without disruption. Only the logic within the user-defined Pblock receives the GSR event for initialization at the end of partial bitstream delivery.

Use the hd_visual scripts to visualize the expanded region for a given RP. In the image below, the blue placement range aligns with the Pblock itself, while the yellow expanded region denotes the overall solution space and partial bitstream range for this RP.

Figure 23: **Routed Reconfigurable Module within an Abstract Shell**

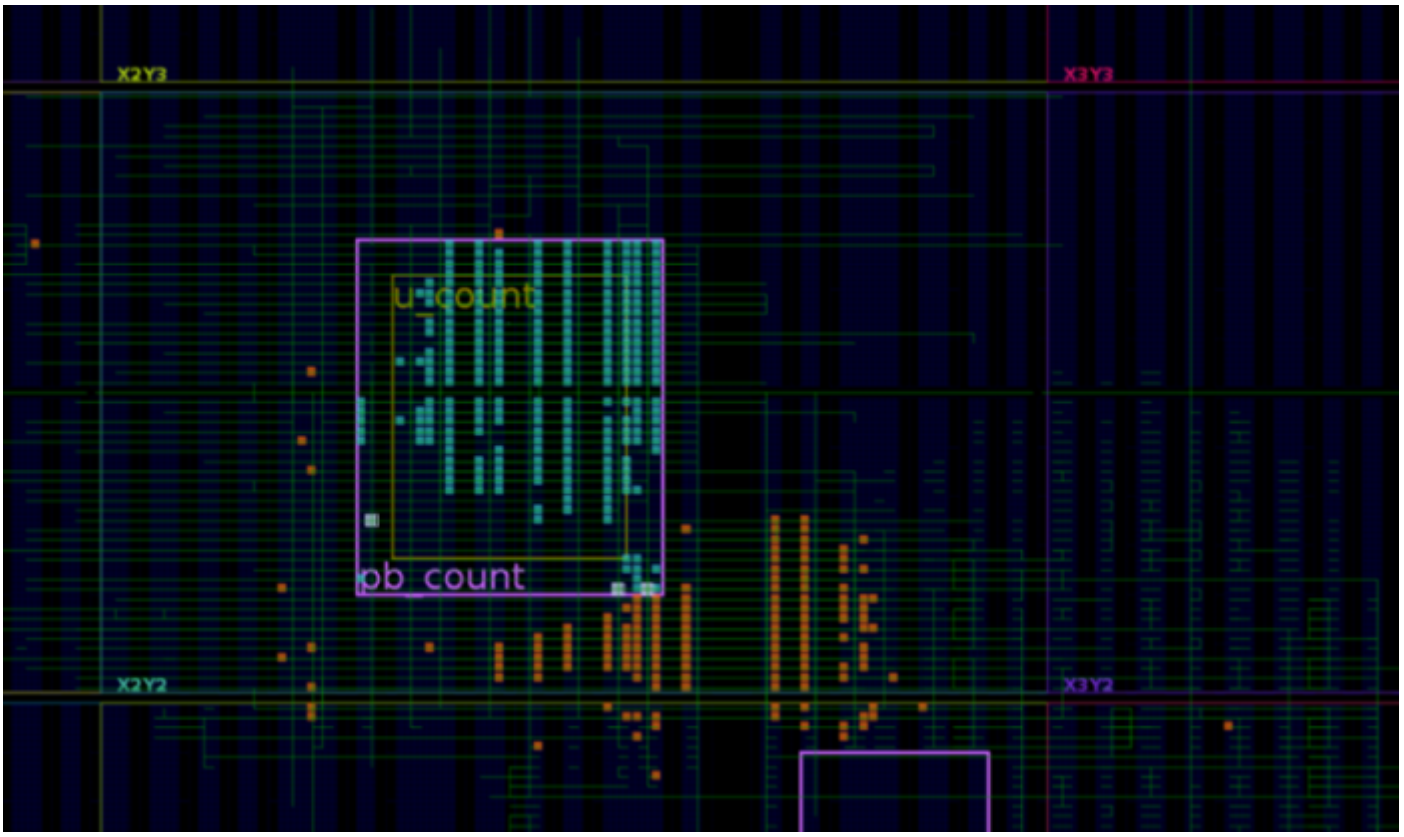
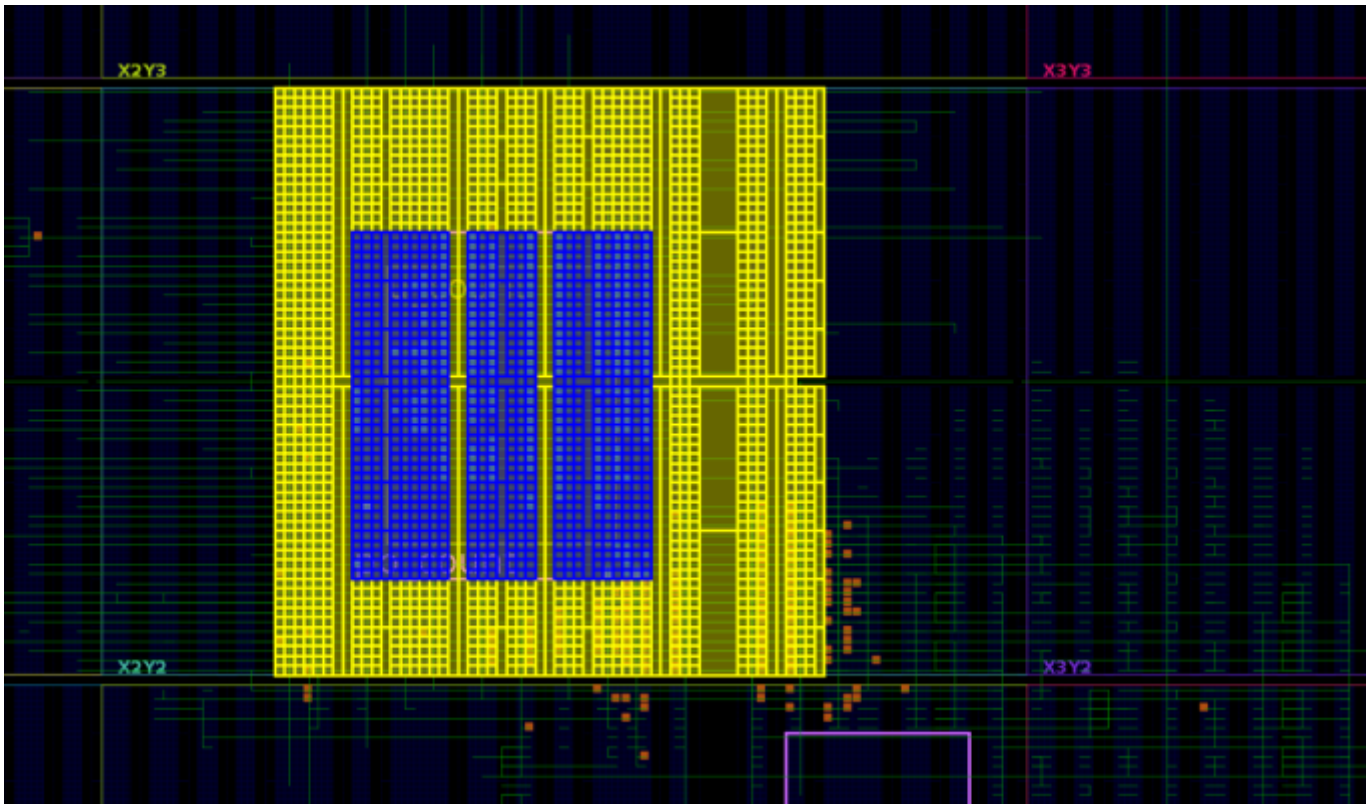


Figure 24: **Expanded Routing Region (yellow) for the Reconfigurable Partition (blue)**



For the RM implemented in the Abstract Shell, its placement is limited to the blue region, which is the Pblock as created by the designer and then snapped in to fundamental programmable unit boundaries. For this RM, the routing is limited to the yellow expanded routing region.

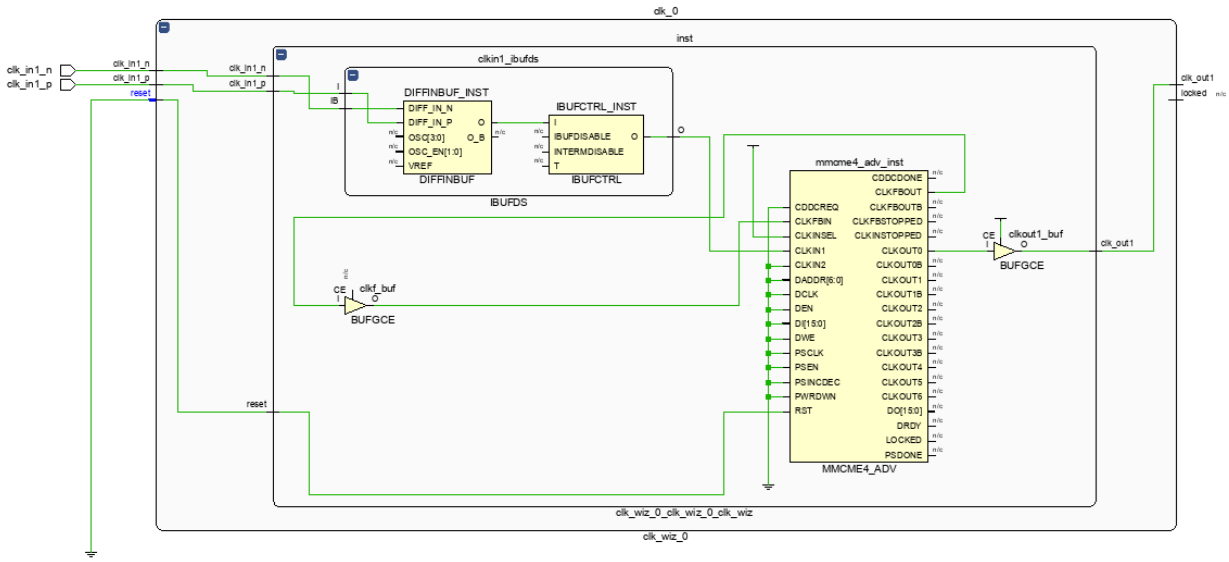
Logical Connectivity Contents

When the Abstract Shell is created, some static logic is included in the Abstract Shell, based on connectivity. All interface paths to and from the RP, up to their first synchronous element, are included so that timing closure can be done on these paths. The static side of these interface paths, up to the partition pin or first element, are locked, but the full path information must be included for timing analysis. This logic may be inside or outside the expanded routing region for the RP pblock. Other static logic in the expanded routing region is removed as long as it has no impact on placement, routing, or timing closure for the RP in that Abstract Shell. When this occurs, programming information in the partial bitstream is skipped, allowing that logic to continue operating during reconfiguration.

Information related to clocks and resets is also included in the Abstract Shell to create a complete picture of the context needed to implement each RM and confirm all timing constraints are met. This means preserving clock sources and clock modifying blocks such as clock buffers, MMCM, or PLL elements, as well as their connectivity, feedback paths, clocks driving boundary logic, and connections to external ports. It also includes anything else that has an impact on RM implementation. This information is used to supply a complete timing picture for the RP so that each RM to be implemented has the same operating conditions and constraints as when using a full static design shell.

For example, the full clock source path to an RP is captured in the Abstract Shell, including clock buffers and clock modifying blocks, as well as the clocking constraints needed to define the clock requirements.

Figure 25: **Full Clock Path within an Abstract Shell**




```
create_clock -period 10.000 [get_ports clk_in1_p]
```

Proprietary Design Information and Licensed IP

Given the static logic trimming previously noted, two benefits emerge.

First, most of the proprietary design information from the static platform is removed, effectively hiding this critical information from users that receive the Abstract Shell. Some fragments may remain:

- Synchronous elements that connect to the RP
- Any combinatorial logic between that synchronous element and the RP boundary routing elements that reside in the expanded routing region of the target RP
- Elements that tie into any common clocking or other global signals

Second, any IP that is contained in the static design is hidden (mostly, if not completely) from the Abstract Shell. Because of this, license checking for any IP (AMD or third-party) in static is bypassed. Designers who implement their RMs in an Abstract Shell do not need a license for any IP that exists in the static design. Any static IP, with or without an explicit license check, behaves just as the proprietary design information described in the previous paragraph.

IMPORTANT! *If you intend to redistribute an Abstract Shell checkpoint to a third party, be aware that you could be distributing a portion of the IP netlist from your static design. Ensure you have the legal rights to distribute this content. If you have any questions on obtaining IP netlist distribution rights, contact ip_admin@amd.com for any AMD IP, or the provider of the IP for any non-AMD IP.*

Abstract Shell and Nested DFX

In UltraScale+ designs only, the Abstract Shell and Nested DFX flows can be used together for the following use cases:

- Abstract Shells can be created for second-order RPs created by running `pr_subdivide`.
- `pr_subdivide` can be run on an Abstract Shell containing a routed RM to create new lower-order RP.
- Creating an Abstract Shell for a second-order RP.

Design Considerations

The flow to create the initial configuration of the design, establishing the static design results, is no different for Abstract Shell than the standard DFX flow. However, watch closely for an Abstract Shell when two RPs connect to each other. If there is no synchronous element on a path between two RPs, you can never be certain any possible RM combination in the two RPs is able to meet timing. While this is technically legal in DFX, AMD strongly recommends avoiding this scenario.

Two similar Design Rule Checks alert you to this scenario:

1. ```
   HDPR-34 - Signal '<object(s)>' is a direct path that connects RP
   '<object(s)>' and '<object(s)>' without a synchronous timing point in
   the static design. This omission might lead to timing failures in
   hardware depending on the RMs that are currently loaded. To close
   timing on all possible synchronous paths, ensure that any possible path
   contains at most a segment in only a single RP.
   ```
2. ```
   HDPR-35 - A path connects an RP '<object(s)>' and '<object(s)>' without
   a synchronous timing point in the static design. This omission might
   lead to timing failures in hardware depending on the RMs currently
   loaded. To close timing on all possible synchronous paths, ensure that
   any possible path contains at most a segment in only a single RP. The
   following is a list of nets (up to the first 15) in the path: <name>.
   ```

Run all DFX DRCs on the initial configuration of the design before creating Abstract Shells. If you encounter these particular alerts, please modify your static design accordingly.

False Paths Inferred During Abstract Shell Creation

During Abstract shell Creation, AMD Vivado™ can infer false path constraints on some of the static logic paths which are already placed and routed in the platform compile. There are situations where a specific static logic topology is retained in the abstract shell, however does not require to be timed again because these are not in the input/output cone of the RM compile. These false path constraints are expected to be created only on static logic which are outside the input/output cone of the reconfigurable module thereby users should not see any impact of these false paths during RM compile using abstract shell.

The initial bitstream used for downloading static region logic in hardware is created from the platform compile where the static region was implemented with user constraints. Hence these tool created false path constraints in the static region is never seen in the hardware. During RM compile, these false paths enables the RM compile to focus only on the RM logic along with its input and output cones.

Chapter 4

Vivado Project Flow

Dynamic Function eXchange (DFX) in AMD FPGAs and SoCs introduces new design requirements compared to traditional solutions. These requirements include unique approaches to source and runs management, as both bottom-up synthesis and multi-pass implementation are needed. These needs are met with the AMD Vivado™ Design Suite DFX Project Flow.

DFX flows can be run in project mode as illustrated in the following table for the two methodologies. Users must decide which path is best for their use case and needs, as the two flows cannot be mixed. One approach is an RTL-centric solution and the other is a block design-centric solution. Which flow is best for your needs? This chart compares differences between the two approaches:

Table 10: **Comparison of DFX Project Flows**

	RTL Project Flow	IP integrator Project Flow
Architecture Support	All architectures; not recommended for Versal devices	All architectures
Top Level design source	Verilog or VHDL	Block Design (with RTL wrapper)
Sources supported within RMs	IP, RTL, EDIF, and DCP	IP, BD, RTL, and EDIF
Designer Assistance, Connection Automation	No	Yes

The same DFX Wizard and related design runs are used for both modes, and each uses a consistent set of design rule checks and safeguards. Ultimately, if you are targeting AMD Versal™ devices and/or need to include block design within RMs, the IP integrator flow is the choice for you. Otherwise, either approach is viable.

RTL Project Flow in the Vivado IDE

Flow Summary

The Dynamic Function eXchange Project Flow inserts the key requirements of Dynamic Function eXchange into the existing Vivado project solution, accessible within the Vivado IDE as well as via Tcl commands. These key requirements include:

- Defining Reconfigurable Partitions (RP) within the design hierarchy

- Populating a set of Reconfigurable Modules (RM) for each RP
- Creating a set of top-level and module-level synthesis runs
- Creating a set of related implementation runs
- Managing dependencies as sources, constraints or options are modified
- Checking rules and results
- Verifying configurations
- Generating compatible sets of full and partial bitstreams

These fundamental aspects feature support for a front-to-back implementation for RTL-based designs including IP. Partial Reconfiguration (PR) terminology has been replaced with Dynamic Function eXchange (DFX) terminology. However, underlying Tcl commands have remain unchanged so that existing projects and scripts can safely migrate forward.

One expectation of this flow is that all sources (at least the top-level RTL and post-synthesis netlists for sub-modules) are managed within a single DFX-enabled project. A project cannot be broken up or exported, because the Vivado tools would no longer be able to track dependencies between runs and sources.

Tcl Commands

Like with most everything within the Vivado IDE, the features and tasks for Dynamic Function eXchange you see are driven behind the scenes by Tcl commands. One of the key goals for DFX project support is to be able to work seamlessly between GUI and script and command line on the same project. You can examine the specific Tcl commands called by examining the Vivado journal file for this project. This can be seen by selecting **File → Project → Open Journal File**. These Tcl commands are not currently documented in this user guide. The full set of commands used to create the entire project up to its current state can be generated by selecting **File → Project → Write Tcl**. Additional information for each command can be found using the `-help` option of each command.

Steps for Creating and Using a Dynamic Function eXchange Project

This section describes the general flow and the unique features and capabilities of the Vivado IDE for Dynamic Function eXchange design flows. For front-to-back tutorials using a specific design that target an AMD Evaluation Platform, see the *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947).

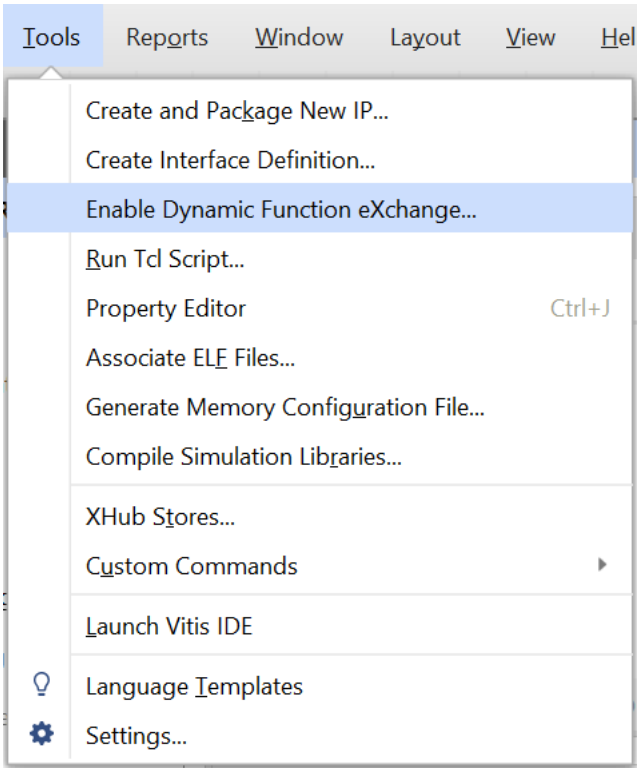
Creating a Dynamic Function eXchange Project

The initial creation of a DFX project is no different than for a standard design flow. Step through the New Project wizard to select the target device, design sources and constraints, and set all the main project details. When creating a new project, all source files and constraints for the static portion of the design should be added. You have the option of including the RTL and IP design sources for the first RM for each RP, or you can leave these as black boxes for now.

***Note*:** Only add sources for one RM during the initial project creation. The DFX wizard is used to add additional RMs to the project. This is discussed in more detail later in this chapter.

Once the project has been created, define it to be a Dynamic Function eXchange project. This is done by selecting **Tools → Enable Dynamic Function eXchange**. This prepares the project for the DFX design flow. Once this is set it cannot be undone, so AMD recommends archiving your project prior to selecting this option.

Figure 26: **Enabling Dynamic Function eXchange**



A subsequent dialog box will ask you to confirm this one-way project transition. When this is done, the project will show a few DFX-specific menu options and window tabs. These include:

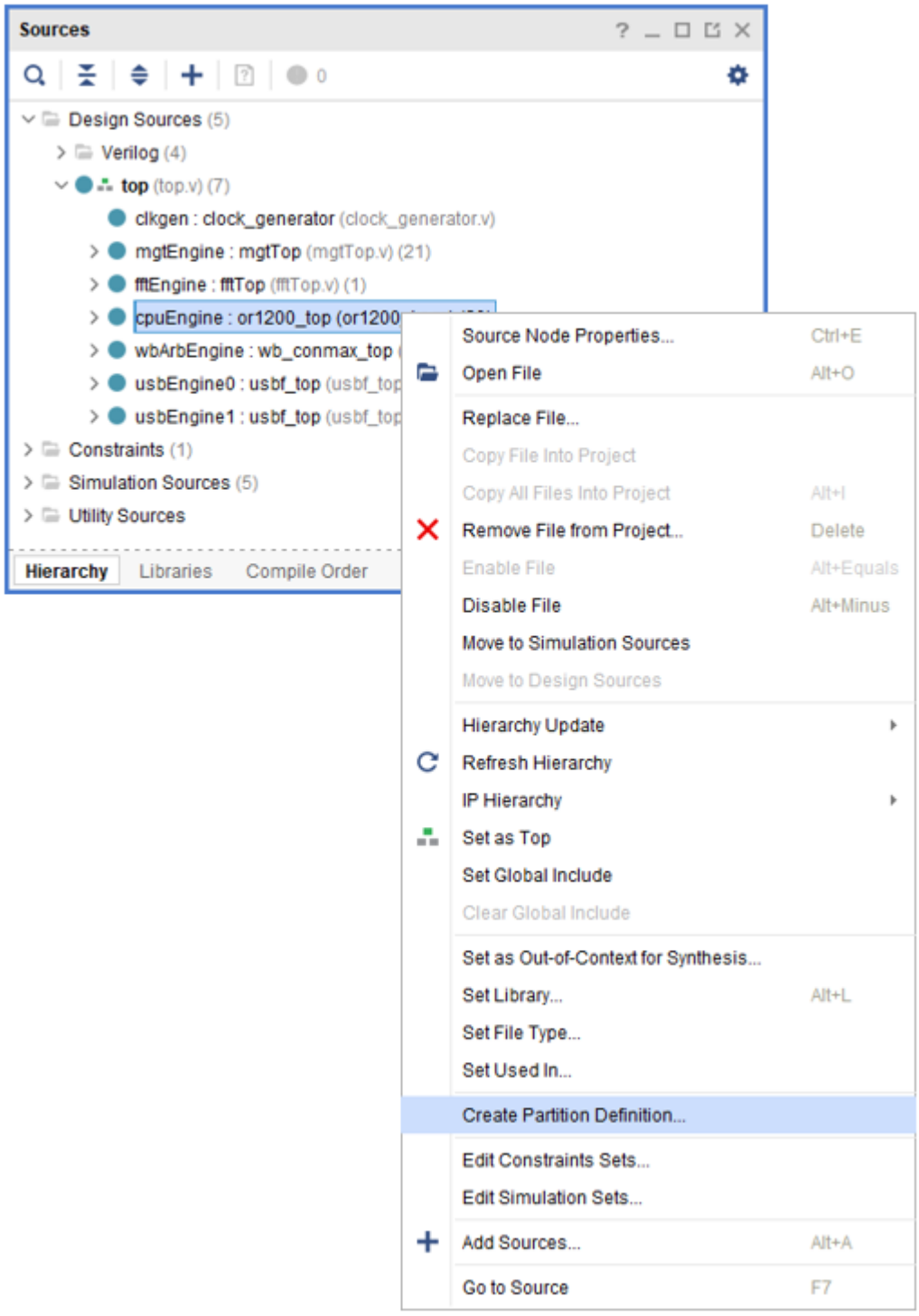
- A link to the Dynamic Function eXchange Wizard in the Flow Navigator
- The Partition Definitions view in the Sources window
- The Configurations window

Defining Reconfigurable Partitions

After the project has been turned into a DFX project, RPs can be defined within the RTL source hierarchy. Appropriate instances within the design hierarchy are those that:

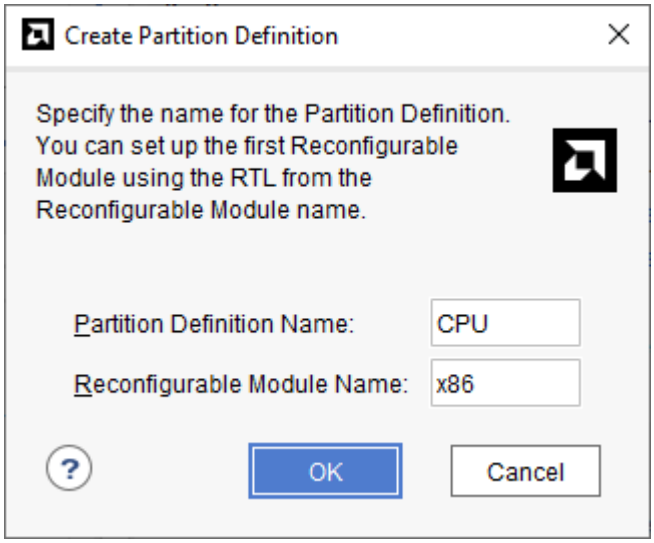
- Are defined by RTL, IP, EDIF, or DCP sources
- Do not pass parameter and generic values to that level of hierarchy from above. Parameters and generics can exist on the RP boundary but must be evaluated locally prior to partition creation.
- Do not contain out-of-context (other than IP) modules in the underlying RTL
- Do not have IP as the top level
- Do not contain block diagram (`.bd`) sources

1. Right-click on the desired module and select **Create Partition Definition** to begin the process of RP creation.

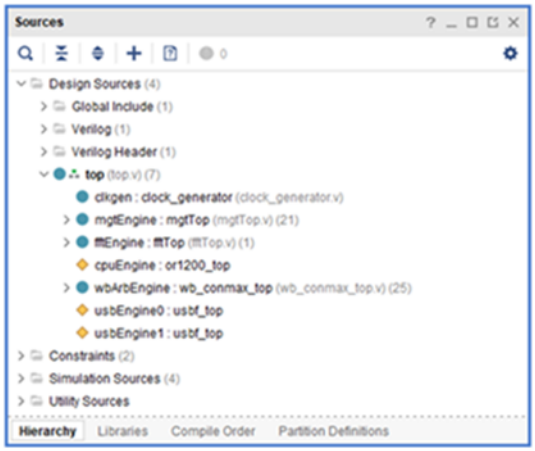


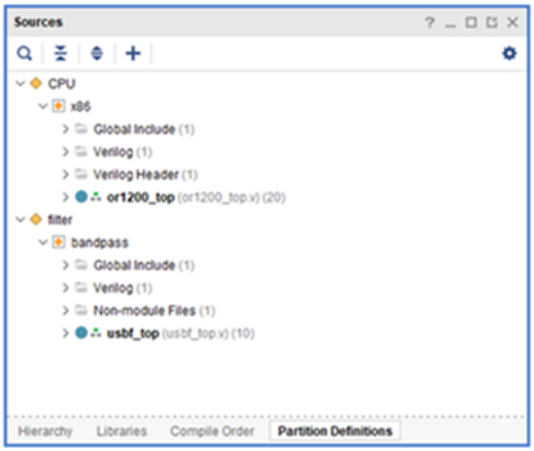
2. In the dialog box that appears, give this partition definition a unique name. Also define a name for the first RM. This RM is created from the RTL sources currently residing in this level of hierarchy. More RM are added or created later in the flow.

IMPORTANT! *Every instance of the selected module is turned into an RP. In order for one instance to be defined as reconfigurable and another instance to remain static, the two instances must be given unique module names.*



3. After clicking **OK**, this module displays differently in the Vivado IDE. Each instance of the module is shown in the Hierarchy view with a diamond, indicating that it is an RP. The design sources are moved to the Partition Definitions view to be managed separately. Repeat this step for all unique RPs required within the design.





Defining Netlist Sources

Netlist sources (EDIF) cannot be included in the current design hierarchy when the Partition Definition is initially created via the Vivado IDE. However, you can use the Tcl Console to call the underlying commands directly if the module is a single EDIF or DCP file. For example:

```
create_partition_def -name <RP_name> -module <RM_name>

create_reconfig_module -name <RM_name> -top <RM_top> -partition_def
[get_partition_defs <RP_name>] -gate_level
```

The `-gate_level` switch indicates that the RM is comprised of a single post-synthesis netlist (either EDIF or DCP). If the component declaration is provided in the top-level design to define the instantiation port list (size and direction), a stub file is not needed.

If the netlist source is a submodule under the RTL, wait until after the partition definition has been created to add it to the project. Vivado flags an error otherwise:

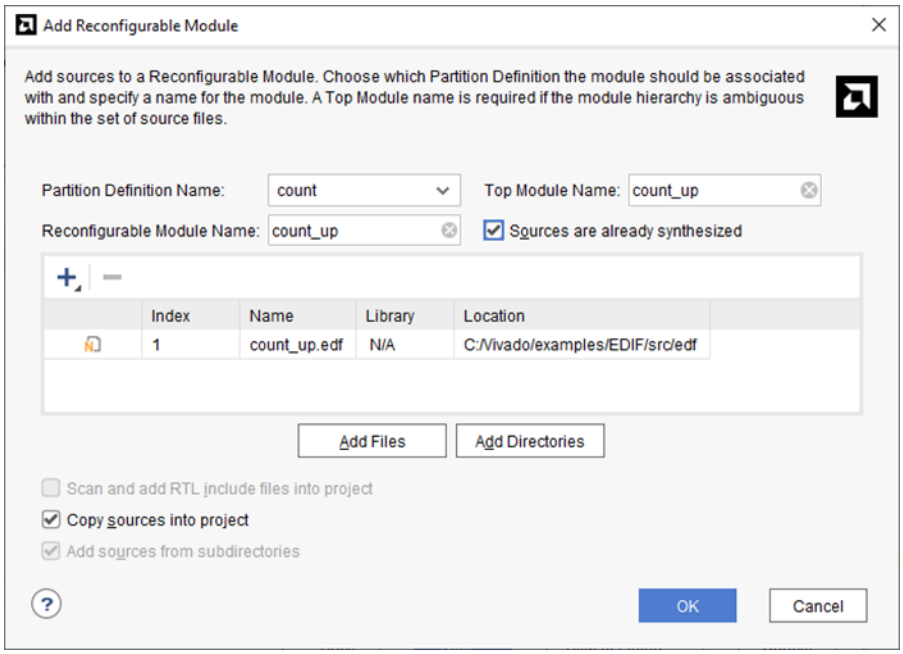
```
ERROR: [Vivado 12-4865] Failed to create reconfig module as Module with an
instance in active top cannot be made into reconfigurable module
```

After the partition is defined, you can move or add sources as needed. For example, if a top-level EDIF or DCP source was present in the instantiation when the partition definition was created, or once a submodule EDIF or DCP has been added to the project, that source must now be moved to the partition definition, specifically within a reconfigurable module. For example:

```
move_files [get_files <file_name>.<edf|dcp>] -of_objects
[get_reconfig_modules <RM_name>]
```

After the partition definition for the RP is created for the first RM, you can use the standard approach to add new RMs through the DFX Wizard. When creating a single new EDIF or DCP RM in the DFX Wizard, you must enable the **Sources are already synthesized** option. When supplying a netlist source, you must also declare the top module name in the Top Module Name field.

Figure 27: **Adding Netlist Sources in the DFX Wizard**



If the new RM is a mix of RTL and EDIF, the full set of files can be supplied in this dialog box. Because the top level is RTL, the Top Module Name and Sources are already synthesized options are not necessary. However stub files (RTL sources defining port names and directions) are required for each EDIF source.

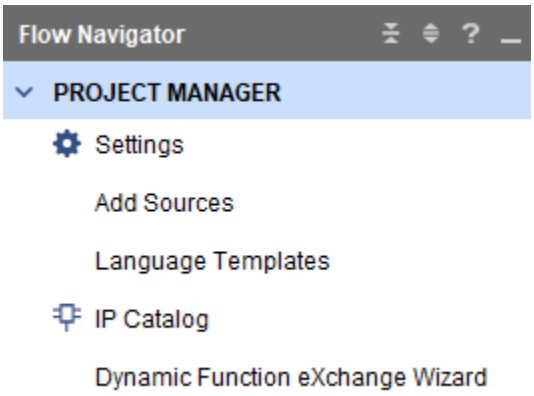
Completing the Dynamic Function eXchange Project Structure

After defining the RPs, enter the full details of the project. This consists of adding more RM for each of the RPs, defining a full set of Configurations that combine RMs with the static design, and declaring the set of runs that will be used to implement all the Configurations. All of these additions are done within the Dynamic Function eXchange Wizard. Any further modifications or additions can be made by returning to the wizard.

TIP: *No actions requested in the Dynamic Function eXchange Wizard will take effect until the Finish button is clicked. You can step forward and back within the wizard until everything is completed to your liking, and you can cancel and throw away all edits at any time.*

Open the Dynamic Function eXchange Wizard by selecting the step in the Flow Navigator or from the Tools menu.

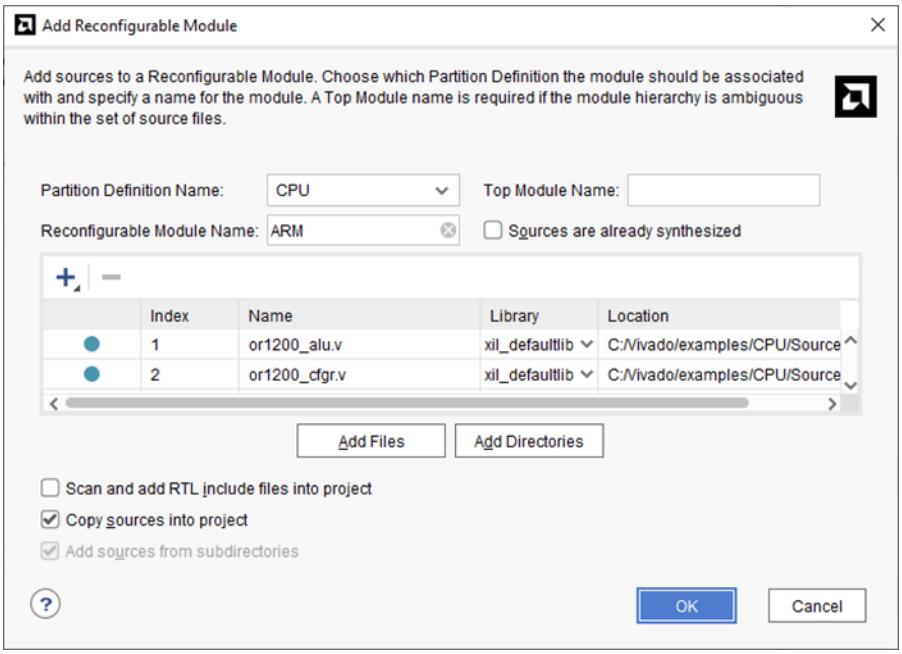
Figure 28: **The Dynamic Function eXchange Wizard in the Flow Navigator**



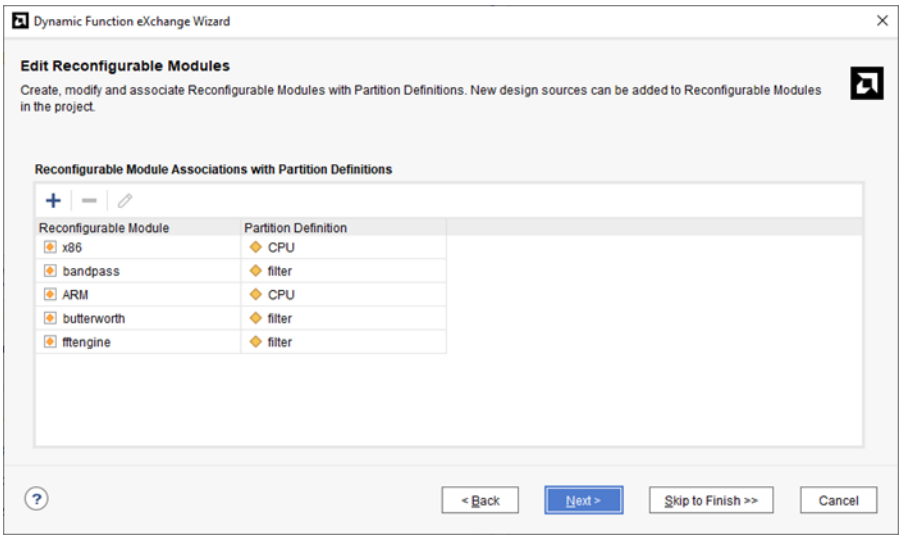
When the DFX Wizard opens, step through each stage of DFX project management.

Editing Reconfigurable Modules

1. After selecting **Next** to progress past the introduction, the first content page allows you to define new RMs for any Partition Definitions (PD) defined. The first RM for each PD has already been included here, if the RTL/netlist source was present when the PD was created. Click the blue **+** to create a new RM and give it a unique name. Be sure to select the correct PD if more than one exists in the design. If netlist sources are selected, select the **Sources are already synthesized** check box and declare the Top Module within the netlist.



2. Repeat this process for all existing RMs for every Partition Definition. If a greybox module is desired, no action is required, because this is a built-in Vivado Design Suite feature that requires no sources. RMs can be edited by clicking on the pencil icon or removed by clicking the red - icon. When all RMs are accounted for, click **Next**.



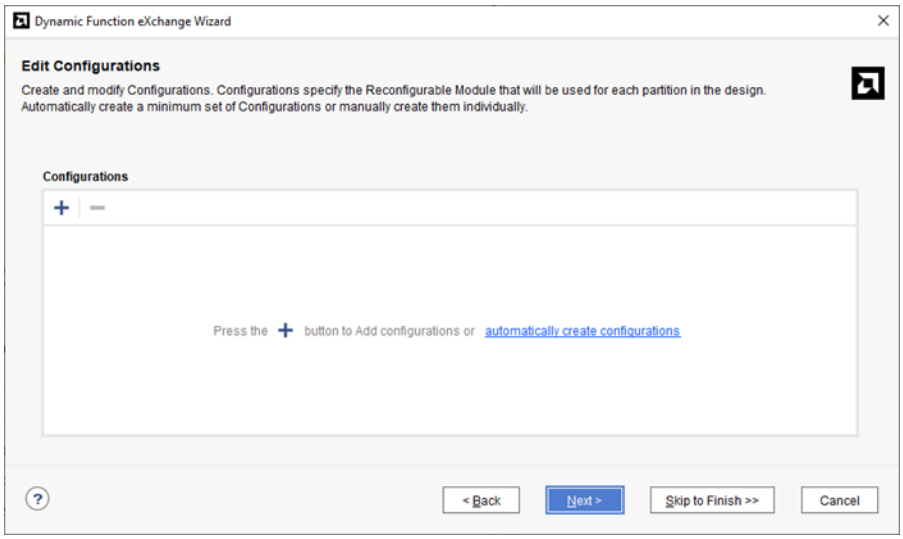
Editing Configurations

With a set of RMs defined, Configurations can be declared. Each Configuration is a combination of the static logic plus one RM per RP; each Configuration is a full design image.

Although each Configuration can be created manually, the simplest path is to let the Vivado tools create the minimum set of Configurations automatically. To do this, select the **automatically create configurations** link in the middle of this screen when targeting 7 series or UltraScale devices. When targeting or Versal devices, select either **Standard DFX** or **Abstract Shell**.Selecting Standard DFX uses full static checkpoints, and the flow described in this section. Selecting Abstract Shell uses the Abstract Shell project flow.

The automatic configuration (or Standard DFX) selection creates as many Configurations as necessary to ensure that all RMs are included at least once. These options are only available if no Configurations have been defined yet.

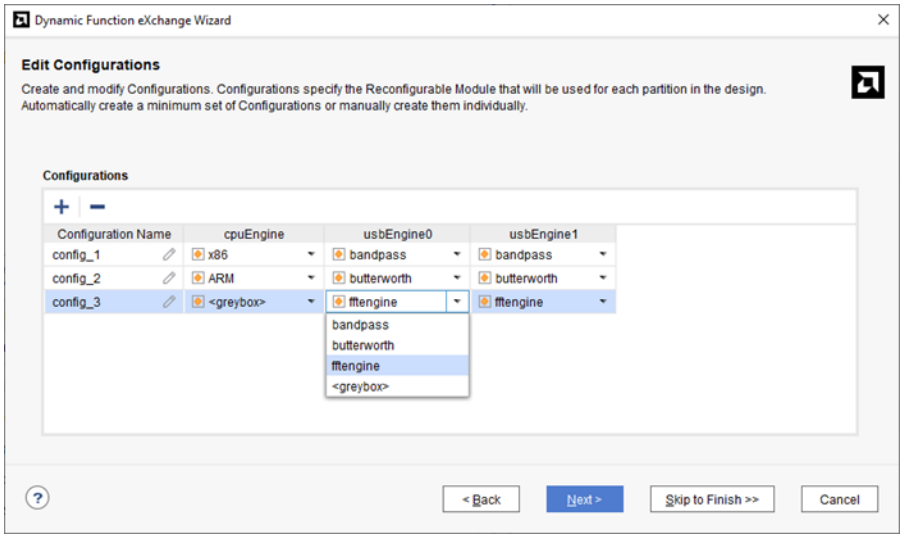
Figure 29: **Edit Configurations Before Creating Any Configurations**



If one Partition Definition has more RMs than another, a greybox RM is automatically used for any RP that has all its RMs covered by prior Configurations. These default Configurations can be modified or renamed, and additional Configurations may be created if desired.

TIP: *Greybox modules are different than black box modules, because they are not truly empty. Greybox RMs have tie-off LUTs inserted to complete legal design connectivity in the absence of an RM and they ensure outputs do not float during operation. To complete legal connectivity, registers can be inserted at clock input ports and clock buffers can be inserted at output ports to drive clock loads. The Vivado tools create these by calling* `update_design -buffer_ports` *on selected modules. Greybox RMs cannot be used in parent runs, because they can lead to sub-optimal results for the static design implementation and/or issues with NoC configuration.*

Figure 30: **Edit Configurations after Automatic Generation of Configurations**



***Note*:** If one RM is used in more than one configuration, the implementation results might be different, because place and route is performed each time, but only if the RM was initially implemented in a child run. RM implementation results are reused if they were originally done in the parent configuration. The `lock_design -level routing` command is set on this reused, implemented RM to lock placement and routing for all subsequent usage in child runs. This allows the Vivado project to track dependencies between parent and child.

If the automatically generated Configurations do not meet your needs, a set of Configurations can be created by selecting the **+** icon, and selecting the name and composition of each one.

Related Information

Abstract Shell Project Flow

Editing Configuration Runs

With all the configurations defined, move to the final screen to manage the configuration runs associated with them. Just like the configurations themselves, the Vivado tools can automatically create a set of configuration runs. The first configuration in the list are defined as the parent, and all remaining configurations are set as children to that parent. Click the **automatically create configuration runs** link to use the auto-creation feature. The rest of this section describes the use of the Standard DFX flow.

The Vivado tools support the Abstract Shell flow within project modes for UltraScale+ and Versal architectures.

Figure 31: **Edit Configuration Runs Before Adding Any Configuration Runs**

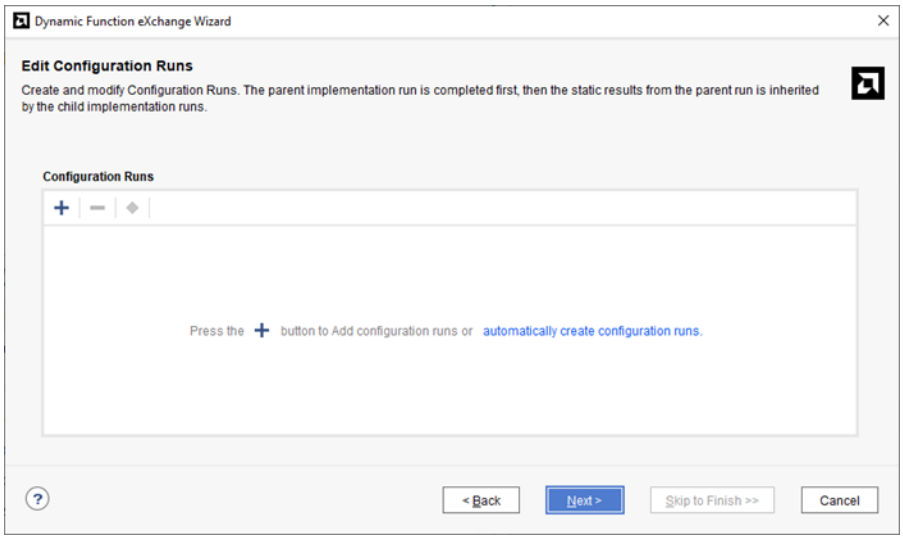
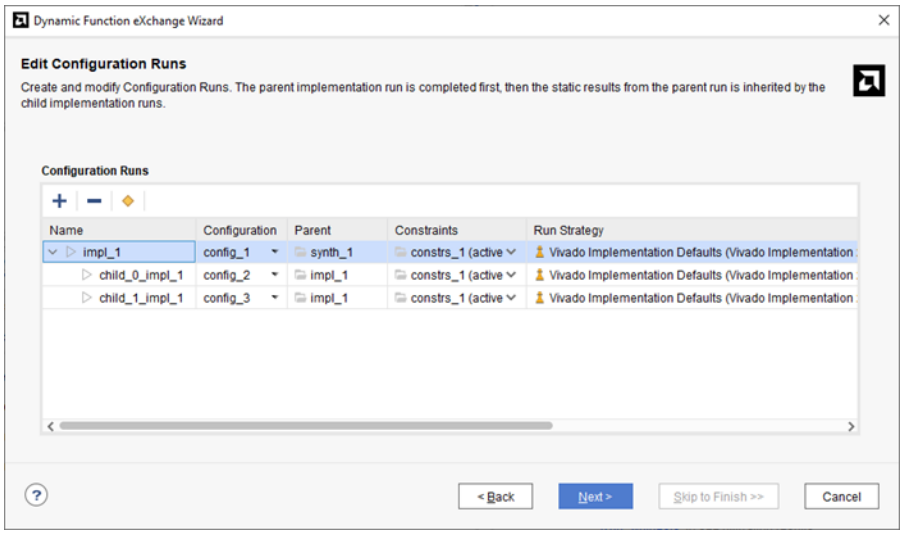


Figure 32: **Automatically Generated Configuration Runs**



***Note*:** When targeting UltraScale+ and Versal devices, columns for DFX Mode (Standard) and RM Instance are shown. The RM Instance field is not used in this context, because the RMs are derived from the Configuration selected for a given run.

This structure assumes that the first configuration is the most critical or challenging. You are free to change the parent-child relationship by setting that value in the **Parent** column. A parent of a synthesis run (**synth_1** here) indicates the configuration (most notably the static part) will be implemented from the synthesized netlist, and a parent of an implementation run (**impl_1** here) indicates the parent's locked static implementation result will be used as the starting point.

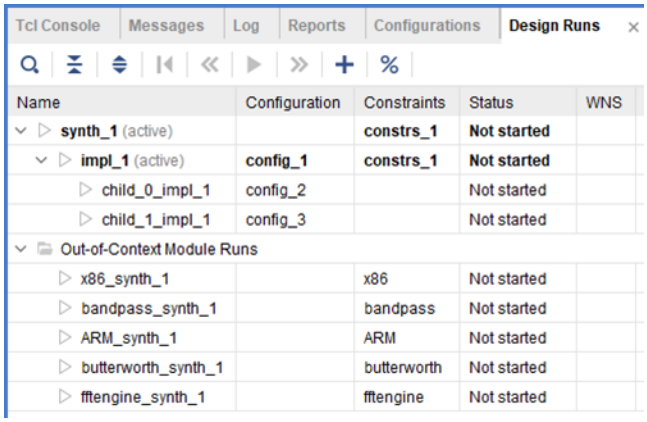
As you explore place and route options, timing closure techniques, and otherwise elaborate on the DFX design, multiple independent parent runs can be used for exploration. Multiple parent runs can be launched in parallel, then child runs can be launched after parent runs complete. Vivado project management handles all the DFX-specific details for creating and storing intermediate checkpoints, including a "static-only" checkpoint for a routed parent run. Ultimately, a single parent run must be selected to establish a golden static implementation result on which all configurations will be based.

IMPORTANT! *To ensure a safe working environment in silicon, a locked static image must remain consistent across all configurations so bitstream generation creates compatible full and partial bitstreams. This is managed in the DFX Project Flow by establishing a parent-child relationship for related configurations. After the initial configuration run in which the complete static design is locked, the static design must not be modified in subsequent configuration runs. If the static design is not identical across configurations, failure during* `pr_verify` *might occur.*

Add new configuration runs by selecting the green **+** icon. When all configuration runs have been created, click **Next**. On the final screen, the number of new elements are listed. Clicking **Finish** will actually perform all the requested changes in the project.

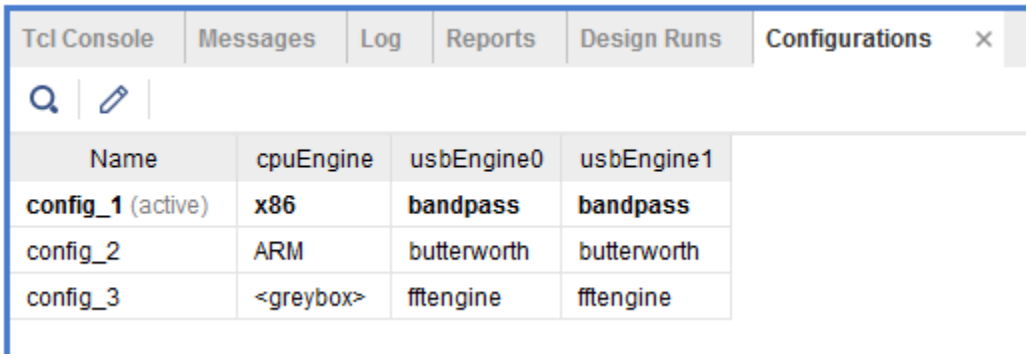
In the Design Runs window, out-of-context synthesis runs are created for each RM, and all configuration runs are generated. Relationships between parent and child runs are shown by the levels of indentation.

Figure 33: **Design Runs for Synthesis and Implementation**



In addition, the Configurations window now shows the composition details of each configuration available in the project.

Figure 34: **Configurations Available in the Project**



Related Information

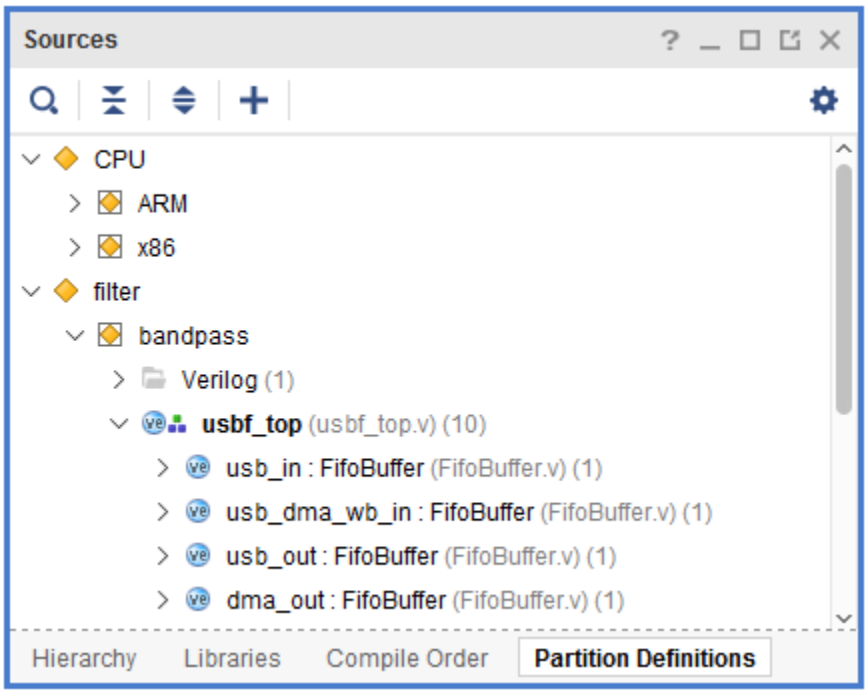
Abstract Shell Project Flow

Adding or Modifying Reconfigurable Modules or Configurations

The Dynamic Function eXchange Wizard is the central mechanism for making any changes to RMs or configurations. This includes adding new RMs, modifying source lists for any RM, creating new configurations or runs, or removing any of the above. When working within the wizard, nothing is saved or executed until the Finish button is clicked, so you can move forward or back through the screens, making adjustments as needed.

When changes to the RTL sources themselves are needed, they can be seen and opened from the Partition Definitions view in the Source window. This shows each RM in the same way as the full design is shown in the Hierarchy view, but scoped to that level of hierarchy and below. This includes all sources and constraints that have been declared for each RM.

Figure 35: **Sources Shown in Partition Definitions View**

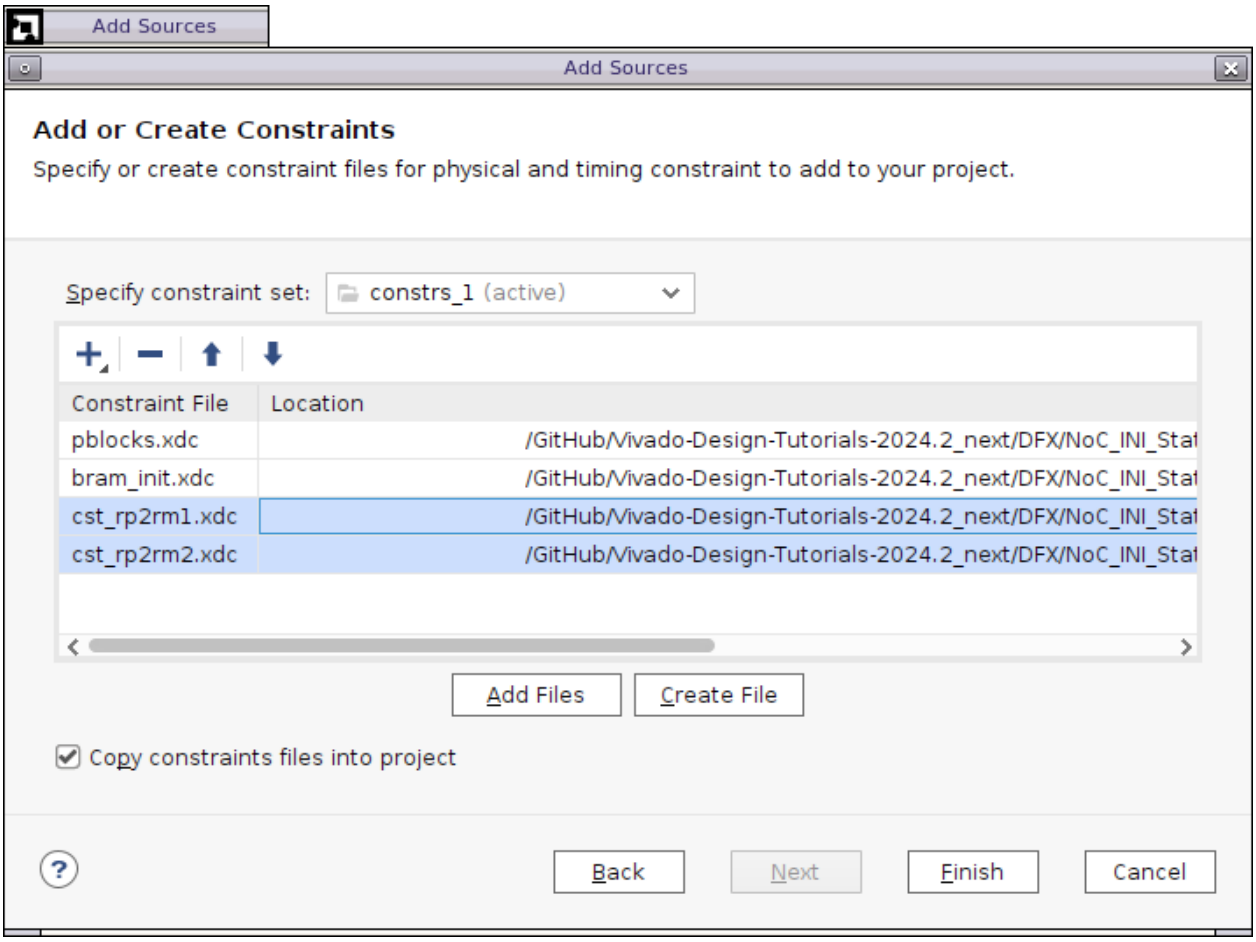


Adding Reconfigurable Module Constraints

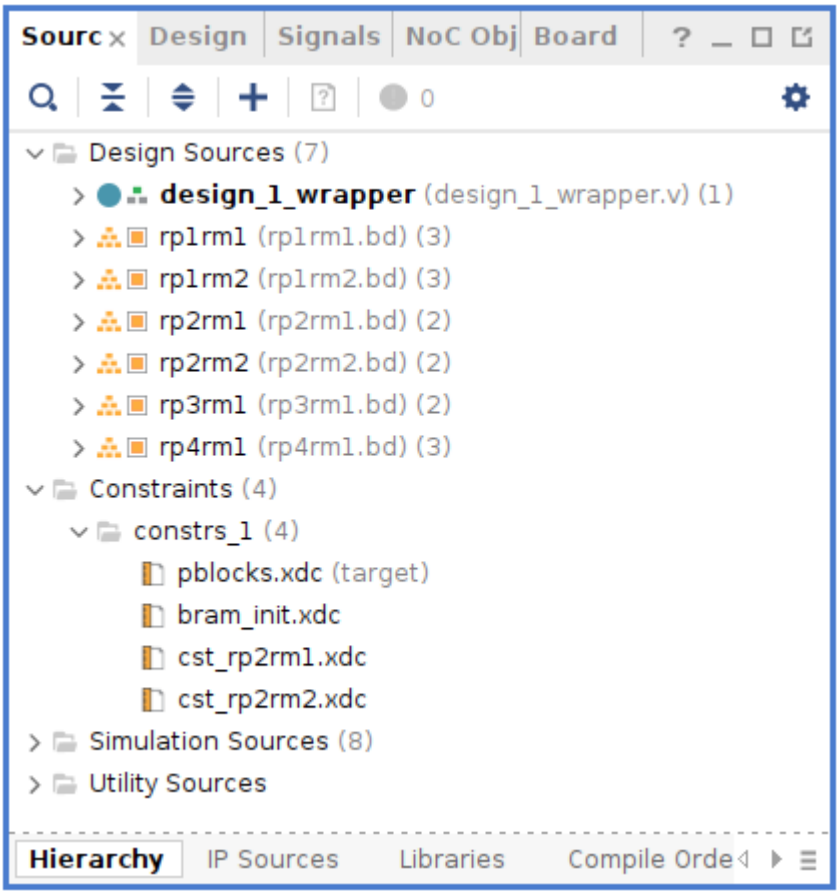
All the constraints in the primary constraint set (constrs_1 by default) are applied through synthesis and implementation of the parent run. All constraints for the static portion of the implemented parent run are then contained within that locked static checkpoint so there is no need to reapply these static constraints for subsequent child runs. While the DFX Wizard shows constraints sets specified for child runs (default or any other constraint set), these child runs do not apply constraints unless explicitly requested in each run by setting the `APPLY_CONSTRSET` run property. Constraints specific to any Reconfigurable Module must be applied for any child run, whether they are constraints in the default constraint set or any new constraint set.

If constraints unique to individual RMs are required, they can be applied in different XDC sources added as sources within each Reconfigurable Module under the partition definition and then scoped to that module. Follow these steps to add and associate RM-level constraints.

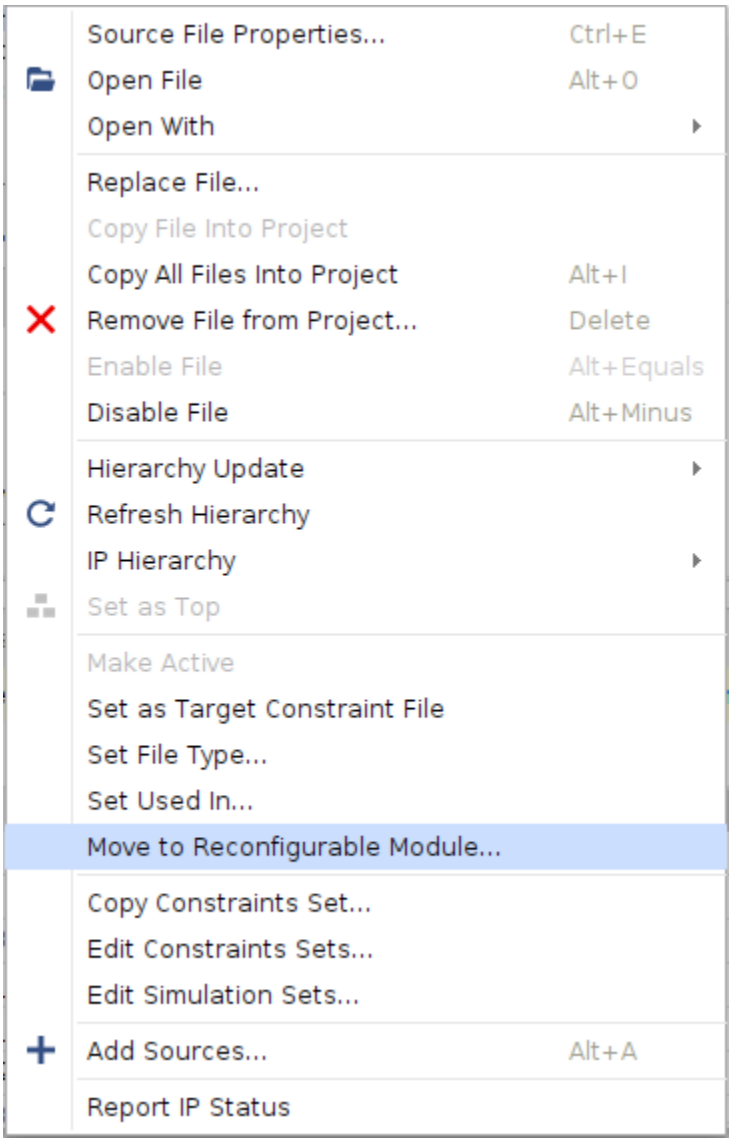
1. Add or create a new constraint file. This can be done by clicking **Add Sources** in the Flow Navigator or clicking the + sign in the Sources window. Select as many XDC files as are required.



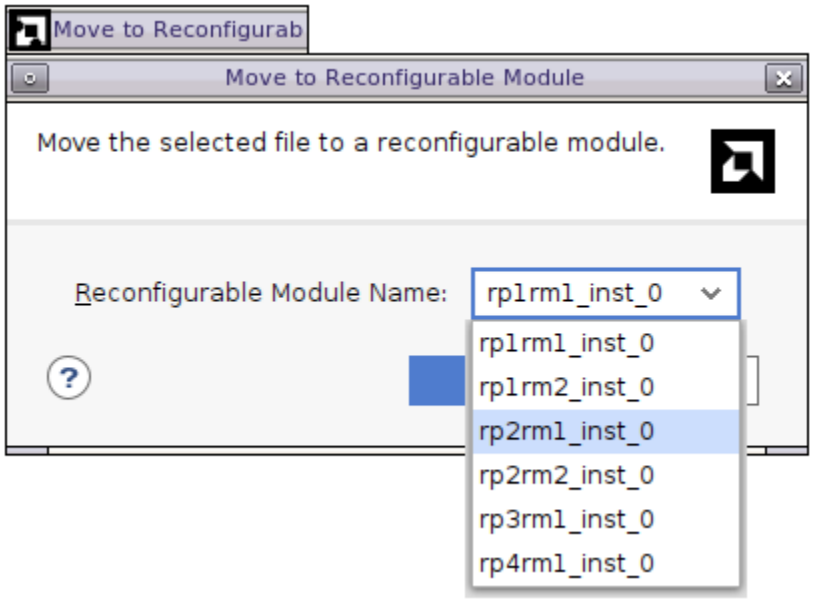
These newly added XDC sources are added to the active top-level constraint file (unless another exists and has been selected in the dialog box in the previous figure), even if the Add Sources action was requested from the Partition Definition pane.



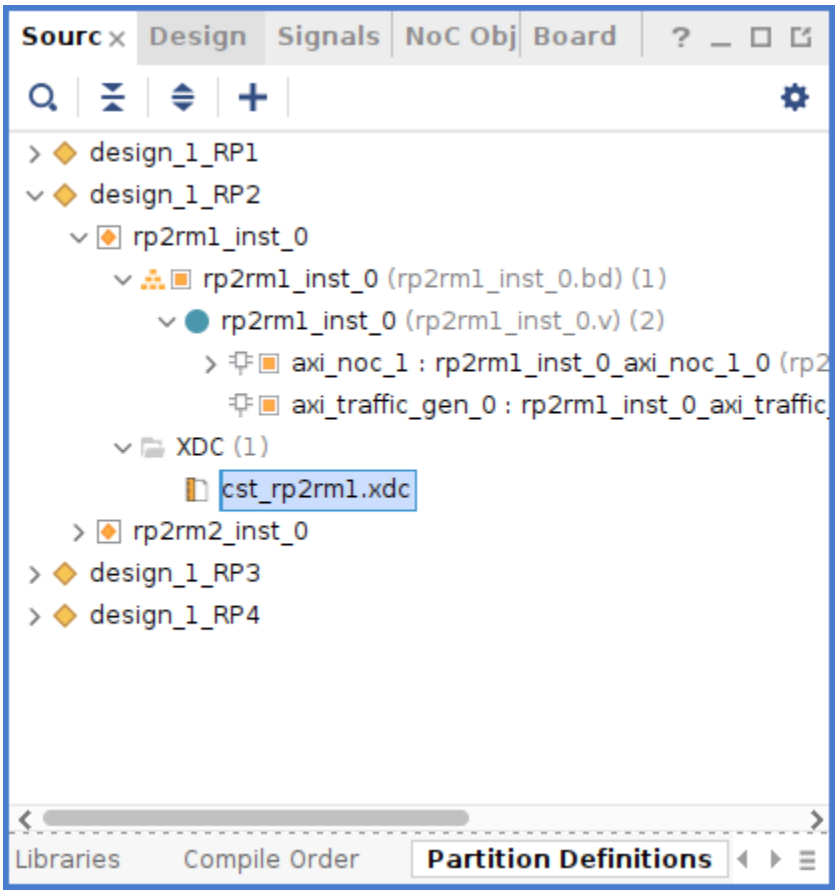
2. For each XDC source, right-click and select the Move to Reconfigurable Module command.



In the resulting dialog box, select the Reconfigurable Module from the list.

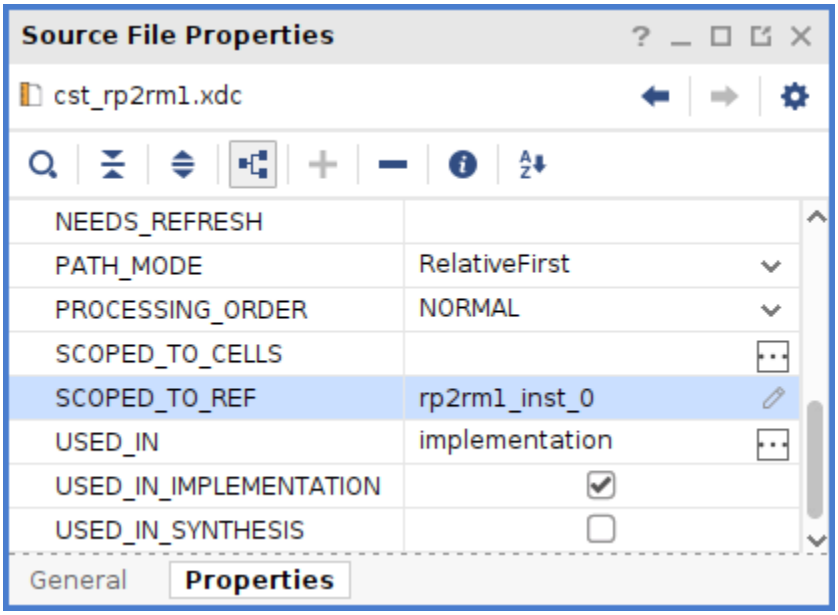


This action removes the selected XDC from the top-level constraint set and places it in the file set for the target RM. The result can be seen by examining the RM within the Partition Definitions tab. Repeat for all RM-level XDC added to the project

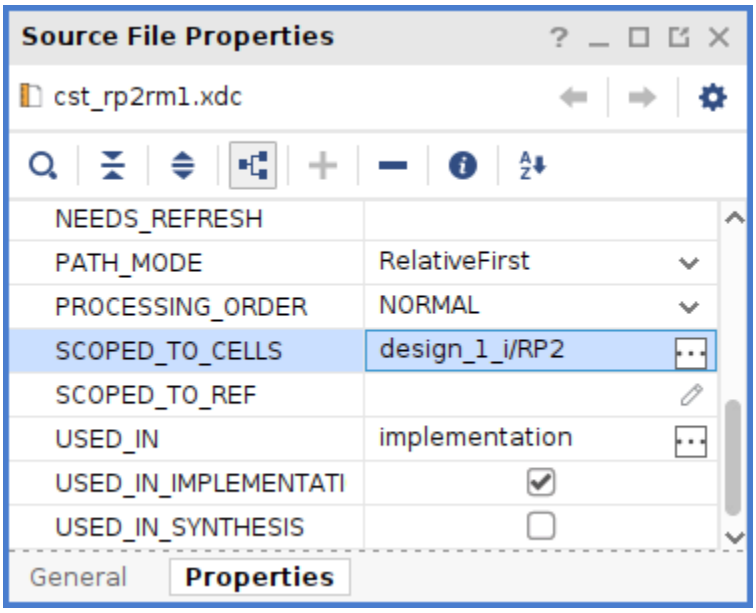


Note: This process can be undone by right-clicking on an XDC file in the Partition Definitions pane and selecting **Move to Design Sources**. This returns that XDC to the active top-level constraint set.

3. Select the XDC within the Partition Definition tab and examine the Source File Properties. Set the `SCOPED_TO_REF` property to the name of the design module. This allows all the constraints within this module to include hierarchical references starting at the RM top rather than the design top. The `USED_IN` properties can be set for these constraint files the same as they can for constraints in the top-level constraint set, and the `PROCESSING_ORDER` can be used to shift when the constraint file is processed by the tools.

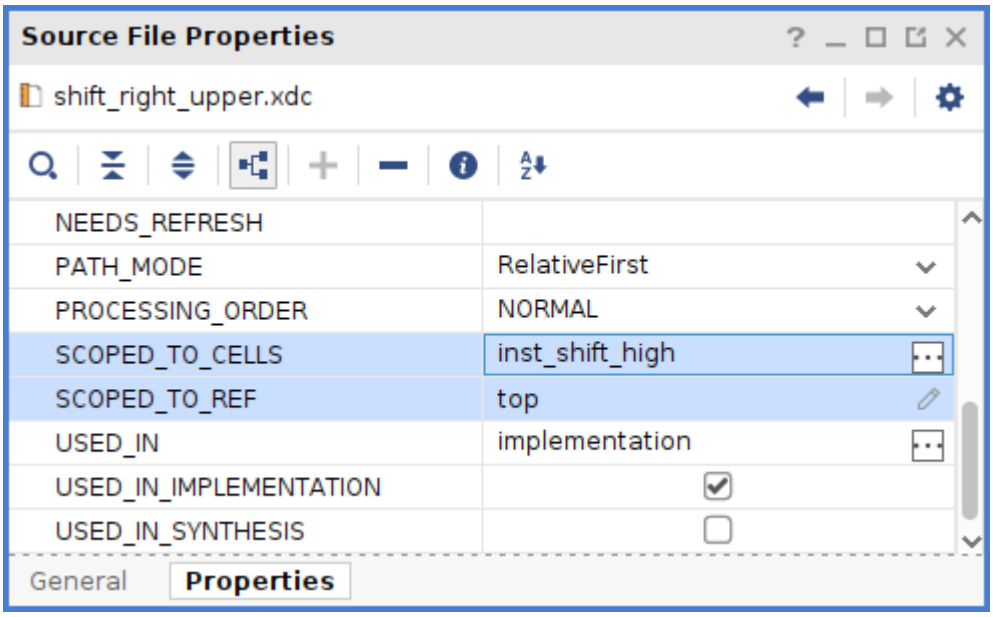


Alternatively, the `SCOPED_TO_CELLS` property can be used to achieve the same result by identifying the hierarchical path to the module instance rather than calling out the module name. Either are valid methodologies for designs with single instantiations of a partition definition.



With the `SCOPED_TO_REF` or `SCOPED_TO_CELLS` value set, all hierarchical instances within the XDC can omit the top-level path to the target RM. This XDC is applied whenever this particular Reconfigurable Module is included in a DFX configuration. No single RM-specific design constraints should be stored in the top-level constraint set; that set should be reserved for static constraints as well as RM-level constraints that are common to all variants within a given RP.

Note: If a single partition definition is used to define multiple RP instantiations in a design, any RM-level constraints is applied for each instance when only using `SCOPED_TO_REF`. This means that any physical location constraints (for example Pblock ranges and LOC assignments) is applied each time. This would be impossible to achieve as each RP has a different physical location, so physical assignments must be tied to a specific RM in a specific RP. This can be done by using the `SCOPED_TO_CELLS` property in addition to the `SCOPED_TO_REF` property.



For more information on constraints scoping, see Constraints Scoping in the *Vivado Design Suite User Guide: Using Constraints* (UG903). For a DFX example, see Lab 3 in the *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947).

By placing constraints that apply only to a single RM within the source set for that RM, you have the ability to build any possible DFX configuration and have the appropriate constraints applied for that parent or child run. No additional manipulation of source sets is necessary.

Adding or Creating IP Sources

IP are permitted within RMs. They cannot be the top level of an RM, but they can exist within any level below the top. Include existing `.xci` or `.xcix` files along with RTL when adding sources to a RM.

IP that exist within RMs can be synthesized either globally or out-of-context. Either value for the Synthesis Options shown below can be selected.

Figure 36: **Specifying Global When Generating Output Products**

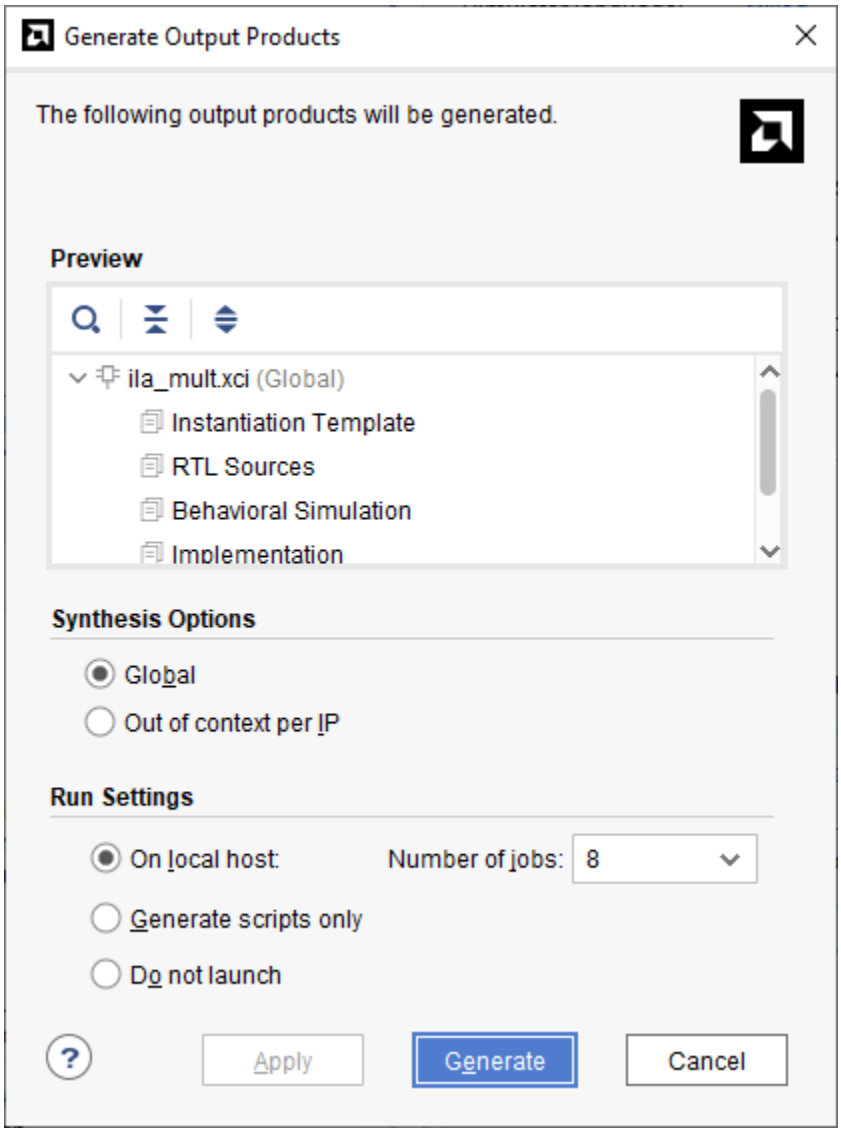
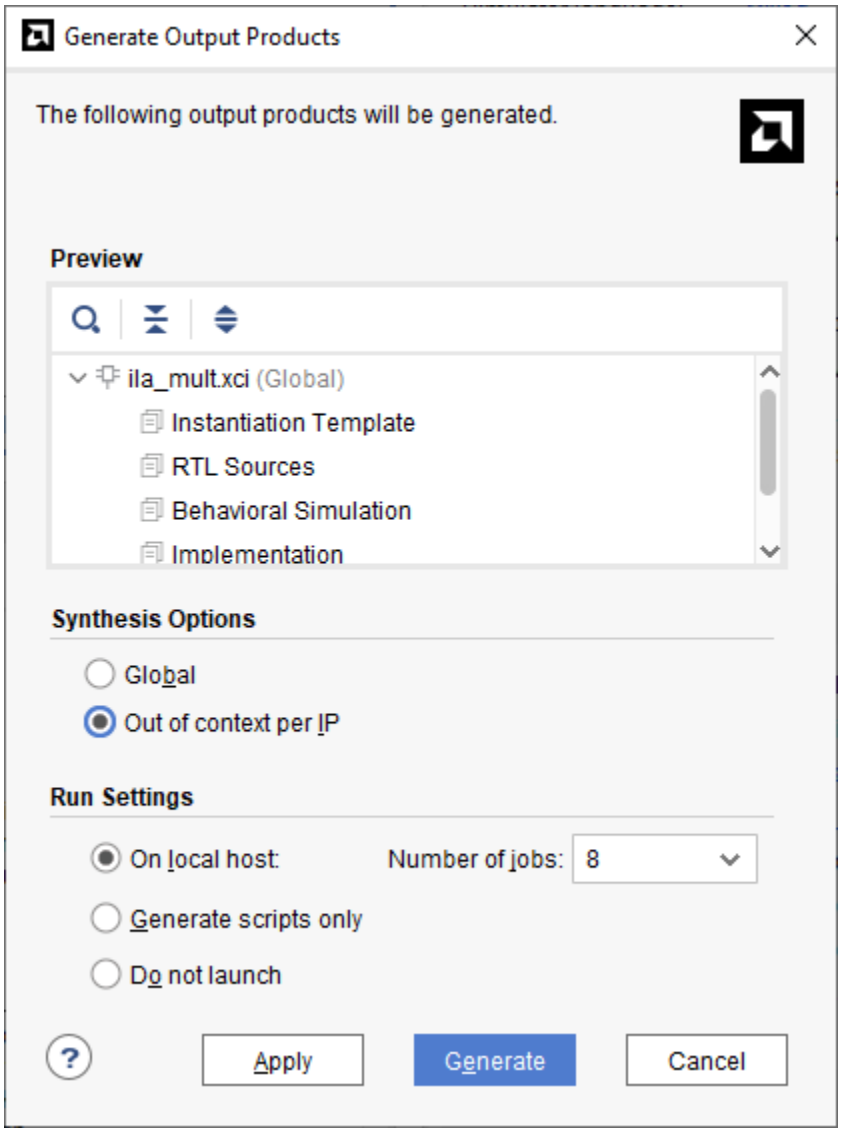
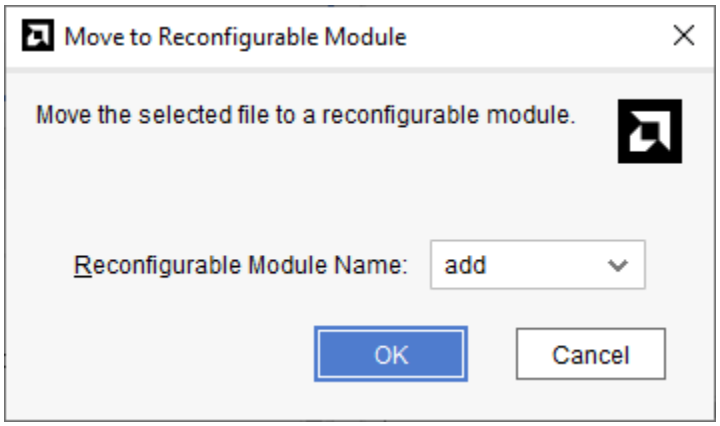


Figure 37: **Specifying Out of Context per IP When Generating Output Products**



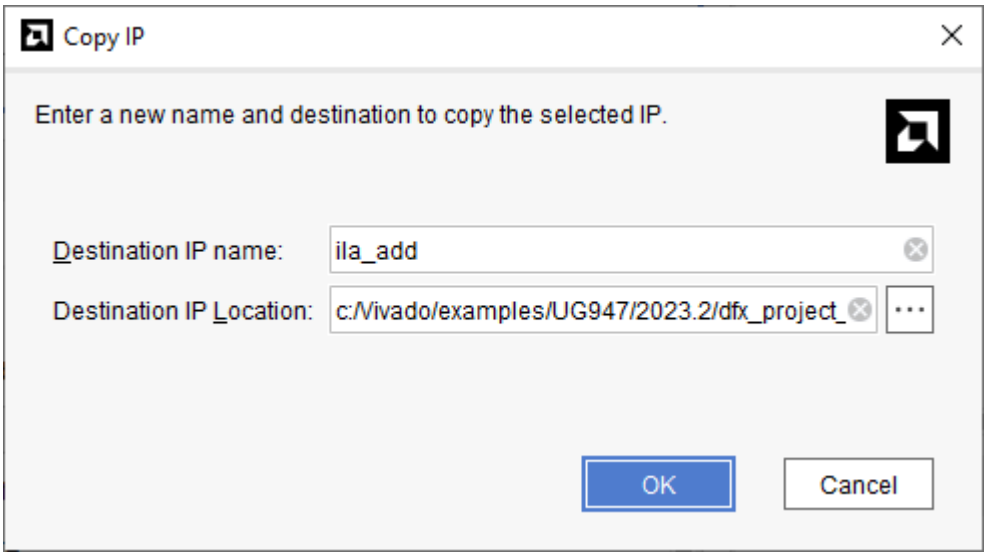
IP can be created from the IP catalog within a DFX project. After the IP has been created, it is added to the primary blockset of the design, as the IP generation flow does not know which RM the IP is for. To assign a new IP to a specific RM, right-click on the IP in the Sources window and select **Move to Reconfigurable Module**. Select the appropriate RM and click **OK**.

Figure 38: **Moving IP to a Reconfigurable Module**



One final requirement is that IP must be unique per RM. If two different RMs each contain the same IP function, two unique instances must be created. The best way to do this is to replicate one IP to create a new identical IP. Right-click on an existing IP and select **Copy IP**. Once created, this new IP must be moved to the target RM as described above.

Figure 39: **Copying IP**



Implementing the DFX Design

With all the necessary Configuration Runs defined, the design can be synthesized and implemented. The Flow Navigator can be used to pull through any steps of synthesis, place and route, and even bitstream generation. The Flow Navigator works on the active run just like a standard flow, but it launches all child runs in addition to the active parent.

One detail that is required for Dynamic Function eXchange designs is a Pblock for each RP. Without a Pblock defined, the following error is issued in `place_design`:

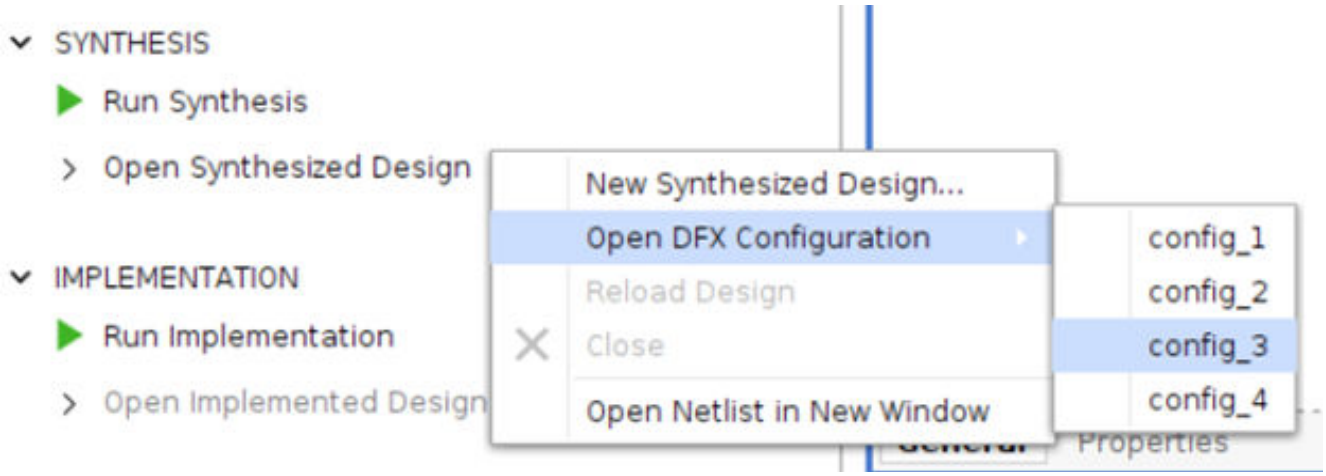
```
ERROR: [DRC 23-20] Rule violation (HDPR-30) Missing PBLOCK On
Reconfigurable Cell
```

If this necessary floorplan is present in a top-level design constraints file, you can pull all the way from synthesis to bitstream generation. If not, the easiest way to create one is to stop and open the design after top-level synthesis.

Opening Post-synthesis Configurations

After all the module-level and top-level synthesis runs, any configuration of the design can be opened to define physical or timing constraints. In the flow navigator, a left-click on **Open Synthesized Design** opens the configuration used in the active parent run. However, a right-click on **Open Synthesized Design** presents additional options, including **Open DFX Configuration**. This selection is followed by a list of all valid configurations that have been declared in the DFX wizard and whose reconfigurable modules have completed synthesis.

Figure 40: **Open DFX Configurations**



Selecting any of these configurations opens the combination of static design with the reconfigurable modules defined in that configuration.

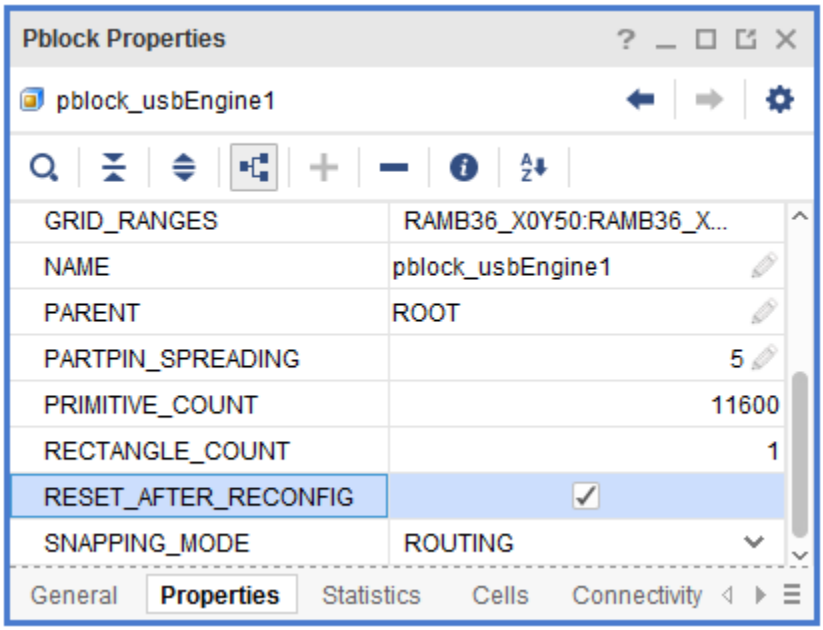
IMPORTANT! *Even though different constraint sets might be available, or that RM-level constraint files might be defined within the partition definitions, all constraints that are added or modified during an interactive session in the Vivado IDE is saved to the target constraint file in the active constraint set.*

Any constraints that are unique to a specific reconfigurable module should be manually moved to a constraint file associated with that RM within the partition definition and then scoped to that hierarchical instance. The only exception to this behavior is the use of Set Up Debug to define the details for ChipScope™. See the Debugging Versal Device DFX Designs section for more information.

With a post-synthesis design configuration open, in the Netlist hierarchy view, right-click the module that corresponds to the RP, and select **Floorplanning → Draw Pblock**.

After you draw the Pblock for a RP, its properties can be seen in the Pblock Properties window under the Properties view. Available here are two options unique to RPs: **RESET_AFTER_RECONFIG** (7 series only) and **SNAPPING_MODE**. The Statistics view reports the resources available and used for the currently loaded RM, so it is important to consider the needs for the other RMs as well.

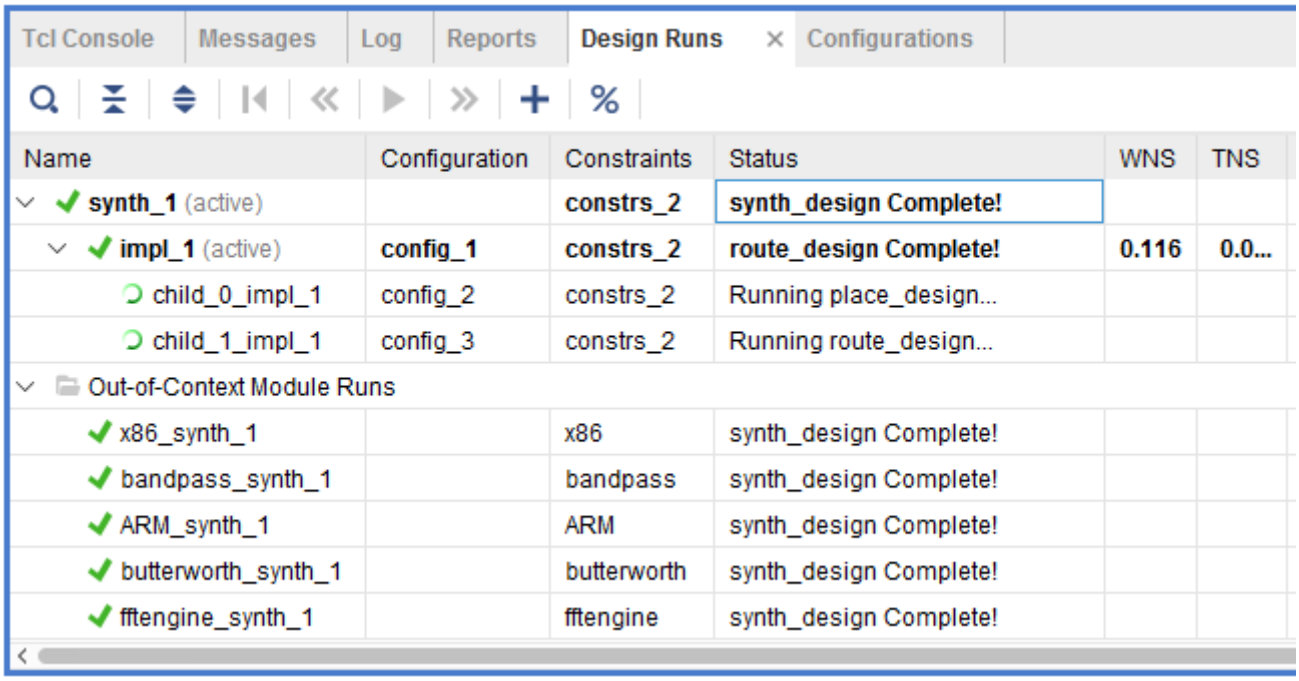
Figure 41: **Dynamic Function eXchange Properties on a Pblock (7 series)**



Once a Pblock has been created for each RP, each Configuration can be implemented. The **Run Implementation** button in the Flow Navigator launches place and route on the active parent run first. Upon completion, all child runs will be launched in parallel, using the static design results of the parent as a starting point.

The Vivado project takes care of the underlying details of the DFX solution. Database management is one of these details. Upon completion of the parent run, the routed database for the entire design is saved, as well as a cell-level checkpoint for each RM. Then Vivado calls `update_design -black_box` to carve out each RM, resulting in a static-only design checkpoint, which is the basis for all of its child runs. When child implementations runs are launched, Vivado assembles the configuration from the static-only routed parent checkpoint and post-synthesis checkpoints for each RM. At this time, only routed module checkpoints from the parent run can be reused in child Configurations; if the same RM is selected for one RP in multiple child runs, the results will be different.

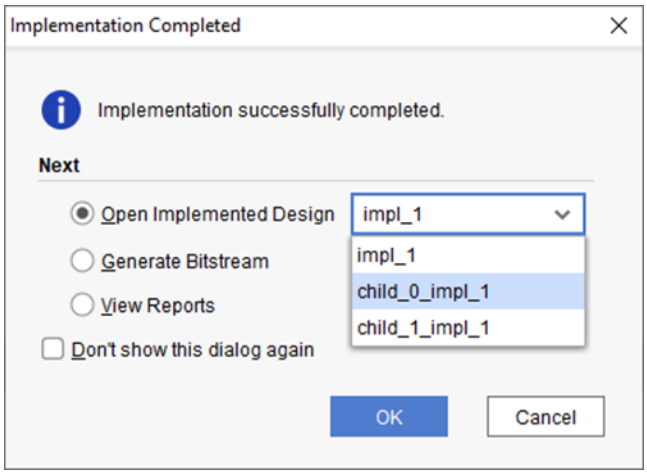
Figure 42: **Implementing Multiple Configurations in Parallel**



Just as with a standard project, Vivado tracks dependencies between runs. When design sources, constraints, options or settings are modified, any synthesis or implementation run that depends on them are marked out-of-date. One example: if an RTL design source for one RM is updated, that out-of-context module run will be marked out-of-date, and any Configuration Runs that include that RM will also be marked out-of-date. Another example: if any implementation option for a parent Configuration Run is changed, it and all child runs will be marked out-of-date.

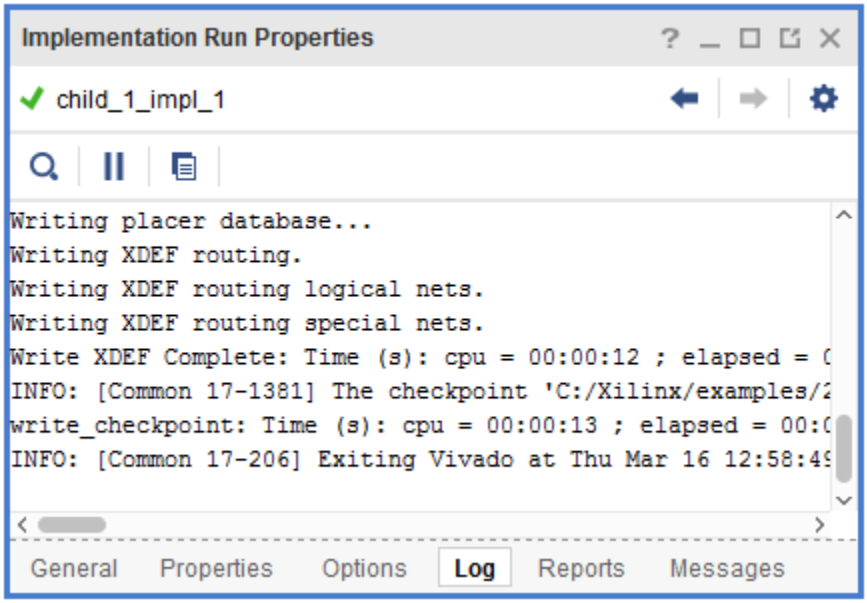
Only parent Configuration Runs can be set as active. The Flow Navigator acts upon the active run, but in the DFX flow, all child runs are also included in whatever action is requested. Pop-up messages (completed run, error, etc.) can relate to multiple runs but default to the parent run. Use the pull down selection to choose the desired implementation run.

Figure 43: **Implementation Completed Dialog Box**



Everything shown in the different windows relate to the active parent run. In order to see details about a child implementation run, select that run and look at the Implementation Run Properties window to see all the inputs (properties, options) and outputs (log, reports, messages) for that specific run.

Figure 44: **Implementation Run Properties Window for a Child Run**



Related Information

Debugging Versal Device DFX Designs

Generating Bitstreams

Once all desired Configurations have been placed and routed, bitstreams may be generated. Just as with Implementation, the **Generate Bitstream** button in the Flow Navigator may be used. This launches `write_bitstream` for all child runs as well as the active parent. A local right-click call to `write_bitstream` on any Configuration is also available.

The `pr_verify` utility is automatically called prior to `write_bitstream` on each child Configuration run. This routed database is compared to the parent database to ensure all DFX rules have been met. The results of this check are stored in the run directory under the name `<impl_name>_pr_verify.log`.

By default, a full design bitstream and all partial (and for AMD UltraScale™, clearing) bitstreams are generated for all routed configurations. You can request specific bitstreams only by utilizing the `write_bitstreams` options available. Under the Write Bitstream options category in the Options pane for the Implementation Run Properties, use the **More Options** field to select one of these options:

- `-no_partial_bitfile` generates only the full configuration file and no partial bitstreams.
- `-cell <cell>` generates only the partial bitstream for the requested cell.

Supported/Unsupported Features

This section lists the current lists of supported and unsupported features for DFX Projects.

Supported Features

- Device support: All 7 series, AMD Zynq™, UltraScale, UltraScale+ and Versal devices supported by the Dynamic Function eXchange flow.
- Source types for RMs: RTL, DCP, EDIF, XDC, XCI, XCIX.
  - XCI or XCIX (AMD IP) cannot be the top level.
  - Netlist source types (EDIF) are limited to a single file that represents the entire RM.
- Module-level constraints must be scoped to the hierarchical instance.
- Greybox (black box module with tie-offs) implementation can be done
- An extensive set of Design Rule Checks can be issued from within the project environment.
- All synthesis and implementation design switches can be used.
- PR Verify is automatically called before bitstream generation for any child configuration.

Unsupported Features

The following features are not currently implemented:

- Block designs cannot be included within RMs when the top-level of the RP is an RTL instantiation.

  IMPORTANT! *To include block designs as an RM, use the Block Design Containers flow.*

- Simulation is not supported from within the project.
- In the project flow, Create Partition definition cannot be set on RTL modules that have EDIF netlists as submodules. This RM is created from the RTL sources currently residing in this level of hierarchy

Known Limitations

- Converting a project to DFX cannot undone. The only way to return to a flat non-DFX project is to create a new one.

  ***Note*:** Partition definitions can be removed by using the `delete_partition_defs` command on the Tcl Console.

- Reuse of implemented RMs from a child run is not supported. Only implementation results from RMs from the parent run can be reused in a child run.
- Child implementation runs cannot be set active. Flow Navigator actions work on only the parent run, or the parent and all child runs, depending on the action.

IP Integrator Using Block Design Containers

A feature called block design containers (BDC) allows you to segment designs into multiple block designs, enabling modular and team-based design flows, including DFX. Access to critical Versal architecture features such as the CIPS, NoC, and SmartConnect IP must be managed through block designs in IP integrator. Versal architecture DFX designs must be processed through IP integrator if NoC or SmartConnect IP are to be dynamically reconfigured. However, all architectures are supported by the BDC flow. For more information on IP integrator and Block Design Containers in particular, refer *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994).

BDC expands the hierarchical blocks capability in Vivado IP integrator. A hierarchical block creates a new level of hierarchy in a block design (BD) that can contain any number of user selected IP blocks. The BDC feature turns a hierarchical block along with the content inside into a separate block design itself. The resulting BD is defined as a `.bd` design source and can also be used as part of another block design project.

A BDC can be set as reconfigurable, turning it into a Reconfigurable Partition (RP) and enabling each design source within it to be considered a RM. The DFX Wizard populates each RP with all possible RMs for each RP before defining Configuration and Configuration Runs, similar to the RTL project flow for DFX.

TIP: *The Design Runs capabilities and features for a DFX design is the same between RTL-centric and IP integrator-centric design flows.*

Top-Down Flow

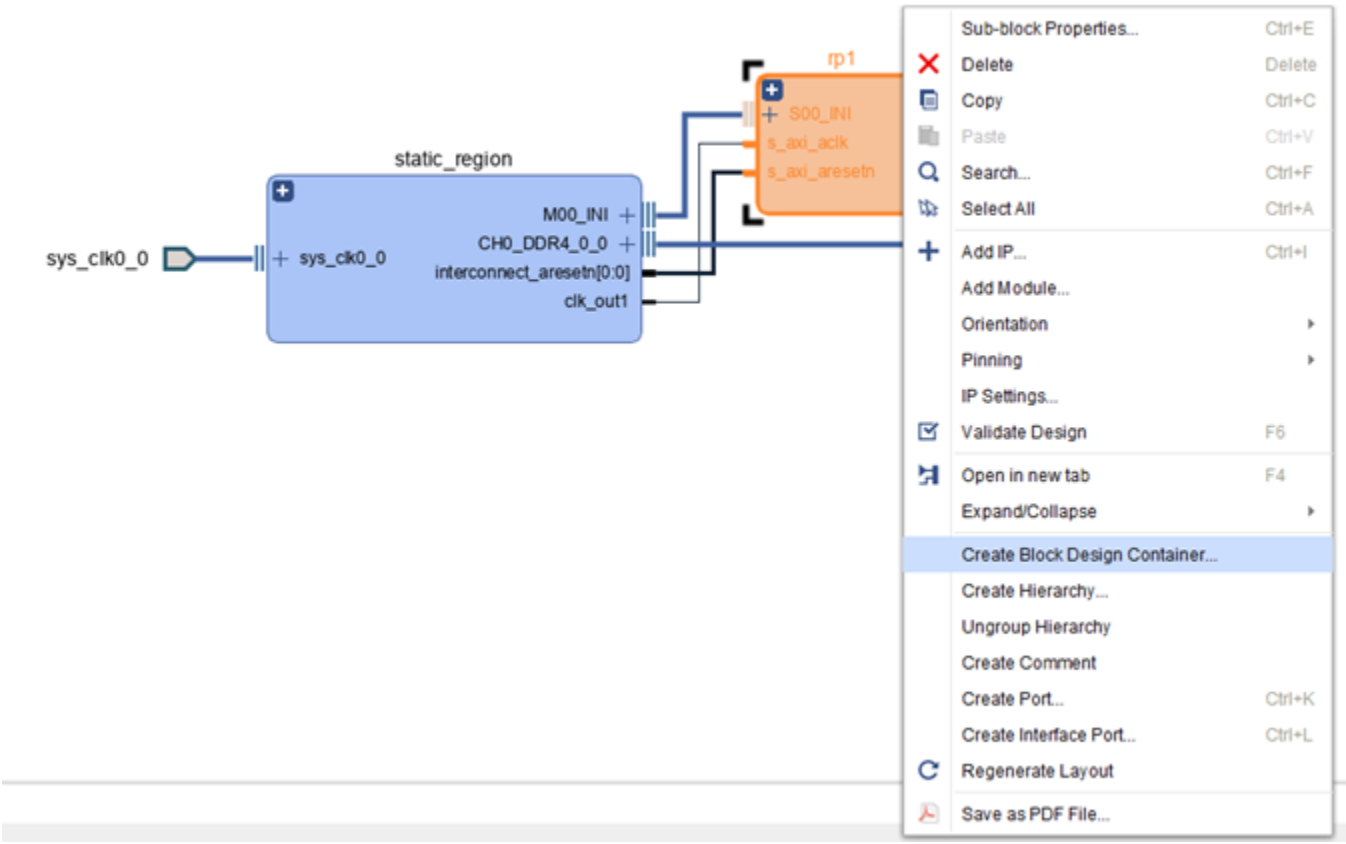
One way to introduce BDCs is by converting a level of hierarchy. To create a level of hierarchy in an existing BD, select one or more elements by using **Ctrl+click**, then right-click and select **Create Hierarchy**. Provide a unique appropriate name for the hierarchy. The name becomes the RP name. After you create this hierarchy you can undo this action by right-clicking and selecting **Ungroup Hierarchy**. You can move IP instances in or out of the hierarchy by dragging them in or out of the hierarchy region on the canvas.

Validate the BD after you create the level of hierarchy by selecting **Tools → Validate Design**. Alternatively, click the **Validate** icon on the block design toolbar, or use the F6 function key.

***Note*:** Any validation errors must be resolved before continuing.

To create the BDC, right-click on the hierarchical instance and select **Create Block Design Container**. The resulting dialog requests a name for the resulting block design. Provide a name appropriate for a RM, as the logical elements within this level of hierarchy becomes the first RM.

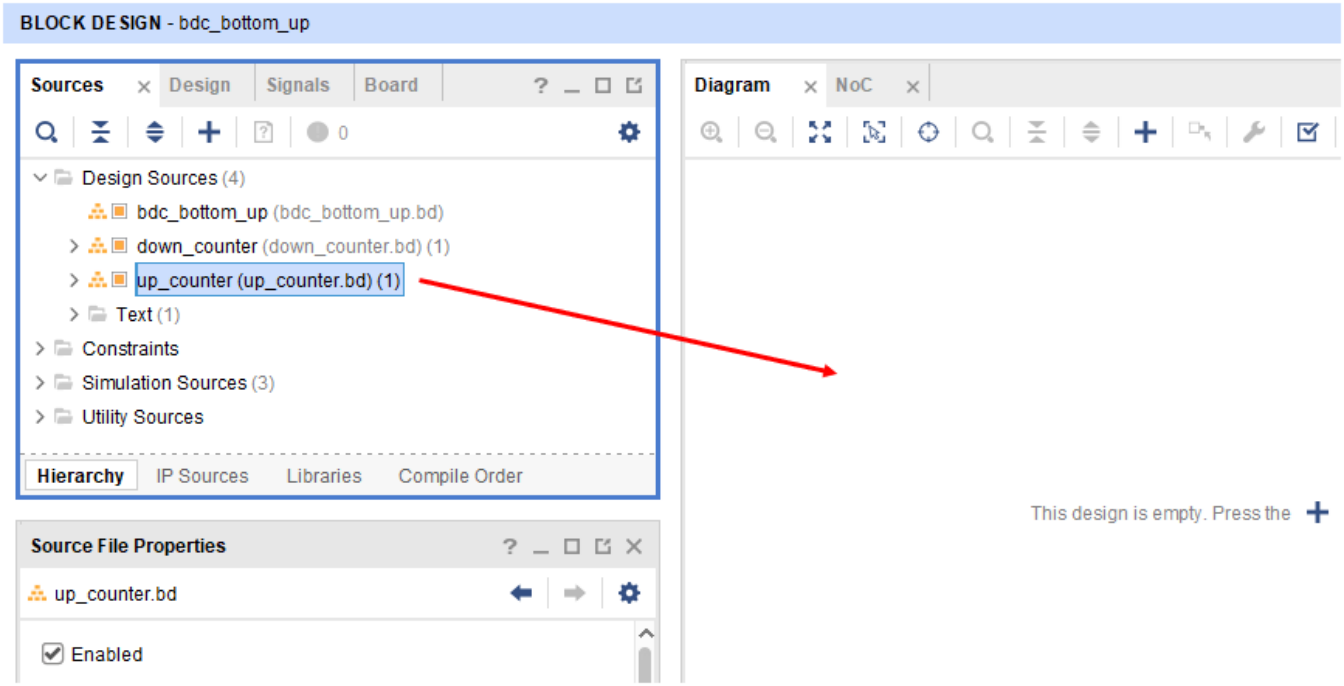
Figure 45: **Create a Block Design Container From a Level of Hierarchy**



Bottom-Up Flow

The bottom-up flow is another entry method to the BDC feature in IP integrator. If a block design for a RM (or even just a sub-module design) already exists in another project or in a storage repository, it can be added to the IP integrator project as a design source. To create a BDC for that BD, drag and drop the design source from the **Sources** window to the canvas of the top-level BD.

Figure 46: **Bottom-Up Flow in a Block Design**

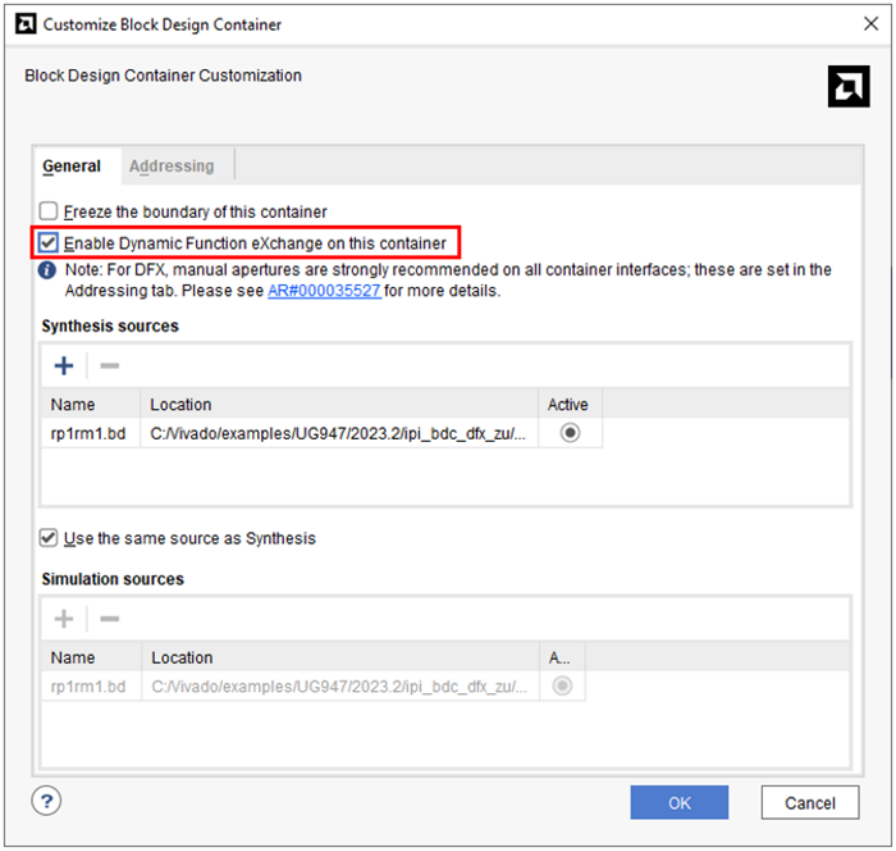


TIP: *This can be done on a blank canvas or one that already contains IP for the static part of the design.*

Turning a Block Design Container into a Reconfigurable Partition

A BDC itself is simply a mechanism to enforce hierarchical processing of the overall IP integrator project. Each BDC can be synthesized out-of-context, but implementation at this point still flattens the design for optimization, place and route tasks. To turn a BDC into a RP, double-click on the BDC to open the Customize Block Design Container dialog box. Check the **Enable Dynamic Function eXchange on this container** box. The current block design is now a RM, as are any new design sources added to this BDC.

Figure 47: **Turn a Block Design Container into a Reconfigurable Partition**



Updating Block Design Container Options

Updating the BDC settings within a DFX project can cause the out-of-context (OOC) synthesis results for one or more RMs to become out of date. You can select the **Freeze the boundary of this container** option to lock the RP boundary and help reduce the chance of OOC runs becoming out of date.

When using this option, be aware of the following:

- Changing the active variant makes the OOC synthesis runs out of date even with a locked BDC. This is because the active variant defines the boundary for the BDC. All of the other RMs in the BDC need to accommodate themselves to fit within this boundary. Also, Smartconnect Master IDs might change based on the topology of the new active variant.
- Changing from DFX to BDC or vice versa also makes the OOC synthesis runs out of date because of changes needed in how the Smartconnect IP is elaborated.
- To maintain an OOC synthesis run status, the DFX BDC must be in a locked state.

Removing a Block Design Container

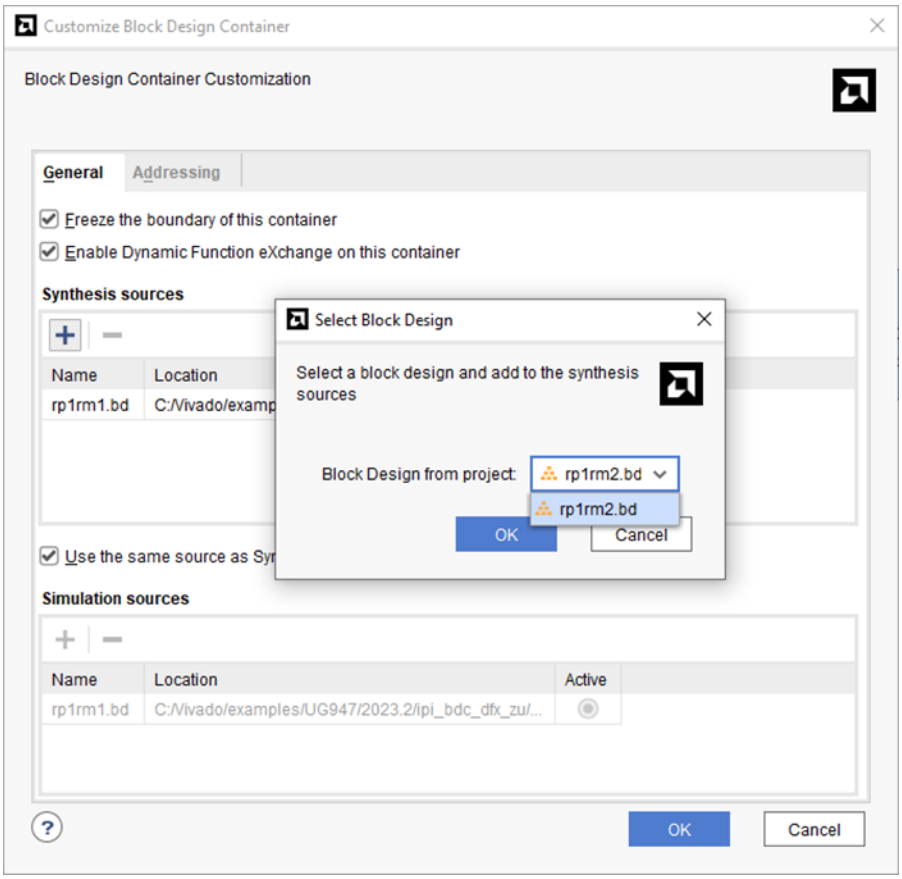
Currently, there is no option or command to remove a Block Design Container, returning it to a standard level of hierarchy. When the container is created, all the IP and connectivity are moved to a new design source in the project, and there is no current mechanism in Vivado to undo this action.

***Note*:** A BDC can be deleted from the top-level block design, but ensure to manually recreate any portion of the design.

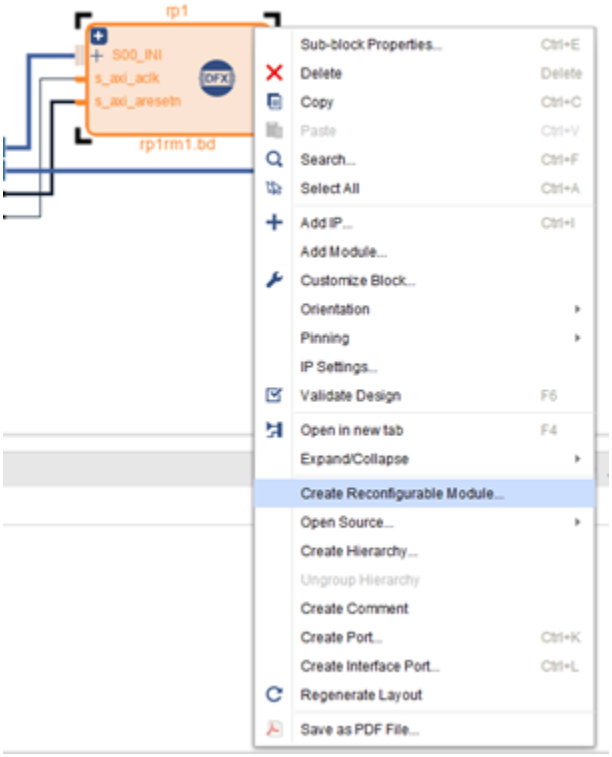
Add New Design Sources to the Block Design Container

There are two ways to add new block designs (for which the DFX designs are each RMs) to an existing BDC:

1. If a BDC exists on the static design canvas, a new BD can be added from block designs that are present in the project. Double-click on the BDC to see the list of current BDs already assigned to that BDC. Use the + icon to pick from a list of compatible BDs currently in the project and unassigned to that BDC. If port lists do match the BDC definition, a BD will not appear in the list.



2. New block designs can be created from an existing BDC. Right-click on the DFX-enabled BDC and select **Create Reconfigurable Module**. Supply an appropriate name for the new RM and click **OK** to generate a new BD source. The new block design will contain no IP, but all the ports have been imported from the BDC definition, so it will be compatible from the start.



Update Port Boundaries within a BDC

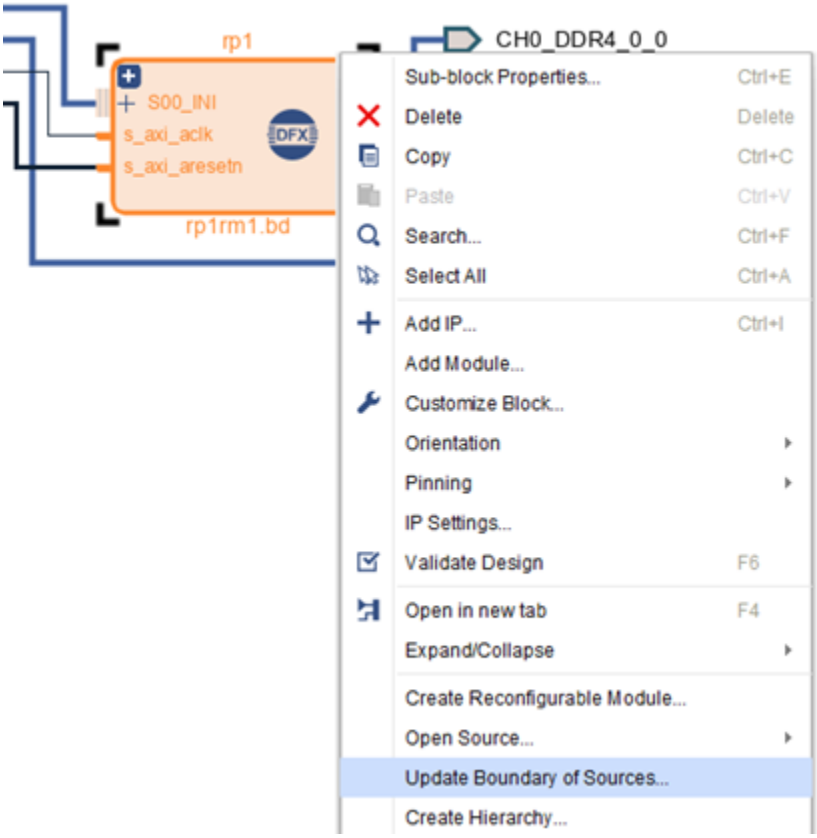
In order to maintain compatibility between RMs for a Reconfigurable Partition, ports must be consistent.

IMPORTANT! *Each design source contained in a block design container (BDC) must have the same port list otherwise a design rule check will be flagged upon validation.*

```
[BD 41-2125] There are boundary differences for </RP_inst> between 'rm1.bd'
and 'rm2.bd':
* Port with name 'reset' exists in rm1.bd but not in rm2.bd
[BD 41-2315] Block Design Container </RP_inst> is enabled for Dynamic
Function eXchange,
which means that all the Synthesis sources should settle upon the same
boundary.
Please resolve earlier errors or disable Dynamic Function eXchange and try
again.
```

Ports can be modified in one RM BD, then the changes can be pushed to all other RMs for the selected BDC. Follow these steps to propagate the changes across multiple RMs:

1. Edit the active block design within a BDC. The active block design is indicated by the radio button selection in the BDC GUI. Add or remove ports as necessary, then validate and save the design.
2. Once this is done, a yellow banner alert is generated in the project, and the top-level block design is shown as modified. Click the **Refresh Changed Modules** link in the banner to update the BDC instance in the top level.
3. The BDC instance shows new and modified ports. Make connections to complete the top-level block design. If you validate the BD, you receive the Critical Warnings listed above, as expected.
4. Right-click the BDC to select **Update Boundary of Sources**. Use the pull-down menu to select the appropriate BD for the master instance. All ports from this BD are echoed in all BDs selected in the main body of the dialog box. Click **OK** to apply the changes.



TIP: *This action runs the* `update_bd_boundaries` *Tcl command.*

Finally, return to all the other block designs used within this BDC. Each has new or modified (or removed) ports available within the BD. The Update Boundary feature provides the ports but not connect them.

Note: Ensure to define how to use new ports.

Using Address Apertures

Each BDC has an addressable space available for connecting masters and slaves. The DFX BDC boundary apertures can either be manually specified, or left onto IP integrator to automatically infer them.



TIP: *For DFX designs, the Manual setting is advised.*

- **Auto:** Apertures will be automatically inferred by looking at all the design sources in the BDC. If an RM has apertures specified manually on the boundary, these will be used to compute the BDC apertures for the container. If not, boundary assignments in the child will be used for calculations. This will occur regardless of whether a bottom-up or top-down approach is used.

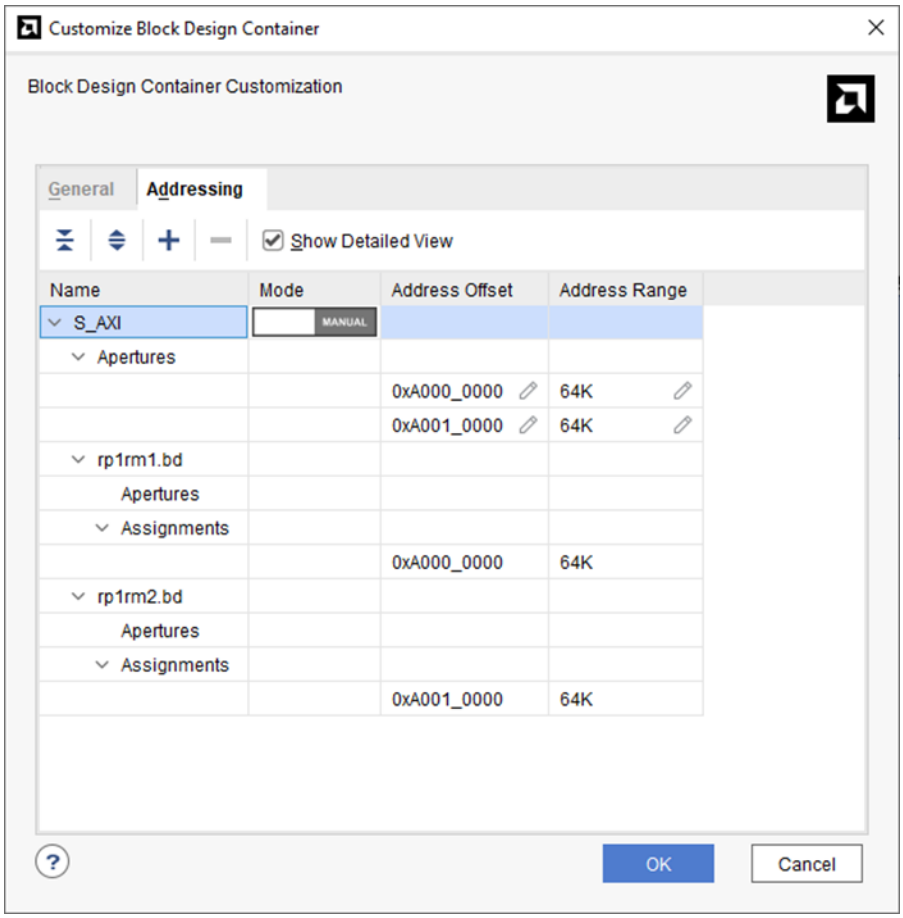
  If your static design has already been implemented with certain auto-computed apertures, adding a new RM source to the BDC will cause IP integrator to re-compute those apertures. If the calculated apertures are different than the previously computed apertures, the static result will be marked out of date and will need to be re-implemented.

- **Manual:** Apertures (either manually specified or derived from address assignments in the child) in the child will be validated against the manually specified apertures. If they are not compatible, DRCs will be issued.

  Any new RM created using the Create Reconfigurable Module command for a BDC will inherit the BDC-specified apertures. This is more relevant to the top-down flow.

  Save Block Design As has been enhanced to Freeze the boundary of the new BD, which copies apertures from the current BD (presumably the default RM source), along with rest of the boundary and freeze it. In bottom-up flows, it ensures that newly created RMs are always restricted to match the BDC boundary.

Figure 48: **Viewing the Address Aperture for a BDC**



Always use manual apertures for DFX designs to ensure that all the RM addressing fits within the top block design address space. If manual apertures are not used for a DFX design, the Vivado tools report a warning.

You can validate block designs using `validate_bd_design -assign_dfx_addressing`. The `assign_dfx_addressing` switch is an optional value used to consolidate addressing across RMs in DFX blocks. This argument makes every RM active in the context of the top and reassigns addressing to make the RM fit into the top. Manual apertures guide the flag on how to change the RM addresses.

Aperture overlaps on interfaces passing though the same network are not allowed, because SmartConnect and NoC instances connected to these interfaces would fail to elaborate. IP integrator overlapping errors can occur when using Auto apertures for multiple RP designs in which the Vivado tools need to adjust the addressing to meet a power of 2 and minimum 64K requirement.

Avoid DFX BDCs that use only a single routing bridge IP (such as a SmartConnect or NoC IP), which is also known as a pass-through RM. The change in routing that can occur when switching RMs between a static master and static slave both at the top level of the design is not supported, because it requires the master to know how to communicate with a slave at potentially two different addresses. A DRC occurs when an RM contains only a pass-through bridge IP. For example:

```
[BD 41-2913] The assignment from address space '/versal_cips_0/
FPD_CCI_NOC_0' at '0x201_0000_0000 [ 64K ]' to slave segment '/axi_gpio_0/
S_AXI/Reg' passes through a BDC with routing bridge IP '/rp1/
axi_gpio_0_smc'. This path is highly discouraged since changing the BDC
variant can cause routing of two different assignments between the same
static address space and static slave, which is not supported by most
routing bridge IPs. Please consider modifying the design to avoid using
BDCs which contain only routing bridge IPs.
```

RECOMMENDED: *Save the BD prior to changing the source BD to ensure any addressing edits are retained.*

Preparing the Design for Implementation

If address offsets or ranges must for a given design source must be modified after they have been associated with a Block Design Container, these edits must be done in the top-level Address Editor. With the top BD open, the Address Editor (and Address Map) show the details for the active design source. If the details of another design source must be modified, set that module as **Active** in the BDC Customization GUI before editing the Address assignments.

Before the design is synthesized and implemented two steps must occur, ensure to execute following steps:

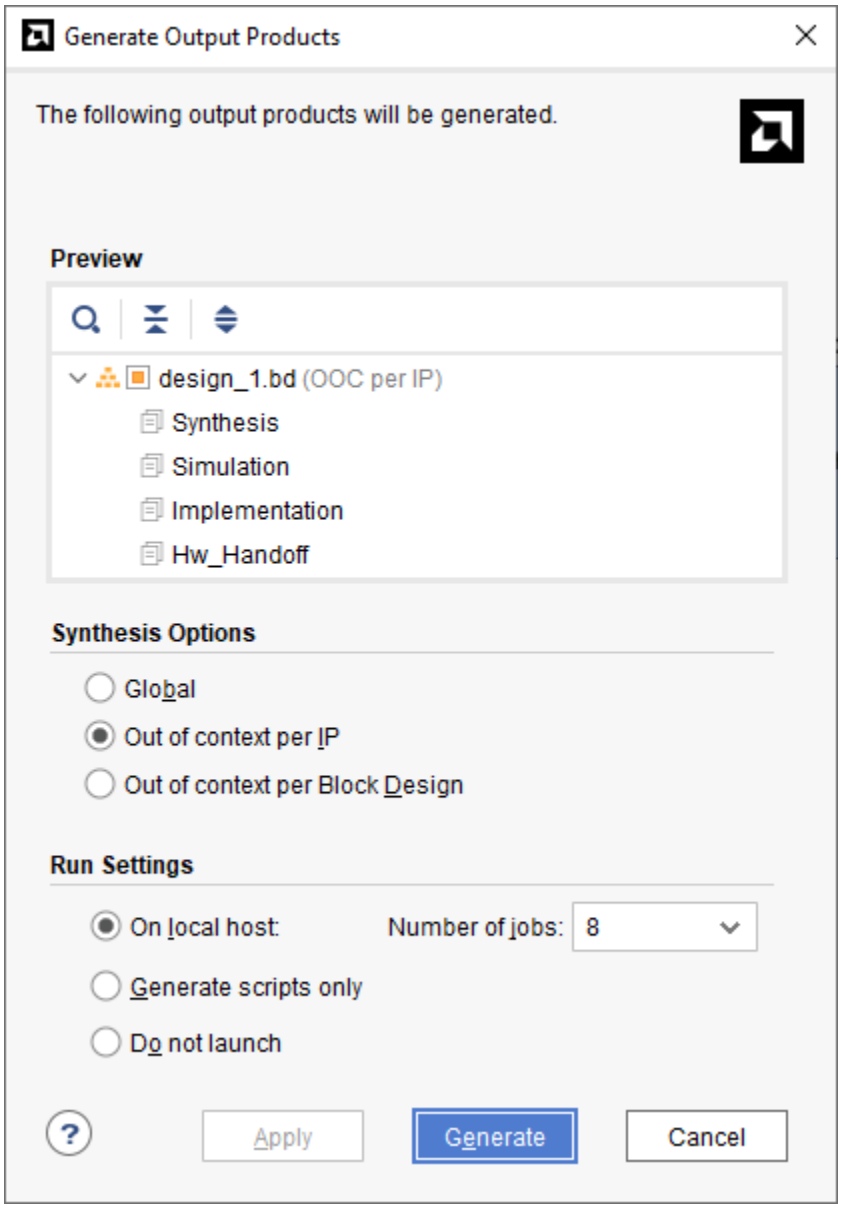
1. Generate a top-level wrapper for the design.
2. Generate the RTL and IP for synthesis.

In the Sources window, right-click on the top level BD, and select **Create HDL Wrapper**. You have the option to create and modify your own top-level RTL code or to let the Vivado tools create and automatically manage this level. Make a selection, and click **OK**.

In the sources, `<design>_wrapper.v` has been created (if the Vivado manage option is selected) and added to the project. This HDL file instantiates the top-level block diagram.

In the Flow Navigator, click the **Generate Block Design** command under the IP INTEGRATOR header. In the resulting dialog box, Out of context per IP or Out of context per Block Design, then click **Generate**. Global will revert to Out of context per Block Design to confirm to DFX rules.

Figure 49: **Generate Output Products for design_1.bd**



Scoping Constraints to the Module Reference Block in the IP Integrator

You can scope constraints to module reference block by adding the XDC file to the CONSTRSET of the OOC run as follows:

1. When the design is ready to synthesize, generate the block design output products.

   The OOC runs appear in the Design Runs tab.

2. Add the intended file to the CONSTRSET of the OOC synthesis of the IP. For example:

```
add_files -fileset [get_property CONSTRSET [get_runs
PL_inst_0_snet_pl_top_wrapper_inst_0_synth_1]]
./snet_xilinx_example_prj/snet.srcs/constrs_1/imports/cdc/
cdc_level_async_in.scoped.xdc
```

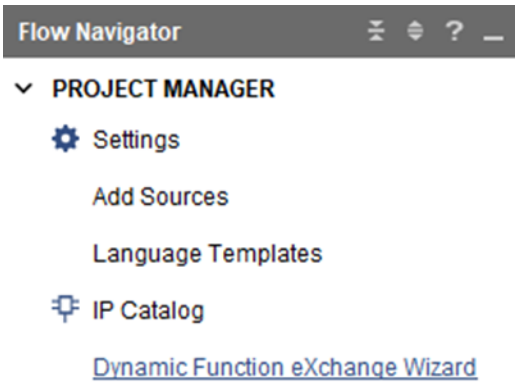
3. Launch **Synthesis**.

Running the Dynamic Function eXchange Wizard

The DFX project flow relies on collections of inter-dependent passes through place and route. Design Configurations and Configuration Runs are defined and managed within the DFX Wizard.

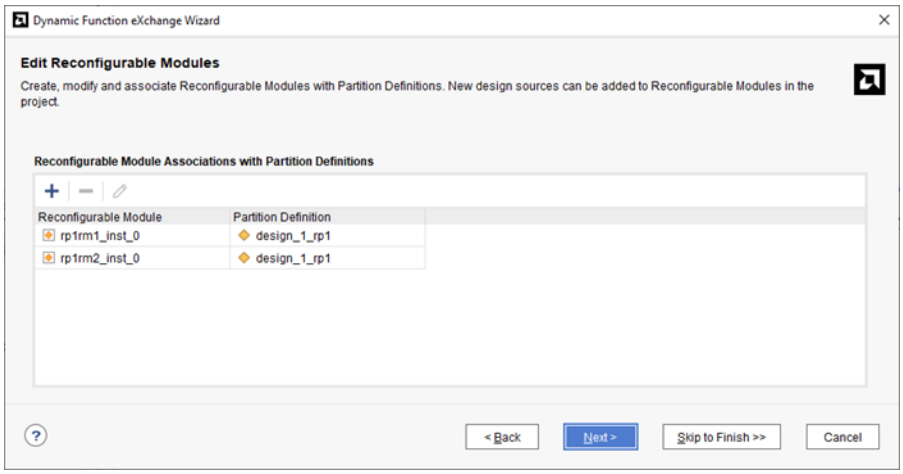
Click on **Dynamic Function eXchange Wizard** in the Flow Navigator, or by selecting **Tools→ Dynamic Function eXchange Wizard**.

Figure 50: **DFX Wizard in the Flow Navigator**



The first screen lists the set of Reconfigurable Partitions (block design containers set to DFX), and all RMs available for each RP.

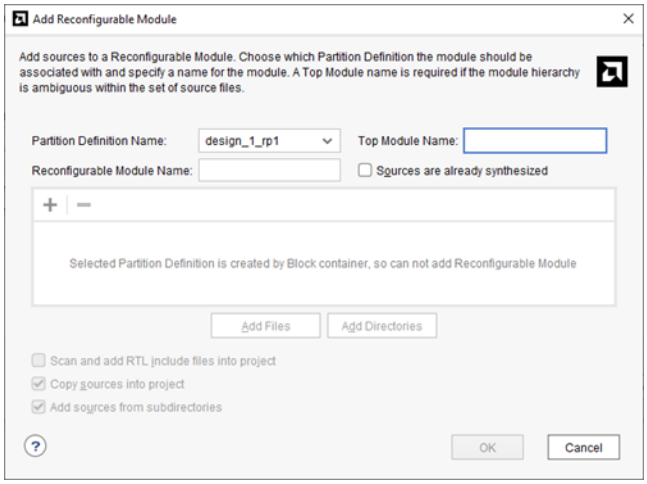
Figure 51: **List of Reconfigurable Modules within the DFX Wizard**



WARNING! *Do not attempt to add new Reconfigurable Modules within the DFX Wizard.*

Unlike with the RTL project flow, the DFX Wizard is NOT a supported entry point for new Reconfigurable Modules. All new RMs must be introduced directly via the block design container. If you click the **+** icon to create a new RM in the DFX Wizard, the Add Reconfigurable Module dialog box does not permit you to create a new RM.

Figure 52: **Add Reconfigurable Module**



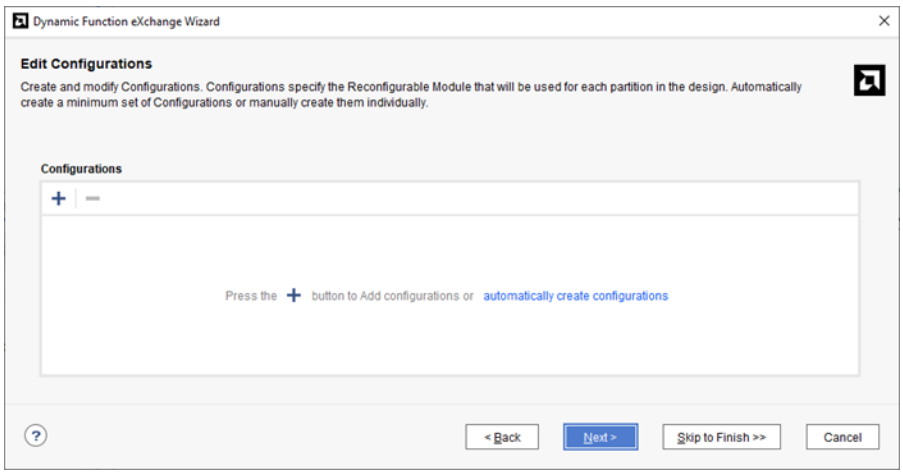
Editing Configurations

With a set of RMs defined within block designs, Configurations can be declared. Each Configuration is a combination of the static logic plus one RM per RP; each Configuration is a full design image.

While each Configuration can be created manually, the simplest path is to let Vivado create the minimum set of Configurations automatically. This is done by selecting the **automatically create configurations** link in the middle of this screen. This will create as many Configurations as necessary to ensure that all RMs are included at least once.

TIP: *This option is only available if no Configurations have been defined yet.*

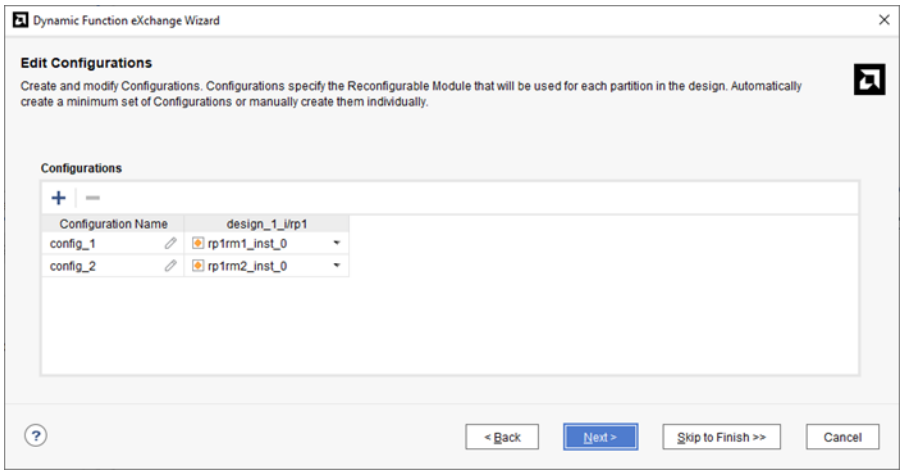
Figure 53: **Edit Configurations before Creating Any Configurations**



If one BDC has more RMs than another, a greybox RM will automatically be used for any RP that has all its RMs covered by prior Configurations. These default Configurations can be modified or renamed, and additional Configurations can be created if desired.

TIP: *Greybox modules are different than black box modules, because they are not truly empty. Greybox RM shave tie-off LUTs inserted to complete legal design connectivity in the absence of an RM and they ensure outputs do not float during operation. To complete legal connectivity, registers can be inserted at clock input ports and clock buffers can be inserted at output ports to drive clock loads. The Vivado tools create these by calling* `update_design -buffer_ports` *on selected modules. Greybox RMs cannot be used in parent runs, because they can lead to sub-optimal results for the static design implementation and/or issues with NoC configuration.*

Figure 54: **Edit Configurations After Automatic Generation of Configurations**



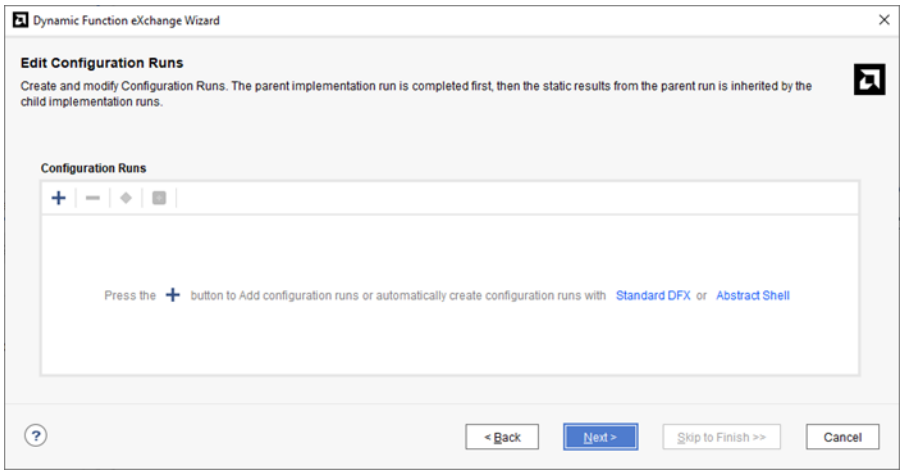
***Note*:** If one RM is used in more than one configuration, the implementation results might be different, because place and route is performed each time, but only if the RM was initially implemented in a child run. RM implementation results are reused if they were originally done in the parent configuration. The `lock_design -level routing` command is set on this reused, implemented RM to lock placement and routing for all subsequent usage in child runs. This allows the Vivado project to track dependencies between parent and child.

Editing Configuration Runs

With all the configurations defined, move to the final screen to manage the configuration runs associated with them. Similar to the configurations themselves, the Vivado tools can automatically create a set of configuration runs. The first configuration in the list is defined as the parent, and all remaining configurations are set as children to that parent.

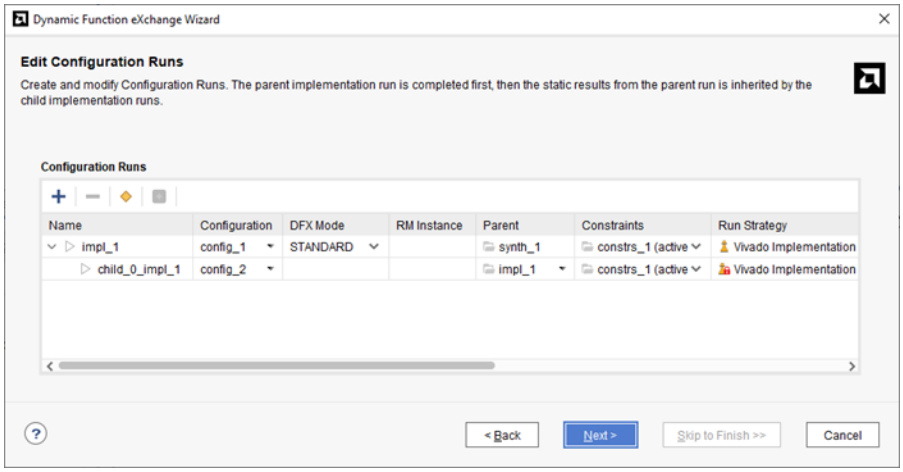
The Vivado tools support Abstract Shell within project modes for UltraScale+ and Versal architectures. This capability is described in detail later in this chapter in Abstract Shell Project Flow. The first time you use the Edit Configuration Runs screen, select **Standard DFX** or **Abstract Shell**.

Figure 55: **Edit Configuration Runs Before Creating Configuration Runs**



The rest of this section shows the flow when Standard DFX is selected.

Figure 56: **Automatically Generated Configuration Runs**



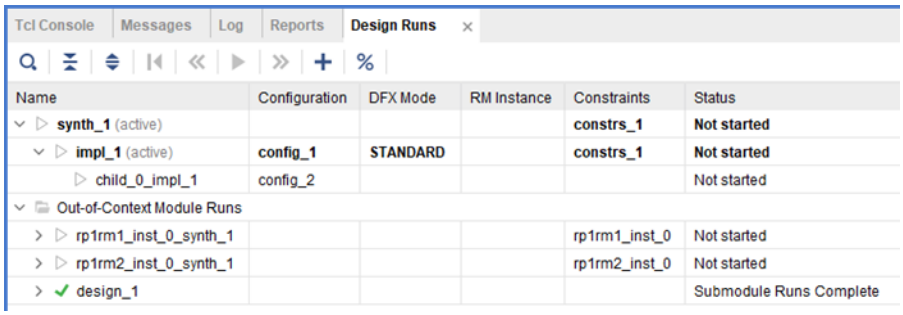
This structure assumes that the first configuration is the most critical or challenging. You are free to change the parent-child relationship by setting that value in the Parent column. A parent of a synthesis run (synth_1 in this example) indicates the configuration (most notably the static part) will be implemented from the synthesized netlist, and a parent of an implementation run (impl_1 in this example) indicates the parent's locked static implementation result will be used as the starting point.

As you explore place and route options, timing closure techniques, and otherwise elaborate on the DFX design, multiple independent parent runs can be used for exploration. Multiple parent runs can be launched in parallel, then child runs can be launched after parent runs complete. The Vivado tools project management handles all the DFX-specific details for creating and storing intermediate checkpoints, including a static-only checkpoint for a routed parent run. Ultimately, a single parent run must be selected to establish a golden static implementation result on which all configurations will be based.

IMPORTANT! *To ensure a safe working environment in silicon, a locked static image must remain consistent across all configurations so bitstream generation creates compatible full and partial bitstreams. This is managed in the DFX Project Flow by establishing a parent-child relationship for related configurations. After the initial configuration run in which the complete static design is locked, the static design must not be modified in subsequent configuration runs. If the static design is not identical across configurations, failure during* `pr_verify` *might occur.*

Add new configuration runs by selecting the + icon. When all configuration runs have been created, click **Next**. On the final screen, the number of new elements are listed. Clicking **Finish** actually perform all the requested changes in the project.

Figure 57: **Synthesis and Implementation Design Runs Ready to be Launched**



In the Design Runs window, out-of-context synthesis runs are created for each RM, and all Configuration Runs are generated. Relationships between parent and child runs are shown by the levels of indentation.

The Dynamic Function eXchange Wizard is the central mechanism for making any changes to Configurations or Configuration Runs in the IP integrator flow. This includes creating new Configurations or Runs, modifying the relationships between runs, or removing any of the above. When working within the wizard, nothing is saved or executed until you click the **Finish** button, so you can move forward or back through the screens, making adjustments as needed.

Using Export Hardware

Export hardware supports both fixed and extensible platforms. Fixed platforms contain the complete set of hardware files for the device and are exported to the Vitis tools to add software to the design. Extensible platforms allow the Vitis tools to add hardware accelerators to the design. For more information on fixed and extensible platforms, see the *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393).

You can export the design using the `write_hw_platform` Tcl command or using the **File → Export → Export Hardware** menu command in the Vivado IDE. Using this command from the Vivado IDE defaults to the active parent configuration and generates the XSA type defined in the project settings. To generate the XSA for a child run, open that implementation run first before exporting hardware via `write_hw_platform`.

For fixed hardware platforms, there are several command-line switches, currently available in Tcl only, to configure `write_hw_platform`. The following table shows the contents of the Xilinx support archive (XSA) file for specific command line arguments related to the programmable device image (PDI) file and hardware hand off (HWH). These options are available for Versal devices only.

Table 11: **Contents of a Fixed XSA File for Specific `write_hw_platform` Arguments**

Command Line Arguments	Exported PDI	Exported HWH
Default (only option available in the GUI via Export Hardware)	Complete PDI for active configuration	Complete HWH for active configuration
`-static`	Complete PDI for active configuration	Reconfigurable partitions are black boxed
`-rp -rm`	Partial PDI for the specified RM in the specified partition	HWH for the specified RM in the specified partition

After export, the Vitis tools use the XSA file to complete the software platform.

Extensible platforms require the following additional properties to be set so they can be passed to the Vitis tools:

- `set_property platform.platform_state "impl" [current_project]`
- `set_property platform.uses_pr true [current_project]`
- `set_property platform.dr_inst_path { <dynamic region impl hierarchy path> } [current_project]`

***Note*:** PFM_NAME must be set on the DFX block design.

A call to `write_hw_platform` without the `-fixed` option creates an extensible platform. For more information on extensible dynamic platforms, see the Versal Adaptive SoC Custom DFX Platform Creation Tutorial available from the GitHub repository.

Supported/Unsupported Features

Supported Features

The BDC flow supports all architecture for DFX.

Unsupported Features

The following features are currently not implemented:

- A BDC cannot contain another block design container. Only a single level is currently supported.
- Nested DFX for any architecture in IP integrator or RTL project mode.

Known Issues and Limitations

When using the DFX Decoupler in IP integrator and when the interface is created and the VLNV is subsequently changed, the VLNV property of the port pins do not change. Consequently, the IP integrator does not allow connections to the newly changed ports, because it still appear as the old VLNV type. As a work around, execute the following procedure:

1. Change the VLNV
2. Save and Close the BD
3. Reopen the BD to verify the changed VLNV

Abstract Shell Project Flow

The Abstract Shell design flow is nearly identical to the standard project-based Dynamic Function eXchange design flow. Support is in place for both RTL-based projects that use Partition Definitions to declare Reconfigurable Partitions, as well as the IP integrator solution that leverages Block Design Containers. While the declaration of RPs is done in different ways in these two environments, the synthesis and place and route management is still the same, so the support for Abstract Shells is identical.

The differences between the standard DFX and Abstract Shell flows are limited to the steps where the static design checkpoint is written from the initial (parent) implementation, and where the static design is opened to begin implementation of the second RM (and beyond) for each RP. Changes are seen within the DFX Wizard and the subsequent Design Runs for child instances.

Abstract Shells have two fundamental advantages over standard full-static checkpoints:

1. Compile time for new Reconfigurable Modules is reduced for child runs, as Vivado implementation tools do not need to load or consider much of the information contained in the static part of the design.
2. Static design information, including licensed IP, is hidden from view in an Abstract Shell, enhancing design security and reducing IP license requirements.

Project mode for Abstract Shells leverages the first benefit but not the second. The entire design is always resident in a DFX project so there is no mechanism to hide any details about the static part of the design.

Abstract Shell Creation and Usage

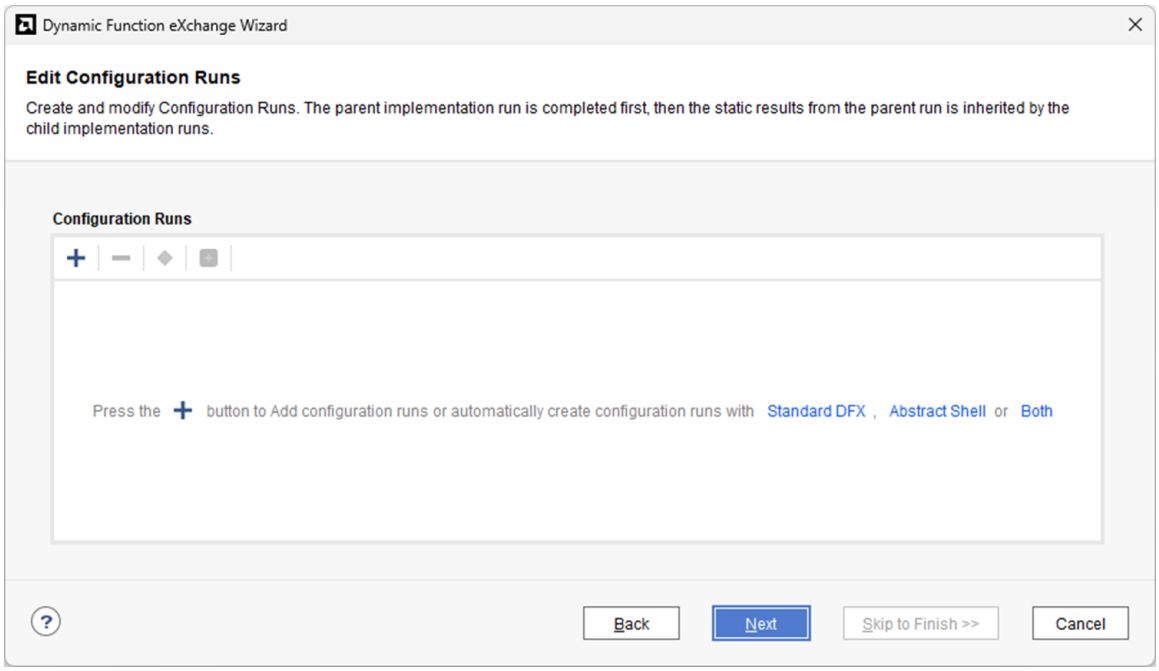
After defining Reconfigurable Partitions (RPs), building the design sources, and generating output products (IP integrator flow) for your DFX design, open the DFX Wizard. The first few pages through the wizard are the same as the standard DFX flow described earlier in this chapter. Manage Reconfigurable Modules and configurations in the same way regardless of Abstract Shell. The capabilities unique to Abstract Shell start on the Configuration Runs page.

Automatically Create Configuration Runs

When you first use the wizard, or if you clear all configuration runs, there are three options for automation:

- Standard DFX
- Abstract Shell
- Both

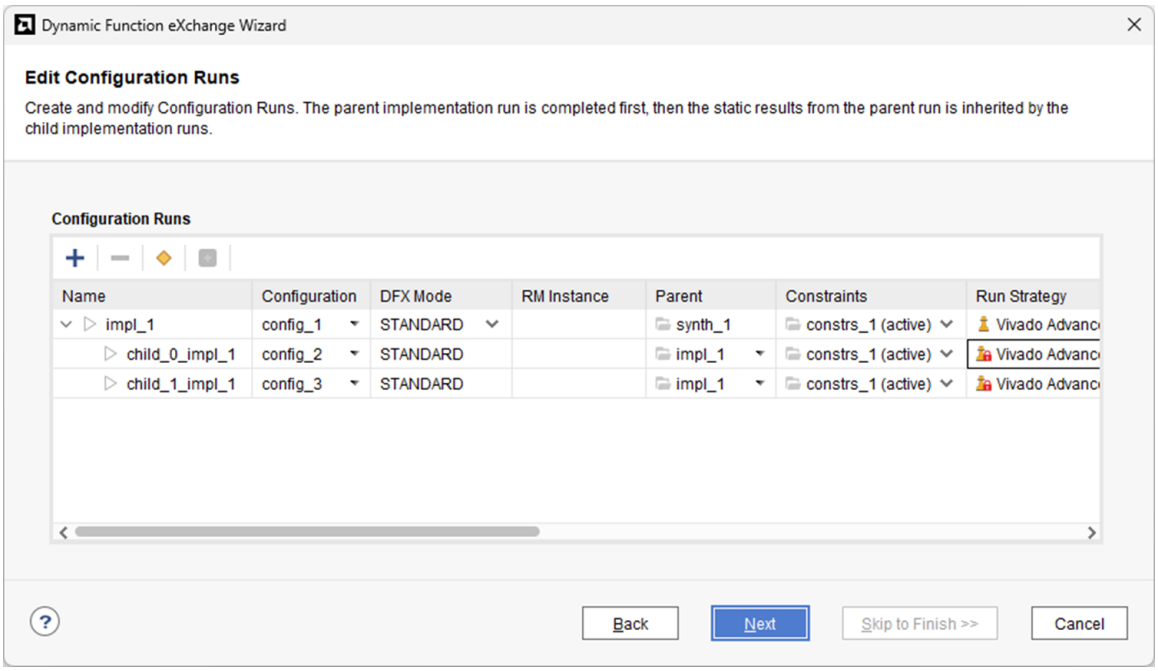
Figure 58: **Edit Configuration Runs**



Standard DFX

Clicking **Standard DFX** uses the basic DFX project flow that uses a full static design image for each configuration run. Config_1 is the parent run, and all remaining configurations are child runs dependent on that parent run. In the following figure, config_2 and config_3 use the full locked static design checkpoint from config_1 as the starting point for implementing new Reconfigurable Modules.

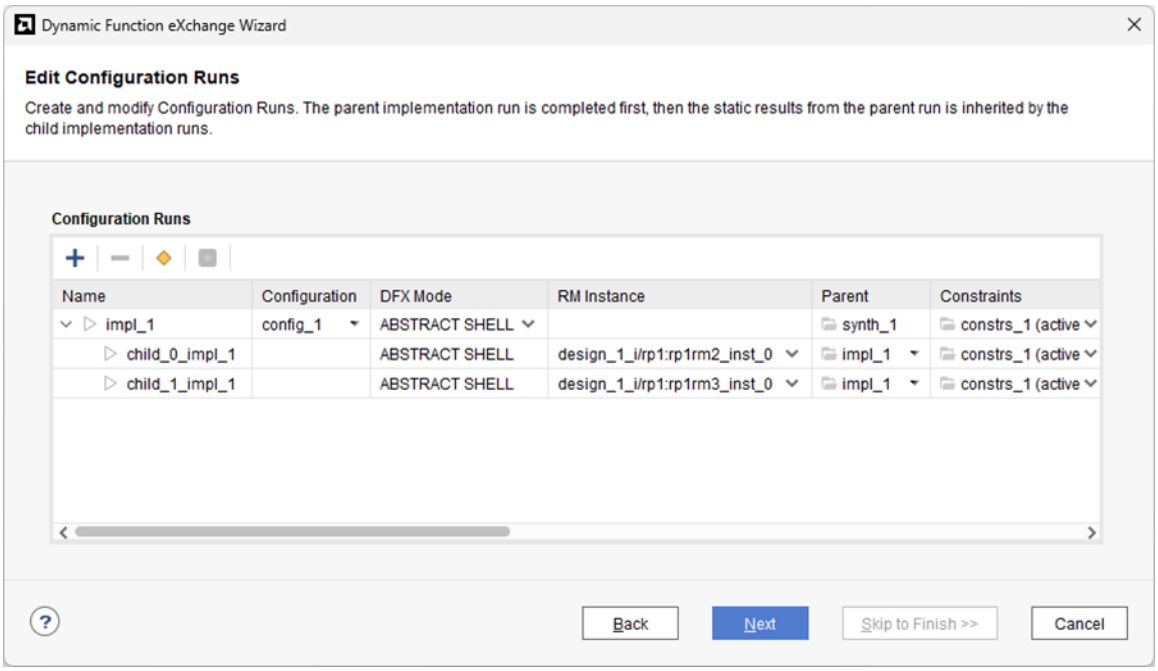
Figure 59: **Standard DFX Configuration Runs**



Abstract Shell

Selecting **Abstract Shell** creates a similar looking result, and the parent configuration is the same. However, each of the child runs uses an Abstract Shell for a specific target Reconfigurable Partition.

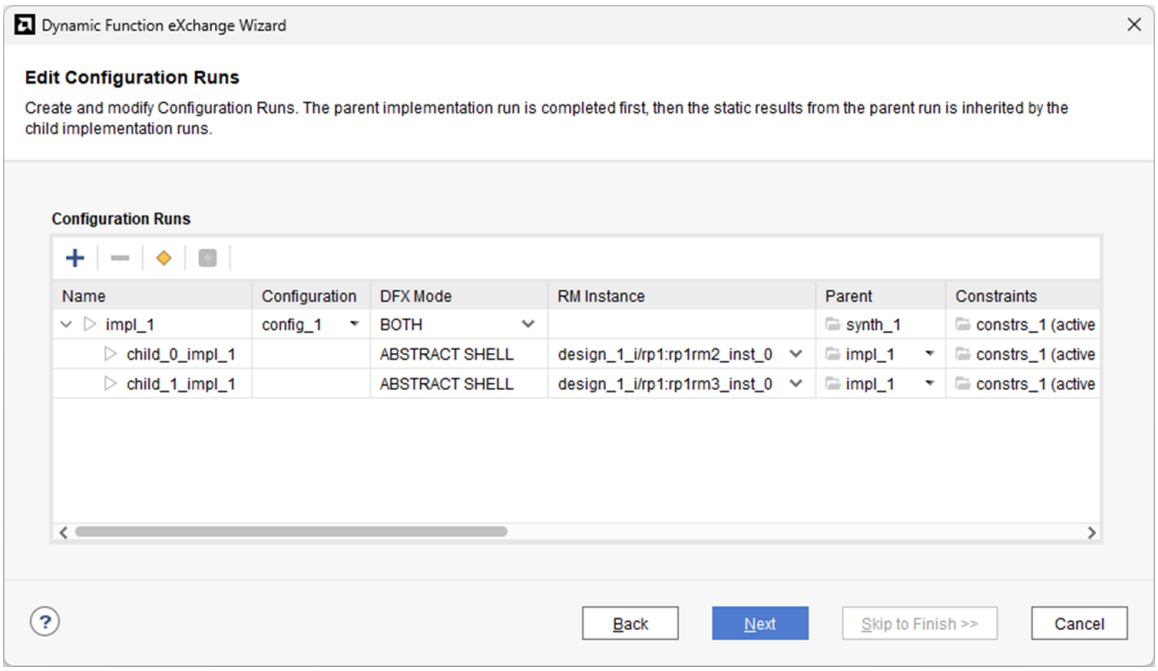
Figure 60: **Abstract Shell Configuration Runs**



Both

Selecting **Both** allows either approach to be used for child runs. Both shell types are generated at the end of the parent implementation run so either can be selected for child runs under that parent. By default, the Abstract Shell approach is used.

Figure 61: **Both Types of Configuration Runs**



As you can see in these images, the option that tracks the difference between these approaches is the DFX Mode option. The choice made here determines the design checkpoints written out at the end of the parent configuration run. The parent run always creates a full design image containing the static and dynamic parts of the design (one RM per RP). The difference is which locked static shells are created for child runs.

- **DFX Mode = STANDARD:** A single checkpoint containing the entire static design, with a black box for each Reconfigurable Partition and all placement and routing locked, is created.
- **DFX Mode = ABSTRACT SHELL:** One or more checkpoints each containing the abstract shell for one Reconfigurable Partition are created. Abstract Shells also have a black box for the RP and locked placement and routing for the static design, but the amount of static information is stripped down to a bare minimum to provide context for implementation of a new RM.
- **DFX Mode = BOTH:** Both flows are available for child runs. Users can select the full static design to implement new Reconfigurable Modules for all Reconfigurable Partitions in the design, or Abstract Shells can be used to implement RMs for a specific RP.

The other option that is used only in the Abstract Shell flow is the RM Instance option. This declares which Reconfigurable Partition is the target for a particular child run, along with which Reconfigurable Module is to be implemented in that RP.

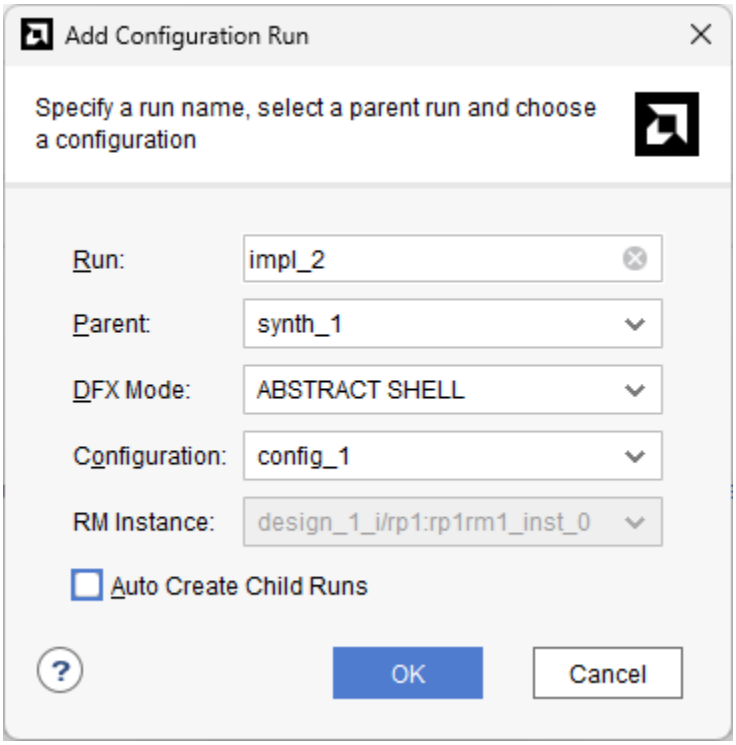
Note: Any configuration containing one or more greybox RMs are added as a child run when Standard DFX is selected for automatic creation of configuration runs, because these configurations were explicitly declared by the user. However, greybox RMs are not added as Abstract Shell child runs when that mode is selected for automatic creation. These runs can be manually created if desired.

Manually Create Configuration Runs

If you do not use the Abstract Shell or Both option for automatic configuration run creation, you can set this directly for a new set of configuration runs by doing the following:

1. in the Edit Configuration Runs window, click the **+** icon to create a new parent configuration run that starts with a synthesized design.
2. In the resulting dialog box, set the DFX Mode to ABSTRACT SHELL or BOTH.

Figure 62: **Create a New Parent Run**



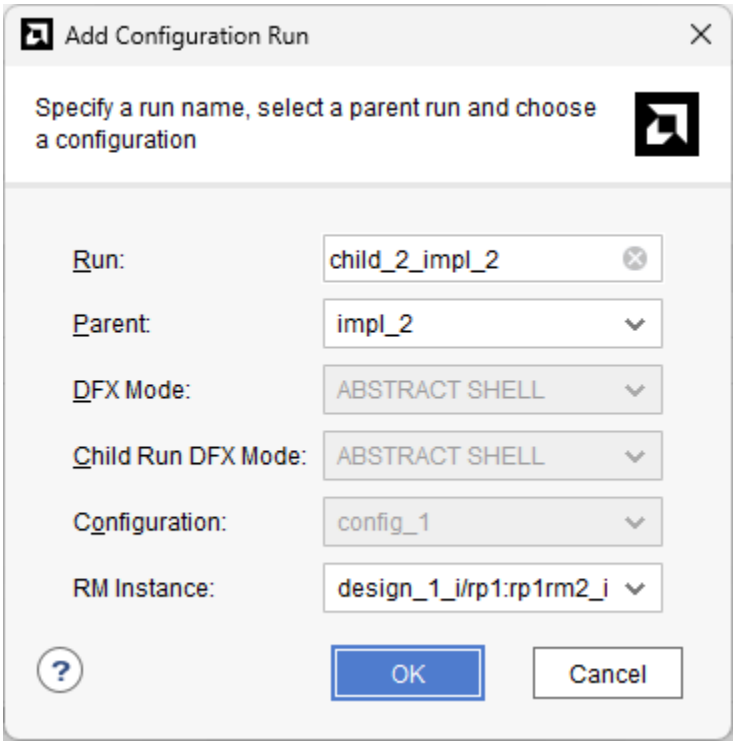
***Note*:** The impl_1 run name is automatically generated behind the scenes and cannot be manually generated in this dialog.

The Auto Create Child Runs option is enabled by default. This option creates as many child implementation runs as necessary to implement all existing RMs (except greybox RMs) not included in the parent configuration.

Create these child runs manually by deselecting the **Auto Create** option and clicking **OK**.

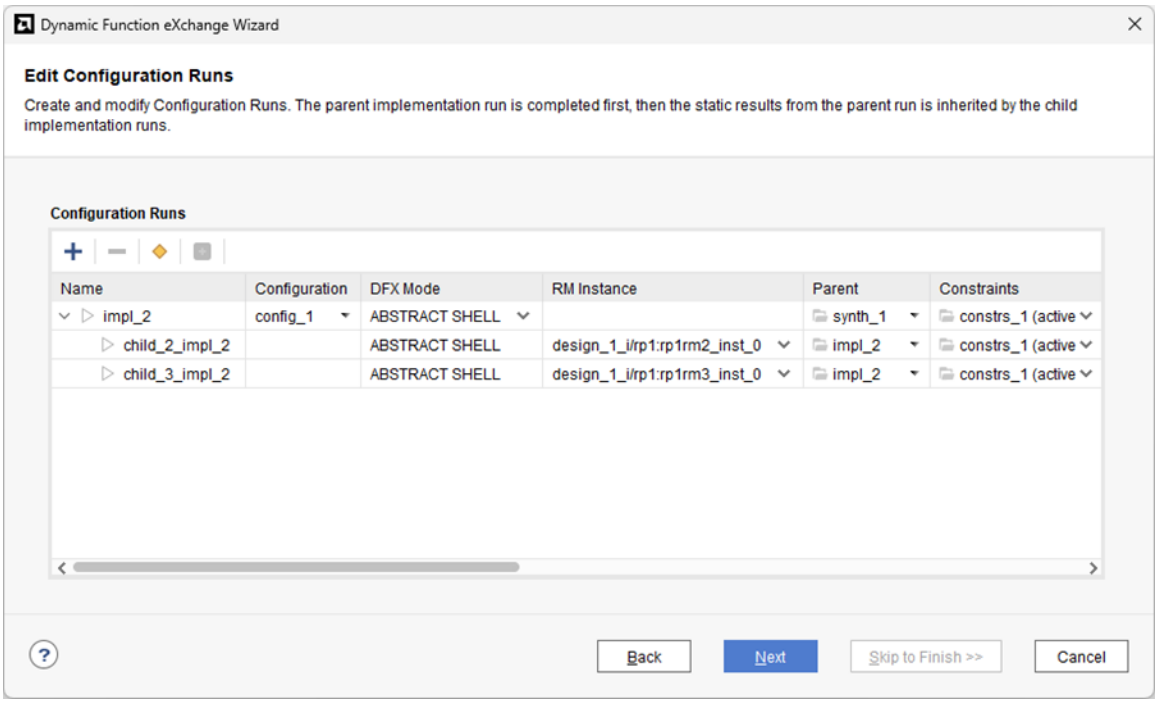
Click the + again to create another run. Set the parent to the recently created run. If you set the Parent to the Abstract Shell mode parent implementation, the DFX Mode and Configuration fields are ignored, as this child run must follow the structure of the parent it relates to.
Select the new RM Instance to implement within this Abstract Shell. Thiis instance references both the Reconfigurable Partition and the Reconfigurable Module. For designs with multiple Reconfigurable Partitions, many more child runs are necessary compared to a Standard DFX flow, as every RM is implemented independently.

Figure 63: **Create a Child Run for the Existing Parent Run**



Back in the Configuration Runs page, you can review and edit the RM Instances implemented in any of the runs, and can set any constraints or strategies to be used.

Figure 64: **Set the RM Instance for the Child Run**

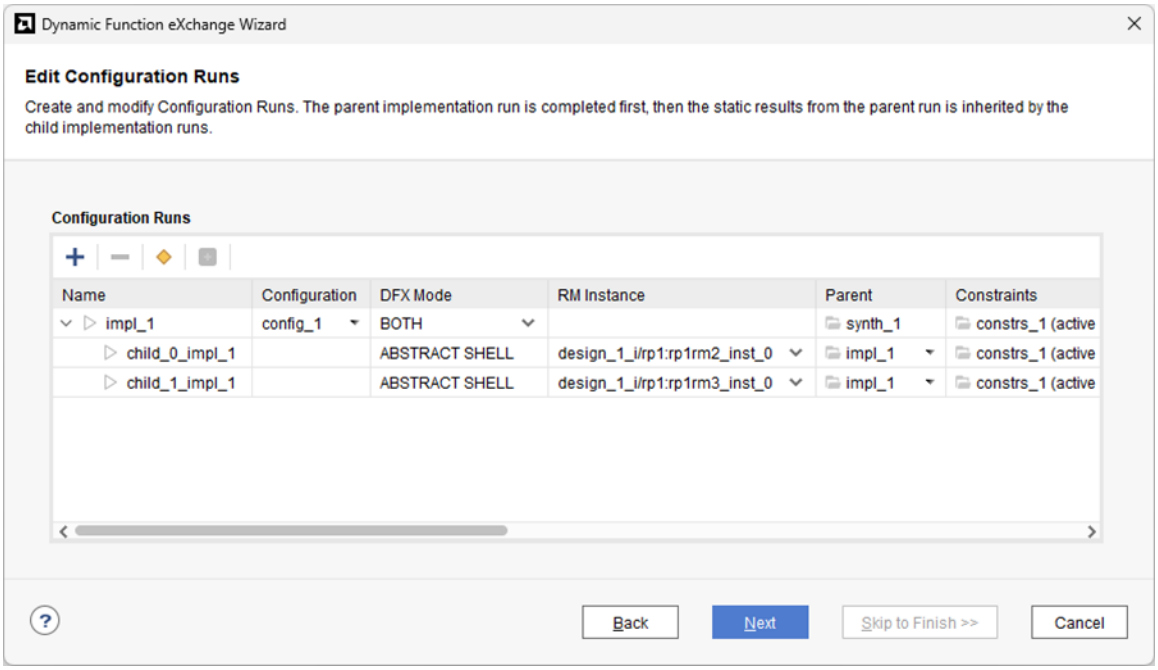


When implementing the design, the parent run is compiled first, then all child runs can be launched in parallel. It is expected that in nearly every case, Abstract Shell runs are faster than an equivalent full configuration child run. With designs containing more than one Reconfigurable Partition, compiling each Reconfigurable Module in parallel further reduces overall compile time.

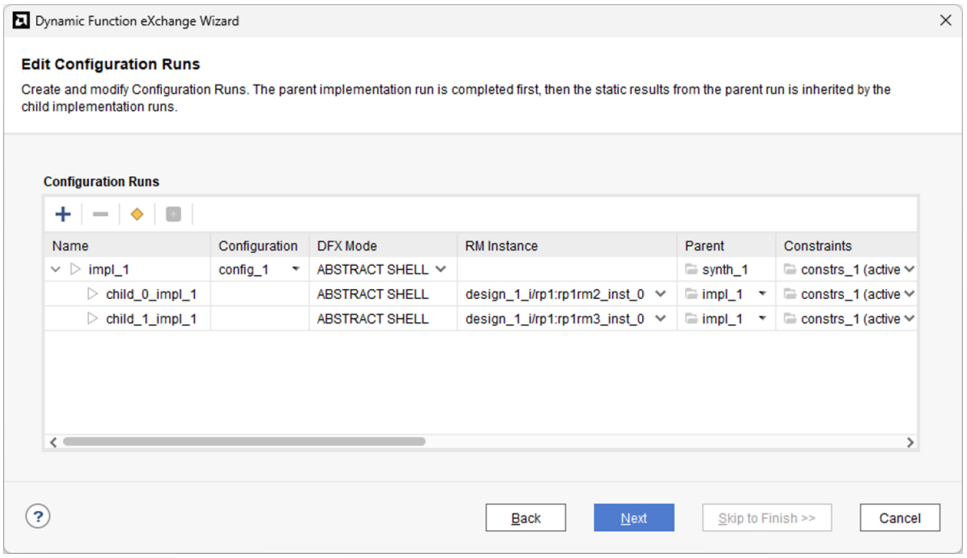
Using DFX MODE = BOTH

If DFX Mode is set to Both, then child runs of either type can be used. This approach takes slightly longer to compile the parent configuration run as both types of shells must be generated, but then you have full flexibility for the types of configuration runs that follow. For example, if you declare a greybox module for each RP, you can process the entire design using the Abstract Shell process, but then create a greybox child run to use as the initial boot of the device.

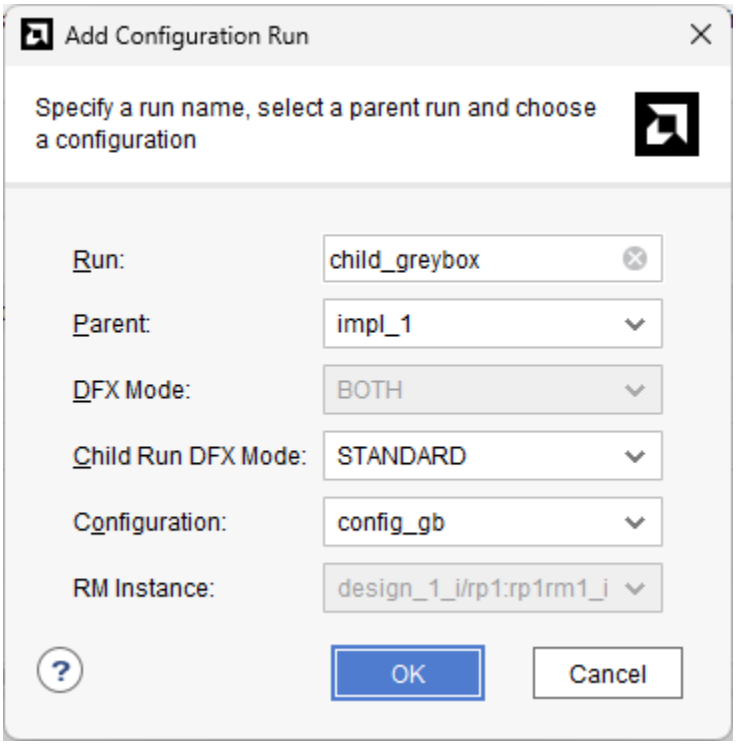
1. Create a greybox configuration by adding a new configuration and setting each RP to the predefined greybox setting.



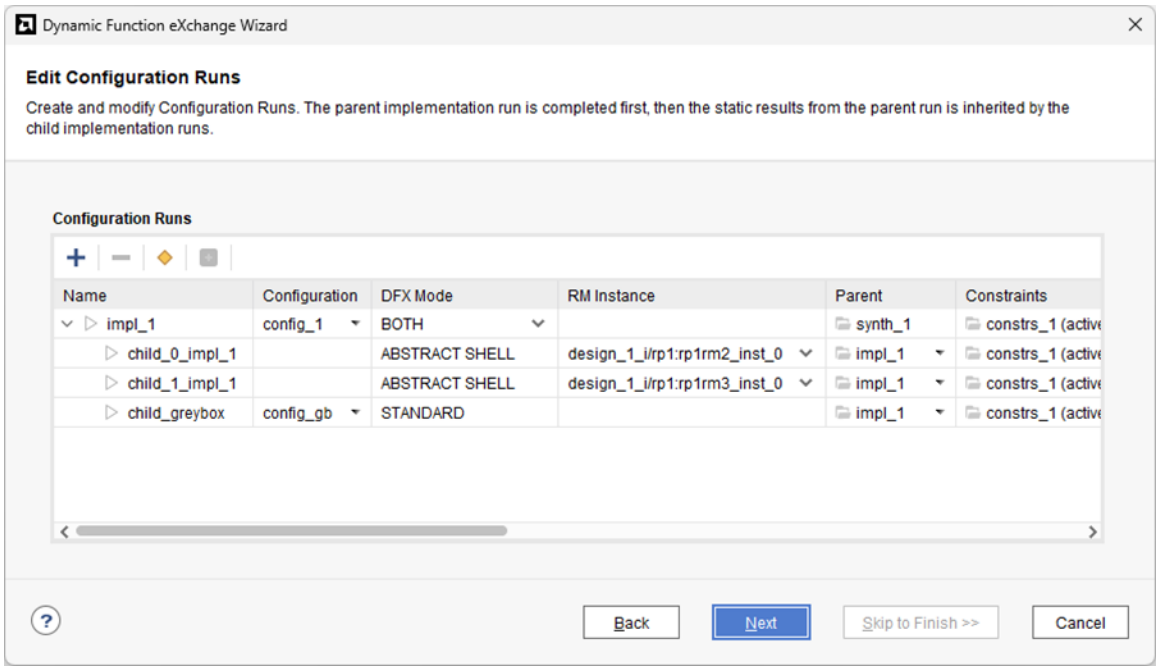
2. Automatically generate the configuration runs using the Both option as described above. The greybox configuration is ignored.



3. Click + to create a final configuration run. When the parent is set to the impl_1 run in this case, the child run DFX mode option can be set to either STANDARD or ABSTRACT SHELL. The choice for this option determines which of the final two options are available.



The final set of Configuration Runs is a mix of Standard DFX and Abstract Shell runs. All child runs can be launched in parallel.



Device Image Creation

Bitstream generation is different for the two modes. At the end of a Standard DFX configuration run (parent and each child), a call to generate bitstream or generate device image produces multiple bitstreams: a full device `.bit` or `.pdi` file containing static and reconfigurable logic in that configuration, as well as one partial .bit or .pdi file for each reconfigurable module present in that run.

At the end of the parent run for an abstract shell configuration, this is also true, as the full static design is present so both full and partial `.bit` or `.pdi` files can be generated. However, at the end of an abstract shell child run, only a single partial `.bit` or `.pdi` file for the target reconfigurable module implemented in that run might be created. A full device image cannot be generated for an abstract shell child run simply because the entire static design is not present in that checkpoint.

If you require a full device bitstream that contains reconfigurable modules from child abstract shell runs together with the full static design, an in-memory design must be assembled by using `add_files` and `link_design` (or `open_checkpoint` followed by `read_checkpoint -cell`). These commands should be used to pull together routed design checkpoints containing the static (from the parent run) and reconfigurable (from child runs) results, then a call to `write_bistream -no_partial_bitfile` (UltraScale+) or `write_device_image -no_partial_pdifile` (Versal) can be used to generate a full device programming image.

Just as for the Standard DFX mode, `pr_verify` is automatically called for any child run to validate the implemented result compared to the locked static image it was built from – in this case it is the abstract shell checkpoint starting point for child runs. Device image creation does not run if this check fails.

Supported/Unsupported Features

This section shows the current lists of supported and unsupported features in this version of Vivado.

Supported Features

- Abstract Shell creation and implementation for all UltraScale+ and Versal devices that support DFX
- Abstract Shell flow support for both IP integrator block design and RTL design flows

Unsupported Features

- Abstract Shell does not and will never support UltraScale or 7 series devices

- The current solution is set up for single-project environments. There is no support for exporting an Abstract Shell checkpoint to spawn a new “child project” for users to compile only Reconfigurable Modules. To share an Abstract Shell as a starting point for a secondary user, the flow must switch to a non-project Tcl scripted approach.

Known Limitations and Restrictions

Once a child run has been implemented, do not return to the DFX Wizard and change the DFX Mode of the parent. The child run structure will no longer match the parent style. To change modes, create a new parent run with new child runs.

Chapter 5

Design Considerations and Guidelines for All AMD Devices

This chapter explains design requirements that are unique to Dynamic Function eXchange (DFX), and covers specific DFX features within the AMD design software tools.

To take advantage of the dynamic reconfiguration capability of AMD FPGAs, you must analyze the design specification thoroughly, and consider the requirements, characteristics, and limitations associated with DFX designs. This simplifies both the design and debug processes, and avoids potential future risks of malfunction in the design.

This chapter describes the design requirements that apply to all AMD devices. For design requirements specific to the individual FPGA and SoC architectures, see the following chapters in this manual:

- Chapter 6: Design Considerations and Guidelines for 7 Series and Zynq Devices
- Chapter 7: Design Considerations and Guidelines for UltraScale and UltraScale+ Devices
- Chapter 8: Design Considerations and Guidelines for Versal Devices

Dynamic Function eXchange IP

AMD has created four pieces of intellectual property specifically for use within DFX designs. There is no charge for any of these IPs, and DFX designs do not require them. They are available to assist you to quickly and easily implement key aspects of a reconfigurable design. The IPs are all found under the DFX heading within the IP catalog, and each has its own landing page on Xilinx.com with a detailed product guide.

These four IPs for DFX are available in AMD Vivado™ and can be used for any AMD device that supports DFX. These IPs now incorporate the DFX terminology in their names and throughout the IP, while remaining functionally equivalent to their Partial Reconfiguration-named predecessors. You should use the IP upgrade feature to transition any existing PR IP to DFX IP. For more information, refer to the product guides for each IP.

- **DFX Controller:** The DFX Controller core provides management functions for self-controlling partially reconfigurable designs. It is intended for enclosed systems where all of the reconfigurable modules (RM) are known to the controller. The optional AXI4-Lite register interface allows the core to be reconfigured at runtime, so it can also be used in systems where the RMs can change in the field. The core can be customized for many Virtual Sockets, RMs per Virtual Socket, operations and interfaces. Labs 5, 6, and 7 in the *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947) gives examples of the DFX Controller IP in a sample design.

  ***Note*:** This IP is not applicable for AMD Versal™ devices.

- **DFX Decoupler:** The DFX Decoupler can be used to provide a safe and managed boundary between the static logic and an RP during reconfiguration. The core can be customized for the number of interfaces, type of interfaces, decoupling functionality, status and control.
- **DFX AXI Shutdown Manager:** One or more DFX AXI Shutdown Managers can be used to make the AXI interfaces between a RP and the static logic safe during reconfiguration. When active, AXI transactions sent to the RM, and AXI transactions emanating from the RM, are terminated because the RM might not be able to complete them. Failure to complete could cause system deadlock. When inactive, transactions pass unaltered.
- **DFX Bitstream Monitor:** The DFX Bitstream Monitor can be used to identify partial bitstreams as they flow through the design. This information can be used for debugging or system applications such as blocking bitstream loads. Identifiers embedded at key places in partial bitstreams are extracted and reported by the core. This information can be passed to Vivado HW Debugger using an ILA core to work out what partial bitstream was fetched, if it was fetched in its entirety, and how far through the datapath it went.

Design Hierarchy

Good hierarchical design practices resolve many complexities and difficulties when implementing a partially reconfigurable FPGA design. A clear design instance hierarchy simplifies physical and timing constraints. Registering signals at the boundary between static and reconfigurable logic eases timing closure. Grouping logic that is packed together in the same hierarchical level is necessary.

These are all well known design practices that are often not followed in general FPGA designs. Following these design rules is not strictly required in a partially reconfigurable design, but the potential negative effects of not following them are more pronounced. The benefits of Dynamic Function eXchange are great, but the extra complexity in design could be more challenging to debug, especially in hardware.

For additional information about design hierarchy, see Hierarchical Design Flows.

Related Information

Hierarchical Design Flows

Dynamic Reconfiguration Using the DRP

Logic that is in the static region, and therefore is never partially reconfigured, can still be reconfigured dynamically through the Dynamic Reconfiguration Port (DRP). The DRP can be used to configure logic elements such as MMCMs, PLLs, and serial transceivers (MGTs).

Information about the DRP and dynamic reconfiguration, including how to use the DRP for specific design resources, can be found in these documents:

- *7 Series FPGAs Configuration User Guide* (UG470)
- *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476)
- *7 Series FPGAs GTP Transceivers User Guide* (UG482)
- *MMCM and PLL Dynamic Reconfiguration Application Note* (XAPP888)
- *UltraScale Architecture Configuration User Guide* (UG570)
- *UltraScale Architecture Clocking Resources User Guide* (UG572)
- *UltraScale Architecture GTH Transceivers User Guide* (UG576)
- *UltraScale Architecture GTY Transceivers User Guide* (UG578)

Packing Logic

Any logic that must be packed together must be placed in the same group, whether it is static or reconfigurable. For example, if a LUT and a flip-flop are expected to be placed within the same slice, they must be within the same partition. Partition boundaries are barriers to optimization.

For RPs that include I/O, Clocking, and GT resources, it might be necessary to instantiate any I/O buffers that are automatically inferred by the tools inside that RP level. For example, if `GT_COMMON` is in an RP, the `IBUFDS_GTE` needs to be instantiated. If the associated I/O buffer is in the top-level/static portion, it cannot be packed.

Design Instance Hierarchy

The most simple method is to instantiate the RPs in the top-level module, but this is not required because a RP may be located in any level of hierarchy. Each RP must correspond to exactly one instance—an RP must not have more than one top. The instantiation has multiple modules with which it is associated.

Changes in design hierarchy can be used to merge and/or separate modules and leaf cells into and out of an RP level of hierarchy. There are several reasons to do this:

- To balance device resources between the dynamic region and static region, making the design more efficient. For example, if the target RP takes up most of the device, and there is a module in the static region that requires a high number of Block RAMs unavailable to Static, you can move that module into the dynamic region.
- If you need cells to reside in the same physical area of the device, but they are in a different design hierarchy. For example, if you need `GT_CHANNELS` to be placed in the same UltraScale Clock Region, but the design has GTs in both the Static and RP regions.
- To ensure that dedicated connections, for example from `IBUFDS_GT` to `GT_COMMON`, reside in the same region.

Reconfigurable Partition Interfaces

One of the fundamental requirements of a partially reconfigurable design is consistency between RMs. As one module is swapped for another, the connections between the static design and the RM must be identical, both logically and physically. To achieve this consistency, optimizations across the partition boundary or of the boundary itself are prohibited.

RECOMMENDED: *Keep interface logic connecting to and from RM ports consistent across all RMs. Do not change the levels of logic between RMs, such as using one level of logic in the initial design and five levels of logic in next RM. Also, do not change the driver type, such as using flip-flops in the initial RM and block RAM in next RM. Because the static side of the interface is locked after the initial configuration, the tools are unable to adjust for these changes in later configurations. AMD recommends registering all inputs and outputs of the RMs.*

For optimal efficiency, all ports of a RP should be actively used on the static design side. For example, if static drivers of the RP are driven by constants (0 or 1), they are implemented through the creation of a LUT instance and local tie-off to a constant driver and cannot be trimmed away. Likewise, unconnected outputs remain on RP outputs, creating unnecessary waste in the overall design. These measures must be taken by the implementation tools to ensure that all RM have the same port map during design assembly.

Examine the interface of all RPs after synthesis to ensure that as few constants or unconnected ports as possible remain. By clearing out dead logic, resource utilization is reduced, and congestion and timing closure challenges easier to address.

Six different cases are possible for partition interface usage:

- **Both Static and Reconfigurable Module sides have active logic:** (Applies to partition inputs or outputs)

  This is the optimal situation. A partition pin is inserted.

  If partition inputs are driven by VCC or GND, push these constants into the RM. This reduces LUT usage and allow the implementation tools to optimize these constants with the RM logic.

- **The Static side has an active driver but the Reconfigurable Module does not have active loads:** (Applies to partition inputs)

This is acceptable because it accommodates the situation in which not every RM has the same I/O requirements. A partition pin is inserted, and the unused input ports are left unconnected.

For example, one module might require CLK_A, while a second might require CLK_B. Clock spines are pre-routed to the RP clock regions, but the module only taps into the clock source that is needed. However, if a partition input is not used by any RM, it should be removed from the partition instantiation.

- **The Static side has active loads but the Reconfigurable Module does not have an active driver:** (Applies to partition outputs)

  This is acceptable and similar to the case above. A partition pin is inserted, and it is driven by ground (logic 0) within the RM.

- **The Static side does not have an active driver, but the Reconfigurable Module has active loads:** (Applies to partition inputs)

  This results in an error that must be resolved by modifying the partition interface. The following is an example of an error that may be seen for this scenario:

  ```
  ERROR: [Opt 31-65] LUT input is undriven either due to a missing
  connection from a design error, or a connection removed during opt_design.
  ```

  This error message would be followed by a LUT instance that is within the RM.

- **Reconfigurable Module has an active driver, but the Static side has no active loads:** (Applies to partition outputs)

  This does not result in an error, but is far from optimal because the RM logic remains. No partition pin is inserted. These partition outputs should be removed.

- **Neither Static nor Reconfigurable Module sides have driver or loads for a partition port:** (Applies to partition inputs or outputs)

  Nothing is inserted or used, so there is no implementation inefficiency, but it is unnecessary in terms of the instantiation port list.

  One basic requirement regarding partition interfaces is that each pin must be unidirectional. The use of bidirectional ports (type inout) on the module boundary is not supported. This requirement is due to the fact that routing resources within AMD FPGAs are fundamentally unidirectional and there are no internal tristate resources that could help manage changes in direction. Even if module ports resolve to be input or output after synthesis, Vivado will still consider the port to potentially be bidirectional. Please change any inout port to be explicitly input or output.

Embedded I/O Usage Guidelines

When using embedded I/O buffers within reconfigurable modules (RMs) in a dynamic function eXchange (DFX) design, it is critical to follow a structured approach to avoid tool insertion errors, DRC violations, and routing issues.

Following are the recommended methodologies to correctly implement embedded IOBs within a reconfigurable partition:

- Avoid I/O buffers (IOB) instantiation and inference at the top-level.
  - Do not instantiate I/O buffers (for example, IBUF, IBUFDS) in static RTL for any XPIOs that are part of an RM. This prevents conflicts where both static and RM try to drive the same I/O, leading to cascaded buffer errors.
  - By default, Vivado infers IOBs at the top-level of the design. Apply the RTL attribute `IO_BUFFER_TYPE = "NONE"` on the top-level port connected to the embedded IOB. This prevents Vivado from automatically inferring an IOB at the top level, avoiding duplication and conflicts.
- Instantiate I/O buffers inside the RM.
  - The RM IP or RTL must instantiate the appropriate I/O buffers (for example, IBUF, IBUFDS) for the embedded I/Os.
  - For block designs, you can add the utility buffer IP to instantiate IOBs within the RM or use HDL file containing instantiated buffers as a module reference.
- Ensure netlist continuity for Unused Embedded IOBs
  - When specific embedded IOBs are unused in a specific RM compile, those I/O pads should be connected all the way to the reconfigurable module hierarchy in the netlist and be unconnected within the specific RM.
  - To guide the I/O Placer, users are expected to keep corresponding PACKAGE_PIN constraints for those ports even in those compiles that do not use those I/O pads.
- Reapply the I/O constraints for every configuration
  - If an RP has embedded I/O, the I/O (PACKAGE_PIN, IOSTANDARD, direction) must be reapplied for every configuration, even if the I/O pins are identical between every RM.
  - In the Vivado database, a port is a top-level object. However, the I/O constraint information associated with that port is intentionally cleared out during carving of the RM if the associated I/O buffer is part of an RP.
- Do not share I/O banks between static and RP.
  - I/O banks must be fully contained within either the static region or a single RP.
  - Sharing I/O banks across static and RP or between multiple RPs is not supported.

  - If I/O ports are constrained to a bank that is part of static, the tool raises an HDPR-29 DRC error during implementation.
- Verify I/O bank inclusion and buffer placement.
  - Ensure the Pblock for the RM includes the relevant I/O bank.
  - Validate the physical placement of the embedded IOBs is within the RM's Pblock.

Partition Pin Placement

Each pin of an RP has a partition pin (PartPin). By default the tools automatically place these PartPins inside of the RP Pblock range (which is required). For many cases, this automatic placement can be sufficient for the design. However, for timing-critical interface signals or designs with high congestion, it might be necessary to help guide the placement of the PartPins. The following is an example of how to achieve this:

- Define user `HD.PARTPIN_RANGE` constraints for some or all of the pins.

```
set_property HD.PARTPIN_RANGE {SLICE_Xx0Yx0:SLICE_Xx1Yy1
SLICE_XxNYyN:SLICE_XxMYyM}
[get_pins <rp_cell_name>/*]
```

  By default the `HD.PARTPIN_RANGE` is set to the entire Pblock range. Defining a user range allows the tools to place PartPins in the specified areas, improving timing and/or reducing congestion.

  IMPORTANT! *When examining the placement of PartPins, there are limited routing resources available along the edges, and especially in the corners, of the Pblock. The PartPin placer attempts to spread the partition pins, minimizing the number of partition pins per interconnect along the edges, and increasing the PartPin density towards the middle of the Pblock. When defining a custom `HD.PARTPIN_RANGE` constraint, be sure to make the range wide enough to allow for spreading, or you are likely to see congestion around the PartPins.*

Active-Low Resets and Clock Enables

In AMD 7 series FPGAs, there are no local inverters on control signals (resets or clock enables). The following description uses a reset as the example, but the same applies for clock enables.

If a design uses an active-Low reset, a LUT must be used to invert the signal. In non-DFX designs that use all active-Low resets multiple LUTs are inferred but can be combined into a single LUT and pushed into the I/O elements (the LUT goes away). In non-DFX designs that use a mix of High and Low, the LUT inverters can be combined into one LUT that remains in the design, but that has minimal effect on routing and the timing of the reset net (output of LUT can still be put on global resources). However, for a design that uses active-Low resets on a partition, it is possible to have inverters inferred inside the partition that cannot be pulled out and combined. This makes it impossible to put the reset on global resources, and can lead to poor reset timing and to routing issues if the design is already congested.

The best way to avoid this is to avoid using active-Low control signals. However, there are cases where this is not possible (for example, when using an IP core with an Advanced eXtensible Interface (AXI) interface). In these cases the design should assign the active-Low reset to a signal at the top level, and use that new signal everywhere in the design.

As an example:

```
reset_n <= !reset;
```

Use the `reset_n` signal for all cases, and do not use the `!reset` assignments on signals or ports.

This ensures that a LUT is inferred only for the reset net for the whole design and has a minimal effect on design performance.

Decoupling Functionality

Because the reconfigurable logic is modified while the device is operating, the static logic connected to outputs of RMs must ignore data from RMs during dynamic reconfiguration. The RMs do not output valid data until reconfiguration is complete and the reconfigured logic is reset. There is no way to predict or simulate the functionality of the reconfiguring module.

You must decide how the decoupling strategy is solved. A common design practice to mitigate this issue is to register all output signals (on the static side of the interface) from the RM. An enable signal can be used to isolate the logic until it is completely reconfigured. Other approaches range from a simple 2-to-1 MUX on each output port, to higher level bus controller functions.

The static design should include the logic required for the data and interface management. It can implement mechanisms such as handshaking or disabling interfaces (which might be required for bus structures to avoid invalid transactions). It is also useful to consider the down-time performance effect of a DFX module (that is, the unavailability of any shared resources included in a DFX module during or after reconfiguration).

DFX Decoupler IP is available, which allows you to insert multiplexers to efficiently decouple AXI4-Lite, AXI4-Stream, and custom interfaces. More information about the DFX Decoupler IP is available on the Xilinx.com website.

Initializing Reconfigurable Modules after DFX

RM Initialization Requirements

Due to of a variety of factors, reconfigurable modules (RMs) require in-system mechanisms to ensure they synchronize with the rest of the system after loading. Failure to include these mechanisms can result in sporadic error conditions. Non-DFX designs also require these mechanisms (see AR 44174), although issues are less likely. In DFX designs, the clocks are often already running when reconfiguration happens, which makes synchronization issues between the RM and the static shell much more likely to occur.

When you use partial programming images include steps that occur after you load design data into the target device. At a high level, the RM startup sequence automatically includes these fundamental steps:

- Assert GRESTORE to initialize all of the synchronous elements to their initial values
- Assert global write enable (GWE) to make synchronous elements writable from the PL
- Assert end of startup (EOS) to signal that all configuration startup tasks are complete

This list is not exhaustive, and additional activities occur between these steps. Importantly, configuration is not complete until EOS asserts, and it is not safe to enable the RM logic until this point.

The global signal GWE is not synchronised to your user clock and can take substantial time to propagate to all parts of the device, especially for exceptionally large RPs in devices that use SSI technology. As a result, the synchronous elements in an RM can start operating on different user clock cycles, corrupting the RM's state. Because GWE asserts after GRESTORE, the configuration process has no opportunity to return the RM to an uncorrupted state.

The RM is in a partially operational state in the period during the period between GWE asserting and EOS asserting. Individual DFFs can operate, but you cannot predict which ones operate or what they do. For example, a 4-bit state vector might have bits 1 and 3 activate and transition to new value, while bits 0 and 2 remain inactive and unable to change. When this happens, the RM is in a corrupted state.

There are three solutions to this problem, which individually or combined can help avoid DFX designs from getting into a corrupted state.

- Apply a reset to the RM until EOS asserts, or longer if required.

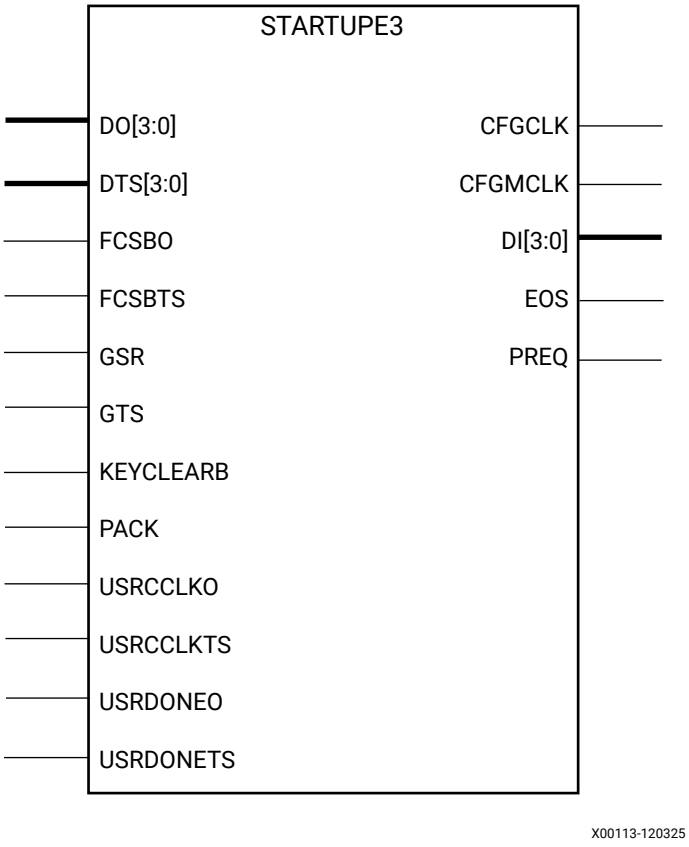
- Stop the user clock until EOS asserts, or longer if required.
- Deassert clock enable to all synchronous elements until EOS asserts, or longer if required.

Access to End of Startup (EOS)

You must keep an RM isolated while partial reconfiguration is occurring. After you complete the full reconfiguration event, including the dedicated initialization, you can release the decoupling. Your configuration management solution or embedded software can control the release of decoupling by sending a signal to the PL when ready. You can also use the end of startup (EOS) pin to drive the release, which indicates when configuration events have completed.

In AMD UltraScale™ and AMD UltraScale+™ devices, the EOS pin is located on the `STARTUP` block. Instantiate the `STARTUPE3` primitive and connect the EOS pin to the control logic that releases decoupling in your design.

Figure 65: **STARTUPE3 Primitive**



In first-generation Versal devices, you can access the EOS signal from the CIPS IP, but you cannot select it from the customization GUI. You must enable the pin in Tcl using the following command:

```
set_property CONFIG.PS_PMC_CONFIG(PS_USE_STARTUP) {1} \
           [get_bd_cells versal_cips_0]
```

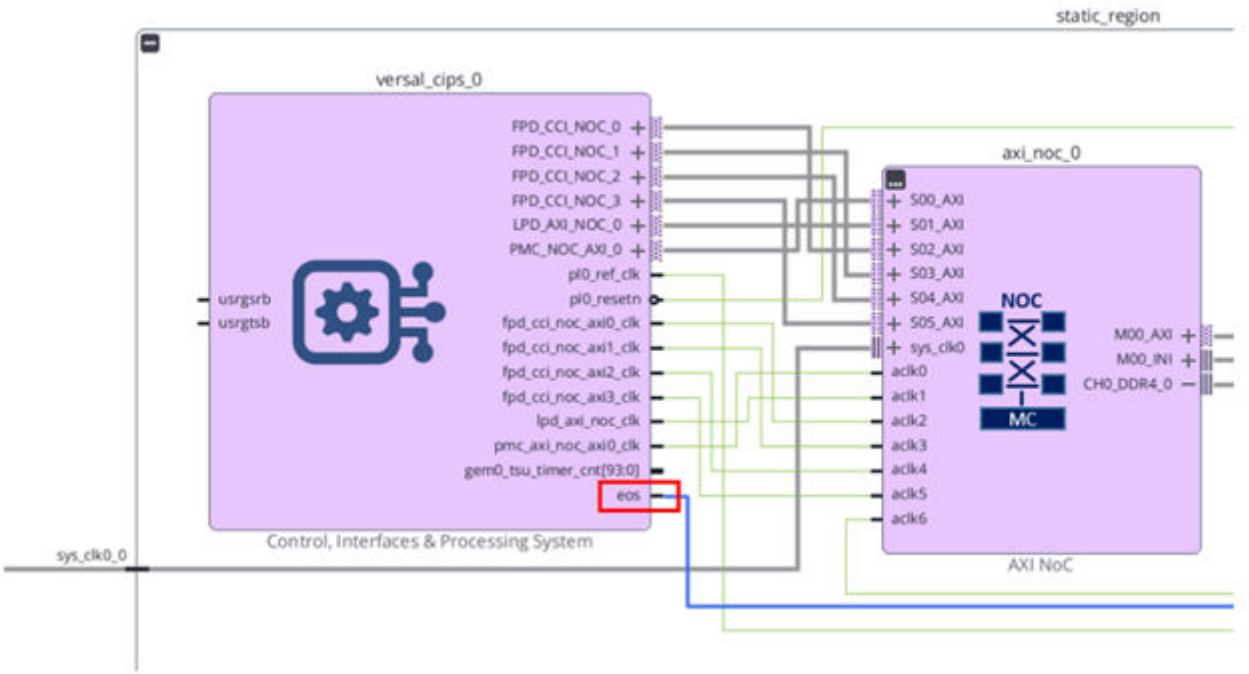
***Note*:** Modify `veral_cips_0` to match the name in your design.

In second generation Versal devices that use the PS Wizard IP for AMD Versal™ AI Edge Series Gen 2, AMD Versal™ Prime Series Gen 2, and AMD Versal™ Premium Series Gen 2 devices, the syntax is similar, with only the IP reference changed:

```
set_property CONFIG.PS11_CONFIG(PS_USE_STARTUP) {1} \
           [get_bd_cells versal2_ps_wizard_0]
```

***Note*:** Modify `versal2_ps_wizard_0` to match the name in your design.

Figure 66: **EOS Pin Enabled on the CIPS IP**



After you expose the EOS pin, connect it to the control logic that releases decoupling in your design.

Reset Generation Methods

Externally Generated Reset

In this solution, you generate the reset signal for the RM in the static design and route it into the RM. Each RP needs its own reset so you can reset the RM independently after loading. You need to assert these resets when the system wants to reset a particular RM and whenever any system-wide reset occurs.

If you apply a reset after EOS, the RM returns to a known state, as long you design the RM correctly (see Potential Problems for more information). However, you must prevent an RM from entering into a corrupted state in the first place. To do that, assert the reset before GWE asserts, and hold it until EOS asserts. You can hold the reset longer if your system needs extra time before the RM becomes operational.

Assume you have a design-specific signal called `rm_being_loaded` with the following properties:

- Asserted when the design requests the RM to be loaded
- Deasserted (at the earliest) when EOS is detected

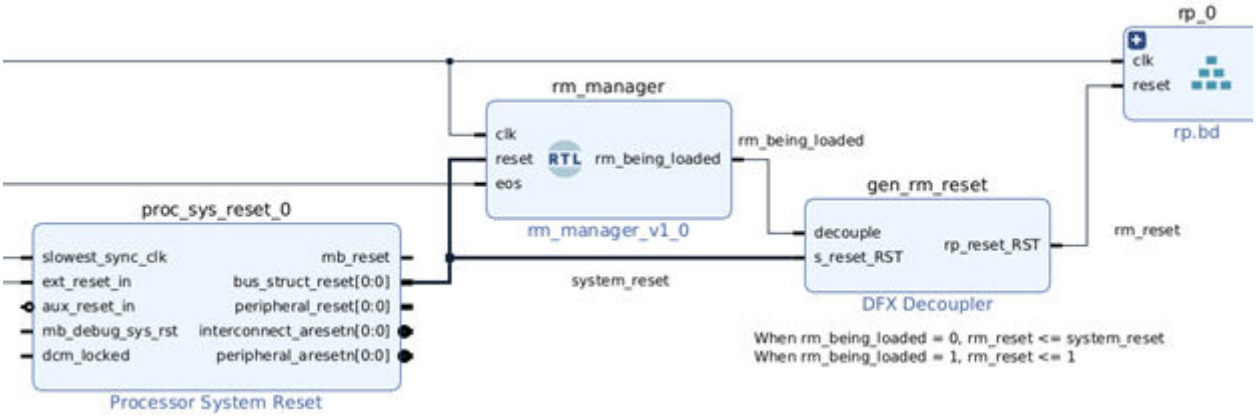
You can then generate the RM reset as the following:

```
rm_reset <= system_reset OR rm_being_loaded;
```

You can register the signal to meet timing if necessary. This example assumes active-High resets. Adjust the equation for active-Low resets accordingly.

Alternatively, you can use `rm_being_loaded` to control a DFX decoupler IP on the `system_reset` input to the RM.

Figure 67: **Using a DFX Decoupler to Help Generate a Reset from an RM**

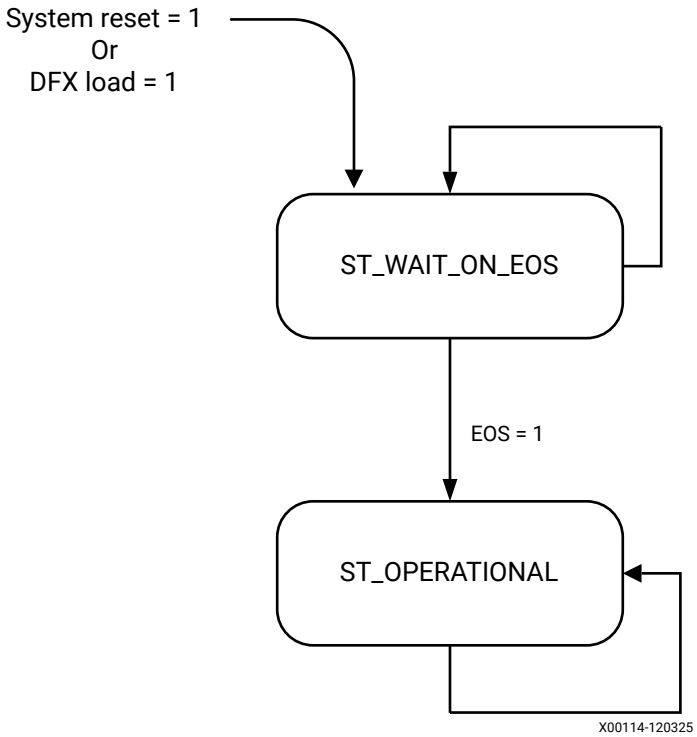


Configure the DFX decoupler to drive the reset active-level (1 in this example) when decoupled. You can select a global buffer to drive the signal if necessary. This approach can be useful if the DFX decoupler IP is already being used to decouple the RM's boundary during DFX.

Internally Generated Reset

An alternative way to generate a reset for an RM is to add code to the RM so it self-generates a reset when it loads. This method requires a finite state machine (FSM) with the following states:

Figure 68: **State Machine Required to Generate a Reset Inside an RM**




```
rm_reset <= '1' when state = ST_WAIT_ON_EOS else '0';
```

The state machine returns to `ST_WAIT_ON_EOS` if a system reset occurs, or if you load the RM using DFX. In both cases, the transition occurs because the underlying DFF is reset, not because of any combinatorial logic.

In this FSM, assert the reset signal to the remainder of the RM in `ST_WAIT_ON_EOS`, and deassert it in `ST_OPERATIONAL`. You need an FSM like this because EOS asserts when RMs in other RPs load, so your RM cannot use EOS directly to generate its own reset signal. Instead, it must respond to EOS only when it is loading. This FSM is sensitive to EOS only when you have loaded the RM or when there is a system-wide reset, in which case it ignores EOS because EOS remains high during the reset.

This approach has several disadvantages compared to an externally generated reset, such as:

- The FSM can only be a single bit unless you carefully place data flip-flops (DFFs) that all see global write enable (GWE) at the same time. This step can be overlooked, and extra states can be added later by someone who does not fully understand the impact.
- You must include and verify the FSM in each RM individually.
- You must route both the system reset and EOS signals into the RM. An externally generated reset only requires one signal to be routed.
- It doesn't handle blocks in the RM that have an edge-sensitive reset. Although you can add extra states to support this, doing so is vulnerable to the problem described in #1.

This approach provides no advantages over an externally generated reset and introduces risks and limitations. AMD does not recommend it.

Potential Problems

Non-resettable Primitives

If your RM contains any non-resettable primitives, a reset alone might not be enough to reverse actions triggered while the RM was in a corrupted state. For example, a corrupted state machine might write to a random address in local distributed memory that must start with specific initial values.

Any RM that contains these primitives and depends on the circuit's initial state can fail. To handle corruption in this case, you must include design-specific circuits that ensure a safe starting state by explicitly initializing these primitives to the correct values. Trigger these circuits on reset assertion or deassertion, depending on your design requirements.

You can mitigate this problem by asserting the RM reset before GWE asserts and keeping it asserted until after EOS. You do this by ensuring all RM primitive power-on values do not trigger any actions and by decoupling any inputs from static logic that could trigger actions. However, risks still remain. See the following sections for details.

Edge-triggered Resets

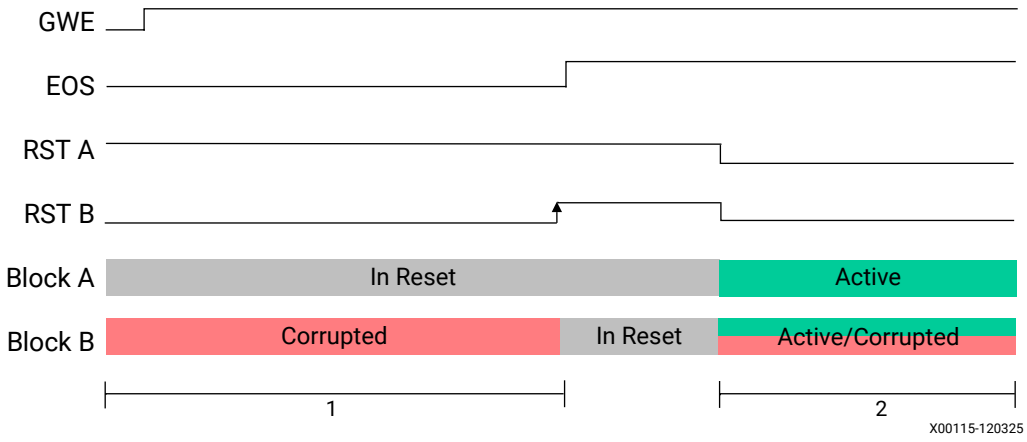
Some blocks in an RM can be sensitive to a specific edge of the reset signal, either to perform their own reset or to trigger operation. For example, a watchdog counter designed to ensure that the reset phase exits might reset itself on the transition of an active-High reset from 0 to 1, and then count while reset remains at 1.

If your RM contains such a block, you must assume it can become corrupted from the moment GWE asserts until it sees the required reset edge. During this time, you should isolate it from the rest of the RM by using DFX decouplers to limit any potential corruption. You can assume the remainder of the RM is held in reset with safe initial values, making corruption of the block from the RM unlikely, but isolating its inputs with decouplers is still a wise precaution.

The safest way to handle a block like this, in addition to the decoupling described earlier, is to route a dedicated reset to it. While you hold the remainder of the RM in reset with an active-level reset, you can pulse the dedicated reset for the edge-triggered block. This limits the chances of the edge-triggered block corrupting the remainder of the RM. With careful timing, you can align the active edge of the reset signal with the deassertion of the level-based reset, allowing all parts of the RM to start operating on the same clock edge.

Block A has an active-High level-base reset. Block B has an edge-triggered reset, sensitive to the rising edge.

Figure 69: **Using Multiple Reset Signals to Handle Level and Edge Sensitive Resets in an RM**



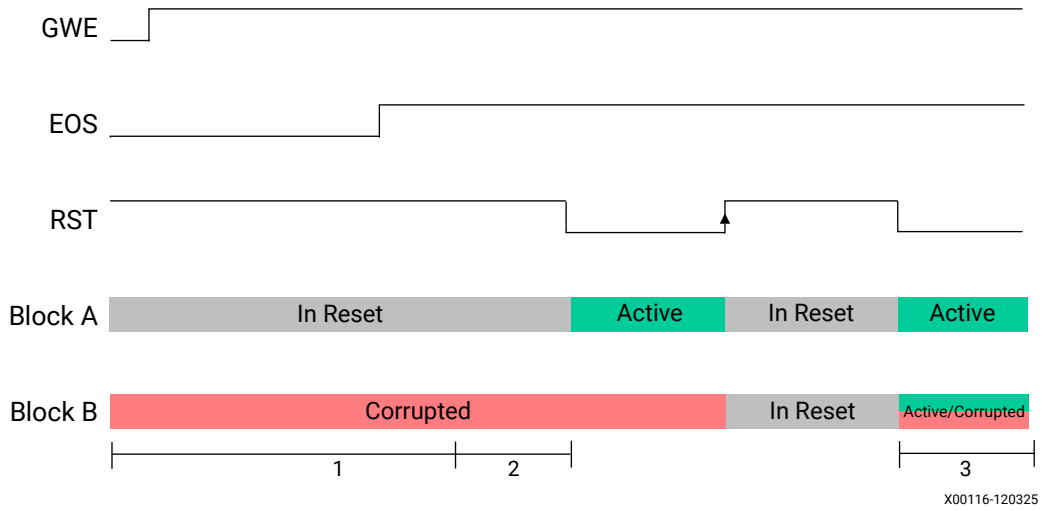
The problems here are in:

- Phase 1, where block B might be corrupted.
- Phase 2, where block B might remain corrupted depending on the efficacy of the reset.

Another way to handle this is to use a single reset signal for all parts of the RM. In this case, you need to hold reset asserted until you see EOS, and then pulse it to generate the edge required by the edge-triggered block. The edge-triggered block can become corrupted from the assertion of GWE until it detects the active reset edge. This approach assumes the reset can fully restore the block's state.

Block A has an active-High level-base reset. Block B has an edge-triggered reset, sensitive to the rising edge.

Figure 70: **Using a Single Reset Signal to Handle Level and Edge Sensitive Resets in an RM**



The problems here are in:

- Phase 1, where block B is corrupted.
- Phase 2, where block A is active and block B is corrupted.
- Phase 3, where block B might remain corrupted depending on the efficacy of the reset.

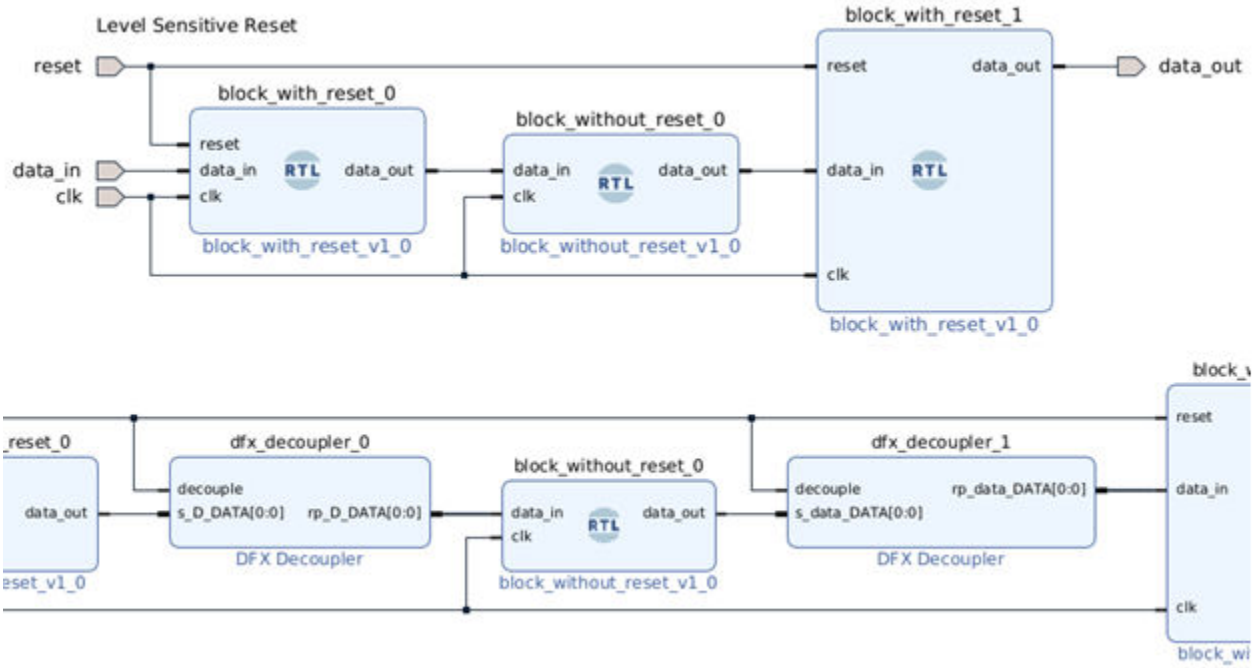
The two-reset approach is superior, although much depends on your reset's ability to bring the block into a known good state. For example, it should be able to repair possibly corrupted memory entries.

In both cases, parts of the RM can become corrupted, and all you can do is try to limit the extent of any infection.

Blocks With no Reset

If any blocks in the RM lack reset inputs, your only option might be to stop the clock, either to the entire RM or to those specific blocks, until EOS is asserted. You might be able to surround these blocks with decouplers inside the RM, allowing you to force all inputs to non-triggering safe values until EOS is reached and prevent problematic signals from contaminating the rest of the RM. However, this might not stop a block without a reset from becoming internally corrupted with no recovery mechanism.

Figure 71: **Using DFX Decouplers to Isolate a Block with No Reset Inside an RM**



This block cannot be reset, but by decoupling its outputs, you can (perhaps temporarily) avoid corruption of the rest of the RM. Decouple its inputs to reduce the chance that other RM signals put it into a non-idle state. If it is not already corrupted from GWE, you are trying to avoid accidentally starting it.

***Note*:** If the block becomes corrupted by GWE, there is no recovery mechanism, and you are relying on it to be self-healing.

Additional Considerations

While your aim is to avoid an RM entering a corrupted state, this might not be possible if it contains components that have edge-sensitive resets or no resets at all. In this case, corruption of the static is a risk. For example, a corrupted RM state machine might accidentally initiate an AXI write transaction from the RM to the static. You should use DFX Decouplers to protect the static until you put the RM into a safe state. See *Dynamic Function eXchange Decoupler IP LogiCORE IP Product Guide* (PG375) for more information.

While the aim is to avoid an RM entering a corrupted state, that might not be possible if it has components that have edge-sensitive, or no, resets. In this case, corruption of the static is a risk. For example, a corrupted RM state machine might accidentally initiate an AXI write transaction from the RM to the static. DFX Decouplers should be used to protect the static until the RM is put into a safe state. See PG375 for more information.

Black Boxes

You can implement an RP as a *pseudo* black box, referred to in Vivado as a greybox. To do this, the RP must be a black box in the static design (either from bottom-up synthesis results or from running `update_design -black_box`). Then the black box can have LUT1 buffers placed on all inputs and outputs using the command `update_design -buffer_ports` on the black box RP cell:

```
update_design -cell <rp_cellName> -buffer_ports
```

Now you can run this design through implementation to place and route the LUT1 buffers (and static logic, if not already placed and routed).

All the inserted LUT1 output buffers are tied to a logical 0. If it is necessary to drive a logical 1 from the RP outputs, this can be controlled using an RP pin property called `HD.PARTPIN_TIEOFF`. This property can be set at any time (all the way up to `pre-write_bitstream`), and it controls the LUT equation of the LUT1 buffer connected to the specified port. The default value is '`0`', which configures the LUT as an inverter (output is 0) which then ties off to V_{cc}. Setting this property to '`1`' configures the LUT as a route-thru (output is 1). You might have to change the output value in some design situations.

```
set_property HD.PARTPIN_TIEOFF 1 [get_pins <RP_cellName>/<output_pinName>]
```

***Note*:** Do not apply values of TRUE or FALSE for this property. That syntax will only result in values of '1' or '0' respectively for driverless RM output ports; it will have no effect on output ports already tied to VCC or GND.

The greybox has no user logic (just the tool-inserted LUT1 buffers). The greybox bitstream contains information for these LUTs, as well as any static logic/routes that use resources inside the RP frames. Static routes that pass through the region, including interface nets up to the partition pin nodes, exist within this region. Programming information for these signals is included in the black box programming bitstream.

Use of greyboxes is an effective way to reduce the size of a full configuration BIT file, and therefore reduce the initial configuration time. The compression feature might also be enabled to reduce the size of BIT files. This option looks for repeated configuration frame structures to reduce the amount of configuration data that must be stored in the BIT file. The compression results in reduced configuration and reconfiguration time. When the compression option is applied to a routed DFX design, all of the BIT files (full and partial) are created as compressed BIT files. To enable compression, set this property prior to running `write_bitstream`:

```
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
```

Effective Approaches for Implementation

There are trade-offs associated with optimizing any FPGA design. Dynamic Function eXchange is no different. Partitions are barriers to optimization, and reconfigurable frames require specific layout constraints. These are the additional costs to building a reconfigurable design. The additional overhead for timing and area needs vary from design to design. To minimize the impact, follow the design considerations stated in this guide.

When building Configurations of a reconfigurable design, the first Configuration to be chosen for implementation should be the most challenging one. Be sure that the physical region selected has adequate resources (especially elements such as block RAM and DSP48) for each RM in each RP, then select the most demanding (in terms of either timing or area) RM for each RP. If all of the RMs in the subsequent Configurations are smaller or slower, it is easier to meet their demands. Timing budgets should be established to meet the needs of all RM.

If it is not clear which RM is the most challenging, each can be implemented in parallel in context with static, allowing static to be placed and routed for each. Examine resource utilization statistics and timing reports to see which configuration met design criteria most easily and which had the tightest tolerances, or which missed by the widest margins.

IMPORTANT! *Focus attention on the configuration that is the furthest from meeting its goals, iterating on design sources, constraints, and strategies until needs are met. At some point, one configuration must be established as the golden result for the static design, and that implementation of the static logic will be used for all other configurations.*

Building Up Implementation Requirements

Implementation of Dynamic Function eXchange designs requires that certain fundamental rules are followed. These rules have been established to ensure that a partial bitstream can be accurately created and safely delivered to an active device. As noted throughout this document, these rules include these basic premises:

- The logical and physical interface of a RP remains consistent as each RM is implemented.
- The logic and routing of a RM is fully contained within a physical region which is then translated into a partial bitstream.
- The logic of the static design must be kept out of the reconfigurable region if the dedicated initialization feature is used.

These requirements necessitate specific implementation rules for optimization, placement and routing. Application of these rules might make it more difficult to meet design goals, including timing closure. A recommended strategy is to build up this set of requirements one at a time, allowing you to analyze the results at each step. Starting with the most challenging configuration and the full set of timing constraints, implement the design through place and route and examine the results, making sure you have sufficient timing slack and resources available to continue to the next step.

1. Implement the design with no Pblocks. Use bottom-up synthesis and follow general Hierarchical Design recommendations, such as registered boundaries, to achieve a baseline result.
2. Add Pblocks for the design partitions that will later be marked reconfigurable. This floorplan can be based on the results established in the bottom-up synthesis run from Step 1. Logic from the RMs must be placed in the Pblocks, but static logic may be included there as well.

   While creating these Pblocks, the `HD.RECONFIGURABLE` property (and optionally, the `RESET_AFTER_RECONFIG` property) can be added temporarily to run PR-specific Design Rule Checks. This ensures that the floorplan created meets PR size and alignment requirements.
3. With the floorplan established, separate the placement of static design resources from those to be reconfigurable by adding the `EXCLUDE_PLACEMENT` property to the Pblocks. This keeps static logic placed outside the defined Pblocks.
4. Keep the routing for RMs bound within the Pblocks by applying the `CONTAIN_ROUTING` property to the Pblocks. With the properties from this and the previous step, the only remaining rules relate to boundary optimization procedures as well as PR-specific Design Rule Checks.
5. Finally, mark the RP Pblocks as `HD.RECONFIGURABLE`. The `EXCLUDE_PLACEMENT` and `CONTAIN_ROUTING` properties are now redundant and can be removed.

If design requirements are not met at any of these steps, you have to opportunity to review the design structure and constraints in light of the newly applied implementation condition.

Configuration Analysis Report

The Dynamic Function eXchange design flow uses multiple versions of the design that must be implemented through place and route. These different configurations have common static design results, but differing modules within each RP. Designers must set up timing constraints and floorplans that account for these different modules that are swapped on the fly.

The DFX Configuration Analysis report compares each RM that you select to give you information on your DFX design. It examines resource usage, floorplanning, clocking, and timing metrics to help you manage the overall PR design.

The DFX Configuration Analysis report is currently run in the Tcl Console or within a Tcl script. The top level design must be open before issuing this command:

```
report_pr_configuration_analysis -cells <RP_name> -dcps
{<list_of_RM_checkpoints>}
```

Either select a single cell (RP) and multiple DCPs (each representing an RM) that can be inserted into that cell for a comprehensive analysis of that RP, or select multiple cells with no subsequent DCPs for a top-level analysis of the static design and interfaces into each RP.

By default, three aspects of the PR design are analyzed. You can select one or more of these switches to narrow the focus of the report.

- The `-complexity` switch focuses the report on resource usage, including the maximum number of each resource type required for the RP.
- The `-clocking` switch focuses the report on clock usage and loads for each RM, helping you plan the overall clocking distribution of the design.
- The `-timing` switch focuses the report on boundary interface timing details, allowing you analyze bottlenecks in and out of RMs.

Additionally, the `-rent` switch adds Rent metrics to the report. The Rent exponent calculates the routing complexity and can be an indication of how much congestion is likely to be seen. For more information on Rent, see the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906). This option can take a long time to run on large designs.

When this analysis is done, each RM is examined based on information in the checkpoints provided. While post-synthesis checkpoints can be supplied, if the RM contains IP that have been synthesized out-of-context, or if debug cores are to be inserted, information is missing from these checkpoints. The most complete information is not available until after `opt_design` when all the linking and expansion has been done. AMD advises you to create fully assembled RM checkpoints after `opt_design` by calling `write_checkpoint -cell` for each configuration, then run the configuration analysis report using these files.

Here are some example sections from a report for a design with a single RP that has three RM.

Complexity

First is the resource usage table for the `-complexity` switch:

```
+------------------+---------+------------+------------+------------+------------+------------+
| Categories       |Grid Type| Current    |         RM1 |         RM2 |         RM3 |         Max |
+------------------+---------+------------+------------+------------+------------+------------+
|Slice Logic       |         |            |            |            |            |            |
| Slice LUTs       |   SLICE | 936(23.40%)| 936(23.40%)| 927(23.17%)|1091(27.28%)|1091(27.28%)|
|  LUT as Logic    |   SLICE | 836(20.90%)| 836(20.90%)| 827(20.67%)| 977(24.43%)| 977(24.43%)|
|  LUT as Memory   |   SLICE |  100(5.00%)|  100(5.00%)|  100(5.00%)|  114(5.70%)|  114(5.70%)|
|  LUT as Distributed RAM| SLICE | 32(1.60%)| 32(1.60%)|   32(1.60%)|   42(2.10%)|   42(2.10%)|
|  LUT as Shift Register | SLICE | 68(3.40%)| 68(3.40%)|   68(3.40%)|   72(3.60%)|   72(3.60%)|
|  Slice Registers |   SLICE |1775(22.19%)|1775(22.19%)|1613(20.16%)|1654(20.68%)|1775(22.19%)|
|  Register as Flip Flop | SLICE |1775(22.19%) | 1775(22.19%)| 1613(20.16%)| 1654(20.68%)|
1775(22.19%)|
```

```
| CARRY8              |   SLICE |  14(2.80%)|    14(2.80%)|   16(3.20%)|   16(3.20%)|   16(3.20%)|
| F7 Muxes            |   SLICE |   6(0.30%)|     6(0.30%)|    6(0.30%)|    6(0.30%)|   6(0.30%)|
| Unique Control Sets | SLICE | 105(21.00%)| 105(21.00%)|100(20.00%)| 102(20.40%)| 105(21.00%)|
|Memory               |         |            |              |            |            |            |
|   RAMB18            |  RAMB18 |   1(5.00%)|     1(5.00%)|   3(15.00%)|   2(10.00%)|   3(15.00%)|
| PPLOCs (INT Tile Ratio)| - | 28(0.09)| 28(0.09)| 28(0.09)| 28(0.09)| 28(0.09)|
+---------------------+---------+------------+------------+------------+------------+------------+
```

Notice that RM1 requires the most resources for Slice Registers, RM2 requires the most block RAM, and RM3 requires the most Slice LUTs. These maximum values for each resource type are summarized in the Max column—this column should be used to plan Pblock resource sizes. Remember that additional overhead is advised—packing densities for a given RP is similar to a complete design.

Clocking

The `-clocking` switch summarizes the full set of clocks in the design, then breaks down the clock loads in each RM. It also reports the number of RM clock loads in each clock region (not shown here).

```
Static Clock Summary
+-----------------------------------------------------------------------+------------+-------------+
|                                                            Clock Name |Static Loads|RP1 Max Loads|
+-----------------------------------------------------------------------+------------+-------------+
|mb_prc_wrapper/mb_prc_i/ddr4/inst/u_ddr4_infrastructure/dbg_clk        |       2889 |           0 |
|mb_prc_wrapper/mb_prc_i/ddr4/inst/u_ddr4_infrastructure/CLK            |       1639 |           0 |
|dbg_hub/inst/BSCANID.u_xsdbm_id/itck_i                                 |        496 |         365 |
|mb_prc_wrapper/mb_prc_i/ddr4/inst/u_ddr4_infrastructure/addn_clkout1   |       8534 |        1403 |
|mb_prc_wrapper/mb_prc_i/ddr4/inst/u_ddr4_infrastructure/c0_ddr4_ui_clk |      15098 |           0 |
|mb_prc_wrapper/mb_prc_i/ddr4/inst/u_ddr4_infrastructure/addn_ui_clkout2|      11791 |           0 |
+-----------------------------------------------------------------------+------------+-------------+
Reconfigurable Module Clocking RP1
+-------------------------------------------+-------+-----------+----------+----------+----------+
| Clock Name                                |Current|  RM1 Loads| RM2 Loads| RM3 Loads| Max Loads|
+-------------------------------------------+-------+-----------+----------+----------+----------+
|Static Clocks                              |       |           |          |          |          |
| dbg_hub/inst/BSCANID.u_xsdbm_id/itck_i    |   365 |       365 |      360 |      350 |      365 |
| mb_prc_wrapper/mb_prc_i/ddr4/inst/        |       |           |          |          |          |
| u_ddr4_infrastructure/addn_clkout1        |  1403 |      1403 |     1385 |     1344 |     1403 |
+-------------------------------------------+-------+-----------+----------+----------+----------+
```

Timing

The `-timing` switch analyzes the worst interface paths on the RP boundary based on logic levels. The default is to examine the 10 worst paths but this can be changed using the `-nworst` option. The Logic Path field shows the levels of logic and defines if each level is in the static (S) or RM partition. Here is a sample of a single boundary path:

```
Reconfigurable Module Boundary Timing RP1
+----------------------+-------------------------------------------------------------------------------+
|    Characteristics   |                                    Paths                                      |
+----------------------+-------------------------------------------------------------------------------+
| Path #1              |                                                                       ------- |
|   RP Boundary Pin    |                                                                S_BSCAN_shift |
|   RM With Worst Path |                                                      RP1 1st Configuration |
|   Static Logic Levels|                                                                           3 |
|   RM Logic Levels    |                                                                           2 |
|   Logic Path         |         FDRE(S) LUT3(S) LUT6(S) LUT3(S) LUT4(RM) LUT6(RM) FDRE(RM)   |
|   Start Point Clock  |                                                                      itck_i |
|   End Point Clock    |                                                                      itck_i |
|   High Fanout        |                                                                          45 |
|   Boundary Fanout    |                                                                           1 |
+----------------------+-------------------------------------------------------------------------------+
```

This information can help you optimize boundary paths. Insertion of pipeline registers can break up these timing challenges and even create a decoupling point between reconfigurable and static logic.

Summary

Run the `report_pr_configuration_analysis` command early in your design flow, after you have established the logic in each RM but before you finalize the floorplan of the design. This report helps you optimize each Pblock for the RPs in your design, give you guidance on the clocking usage throughout the design, and provide insight as you close timing on each configuration in the overall project.

Managing Constraints for a DFX Design

Dynamic Function eXchange requires all the same constraints as any other design, and requires additional physical constraints (Pblocks) and might require various sets of constraints that define the various configurations defined by a set of RMs. Constraints be defined as either global or scoped.

- **Global:** These are constraints that are applied to the entire design, and all object references (cell/pin/net/port) are with respect to the full design hierarchy.
- **Scoped:** These are constraints that are written with respect to a module other than the top module and are scoped to a single or all instances of that module. For the following discussion on Dynamic Function eXchange, it is assumed scoped constraints are scoped to one or more instances of a RM. For more information on XDC scoping, see the *Vivado Design Suite User Guide: Using Constraints* (UG903).

Constraint Creation

Constraints should be broken up into multiple files based on the type of constraint. For Dynamic Function eXchange, AMD recommends breaking up the design constraints into the three following types:

- **Static:** Constraints (physical or timing) that reference only static objects (cell/pins/nets/ports), and do not reference any objects within an RP. These are global constraints and are applied to static synthesis, or at implementation of the initial configuration. It is not recommended to reapply these constraints for subsequent configurations where static is imported, as these constraints should already exist within the static DCP and reapplying them can cause unintended constraint interactions.

- **Boundary:** These are timing constraints (such as `set_false_path`) that reference objects in both static and the RP. Write these constraints referencing the RP pin object, and not objects internal to the RM. For example, take the following two constraints:

```
set_false_path -from [get_pins static_reset/C] -through [get_pins rp_inst/
rst]
set_false_path -from [get_pins static_reset/C] -to [get_pins rp_inst/
*foo*/D]
```

  The first constraint is preferred, as this constraint will not be lost when the RMs are converted to a black box after the initial configuration. Even when the RMs are carved out, the RP interface still exists in the static design, and the `rp_inst/rst` pin referenced in the constraint still exists in the design. If a constraint is defined using objects inside the RM, as shown in the second constraint, the constraint becomes invalid and is dropped from the design after carving. These constraints would have to be reapplied for every configuration if written using this syntax.

- **RM:** Constraints (physical or timing) that reference only RM logic. While these constraints should all be scoped to the RM, how they get applied will depend on what type of constraints they are:

  - **General RM constraints:** These constraints apply to every instance of the RM. An example is a timing constraint, like a false path, multi-cycle exception, or a `create_clock` for a local RM clock. These apply to the RM regardless of the physical location.

  - **RP specific RM constraints:** These constraints are specific to the location (Pblock) in which the RM is being implemented. An example is physical constraints like specific block RAM placement, or PACKAGE_PIN constraints for embedded I/O.

  - **Embedded I/O constraints:** Embedded I/O buffers (IOBs) are I/O buffers instantiated within IPs such as DDR or GTs, so you do not need top-level I/O buffers. When you do not use such IPs in an RM compile, set `IO_BUFFER_TYPE=NONE` at the top RTL to prevent buffer insertion. The I/O pads should still connect to the RM hierarchy but remain unconnected inside the RM.

    To guide I/O placement, retain `PACKAGE_PIN` constraints even when the I/O is unused. For RPs with embedded I/O, reapply I/O constraints (`PACKAGE_PIN`, `IOSTANDARD`, and direction) for every RM configuration, because Vivado clears these constraints during RM carving.

    See Embedded I/O Usage Guidelines for more information on using embedded I/O.

In addition to this, it is also recommended to separate physical (Pblock, LOC, PACKAGE_PIN, etc) and timing constraints into separate XDC files. This allows for better control of when constraints are applied (synthesis vs implementation), and easier management of constraint files if constraints need to be modified.

Constraint Application

Once all of the constraint have been broken up, then the method of how and when to apply these constraints become much easier.

- **Static Timing Constraints:** Apply these at static synthesis and mark them for use in synthesis as well as implementation so that they are passed into the static synthesis DCP. As long as these constraints are properly applied, and exist in the post synthesis DCP, there is no need to reapply them at implementation time.
- **Static Physical Constraints:** Apply these at implementation of the initial configuration. There is no need to reapply these for subsequent configurations, as the routed static DCP contains all of these, as well as the static timing constraints.
- **Boundary Timing Constraints:** Apply these at implementation of the initial configuration. If the constraints were written without referencing any objects internal to the RM, then these constraints do not need to be reapplied for subsequent configurations.
- **General RM Constraints:** Apply these at RM synthesis time and mark them for use in synthesis and implementation. These constraints exist in the RM DCP and are applied when the DCP is linked into the full design.

  IMPORTANT! *If there are additional OOC specific timing constraints that are used for OOC synthesis only (such as a* `create_clock` *for a module port), mark these for use in* `out_of_context` *so that they do not get applied to the full design. Failure to do this can result in unwanted constraint interaction when the top-level constraints are applied and conflict with these lower level OOC constraints.*

- **RP Specific RM Constraints:** Apply these at implementation of any configuration involving the specific RM at the specific RP location. All physical constraints within the RM (including IOB) are cleared out during carving, and must be reapplied.

If a post-route_design RM DCP is being reused, the physical information (block RAM, IOB, etc) is all included within the RM DCP and there is no need to reapply these constraints. However, the RM DCP (created with `write_checkpoint -cell`) does not contain any RM specific timing constraints, and all general RM constraints need to be applied as well as any boundary timing constraint that reference RM objects.

Defining Reconfigurable Partition Boundaries

Partial reconfiguration is done on a frame-by-frame basis. As such, when partial BIT files are created, they are built with a discrete number of configuration frames. The size of a partial bit file depends on the number and type of frames included. You can see this size in the header of a raw bit file (`.rbt`) created by `write_bitstream -rawbitfile`.

Partition boundaries do not have to align to reconfigurable frame boundaries, but the most efficient place and route results are achieved when this is done. Static logic is permitted to exist in a frame that will be reconfigured, as long as:

- It is outside the area group defined by the Pblock
- It does not contain dynamic elements such as block RAM, Distributed (LUT) RAM, or SRLs (7 series only).

When static logic is placed in a reconfigured frame, the exact functionality of the static logic is rewritten, and is guaranteed not to glitch.

Irregular shaped Partitions (such as a T or L shapes) are permitted but discouraged. Placement and routing in such regions can become challenging, because routing resources must be entirely contained within these regions. Boundaries of Partitions can touch, but this is not recommended, as some separation helps mitigate potential routing restriction issues as these partitions connect to the static design. Nested or overlapping RPs (partitions within partitions) are not permitted. Design rule checks (**Reports → Report DRC**) validate the Partitions and settings in a PR design.

Only one RP can exist per physical Reconfigurable Frame.

A Reconfigurable Frame is the smallest size physical region that can be reconfigured, and its height aligns with clock region or I/O bank boundaries. A Reconfigurable Frame cannot contain logic from more than one RP. If it were to contain logic from more than one RP, it would be very easy to reconfigure the region with information from an incorrect RM, thus creating contention. The software tools are designed to avoid that potentially dangerous occurrence.

Avoiding Deadlock

Some transactions across an RM boundary can take multiple cycles to complete. Removing an RM after a transaction has started but before it completes causes the system to deadlock (for example, the master, which initiated the transaction, waits for a response from a slave which no longer exists).

Additionally, the RM itself can cause deadlock. For example, assume some software is polling an RM register for a particular value. If the RM is removed, the software might stall as it continues to wait. It could also stall while waiting on a large block transfer to complete.

Any Dynamic Function eXchange design should be built with some sort of handshaking, ensuring that the removal of a RM occurs when it is safe to do so. This request or acknowledgment pairing is part of the user design and can be built in any fashion you deem appropriate.

Design Revision Checks

A partial bitstream contains programming information and little else, as described in Configuring the Device. While you do not need to identify the target location of the bitstream (the die location is determined by the addressing that is part of the BIT file), there are no checks in the hardware to ensure the partial bitstream is compatible with the currently operating design. Loading a partial bitstream into a static design that was not implemented with that RM revision can lead to unpredictable behavior.

AMD suggests that you prefix a partial bitstream with a unique identifier indicating the particular design, revision and module that follows. This identifier can be interpreted by your configuration controller to verify that the partial bitstream is compatible with the resident design. A mismatch can be detected, and the incompatible bitstream can be rejected, before being loaded into configuration memory. This functionality must be part of your design, and would be similar to or in conjunction with decryption and/or CRC checks, as described in *PRC/EPRC: Data Integrity and Security Controller for Partial Reconfiguration Application Note* (XAPP887).

A bitstream feature provides a simple mechanism for tagging a design revision. The `BITSTREAM.CONFIG.USR_ACCESS` property allows you to enter a revision ID directly into the bitstream. This ID is placed in the `USR_ACCESS` register, accessible from the FPGA programmable logic through a library primitive of the same name. Partial Reconfiguration designs can read this value and compare it to information a user can add to a header of a partial bitstream to confirm the revisions of the design match. More information on this switch can be found in the application note *Bitstream Identification with USR_ACCESS using the Vivado Design Suite* (XAPP1232).

CAUTION! *Do not use the TIMESTAMP feature because this value is not consistent for each call to `write_bitstream`. Only select a consistent, explicit ID to be used for all `write_bitstream` runs.*

Related Information

Configuring the Device

Simulation and Verification

Configurations of Dynamic Function eXchange designs are complete designs in and of themselves. All standard simulation, timing analysis, and verification techniques are supported for DFX designs, though simulation is not currently supported within the Vivado project environment. Partial reconfiguration itself cannot be simulated. Specifically, the delivery of a partial bitstream to a configuration port like the ICAP to see the resulting change (including intermediate states) in an RP.

ELF Association Within DFX

For hierarchical flows including DFX, you can instantiate the MicroBlaze™ IP into a hierarchical portion of a design. When you synthesize the hierarchical sub-design out-of-context (OOC), the MicroBlaze path information used for ELF association has a path that represents the sub-design and not the full design. This can cause `Memdata` or `Updatemem` ELF association failures when the sub-design merges into a larger design and the MicroBlaze path is no longer valid. Using DFX, the sub-design is a reconfigurable module (RM) containing a MicroBlaze core, and the larger top-level design is the static.

Associate the ELF file before the sub-design (RM) is added to the larger design (static) to avoid such failures. Do this at the end of synthesis. The following is an example of an association after OOC synthesis:

```
synth_design -top $top -part $part -mode out_of_context

add_files <application>.elf
set_property SCOPED_TO_REF {base_microblaze_design} [get_files <elf_file>]
set_property SCOPED_TO_CELLS {microblaze_0} [get_files <elf_file>]
refresh_meminit
```

AR76684 describes the same method and includes example archives.

Chapter 6

Design Considerations and Guidelines for 7 Series and Zynq Devices

This chapter explains design requirements that are unique to Dynamic Function eXchange (DFX), and are specific to 7 series and AMD Zynq™ 7000 SoC devices.

To take advantage of the Dynamic Function eXchange capability of AMD devices, you must analyze the design specification thoroughly, and consider the requirements, characteristics, and limitations associated with DFX designs. This simplifies both the design and debug processes, and avoids potential future risks of malfunction in the design.

Design Elements Inside Reconfigurable Modules

Not all logic is permitted to be actively reconfigured. Global logic and clocking resources must be placed in the static region to not only remain operational during reconfiguration, but to benefit from the initialization sequence that occurs at the end of a full device configuration.

Logic that can be placed in a reconfigurable module (RM) includes:

- All logic components that are mapped to a CLB slice in the device. This includes LUTs (look-up tables), FFs (flip-flops), SRLs (shift registers), RAMs, and ROMs.
- Block RAM and FIFO:
  - RAMB18E1, RAMB36E1, BRAM_SDP_MACRO, BRAM_SINGLE_MACRO, BRAM_TDP_MACRO
  - FIFO18E1, FIFO36E1, FIFO_DUALCLOCK_MACRO, FIFO_SYNC_MACRO

  Note: The IN_FIFO and OUT_FIFO design elements cannot be placed in an RM. These design elements must remain in static logic.
- DSP blocks: DSP48E1
- PCIe® (PCI Express®): Entered using PCIe IP

All other logic must remain in static logic and must not be placed in an RM, including:

- Clocks and Clock Modifying Logic - Includes BUFG, BUFR, MMCM, PLL, and similar components
- I/O and I/O related components (ISERDES, OSERDES, IDELAYCTRL, etc.)
- Serial transceivers (MGTs) and related components
- Individual architecture feature components (such as BSCAN, STARTUP, XADC, etc.)

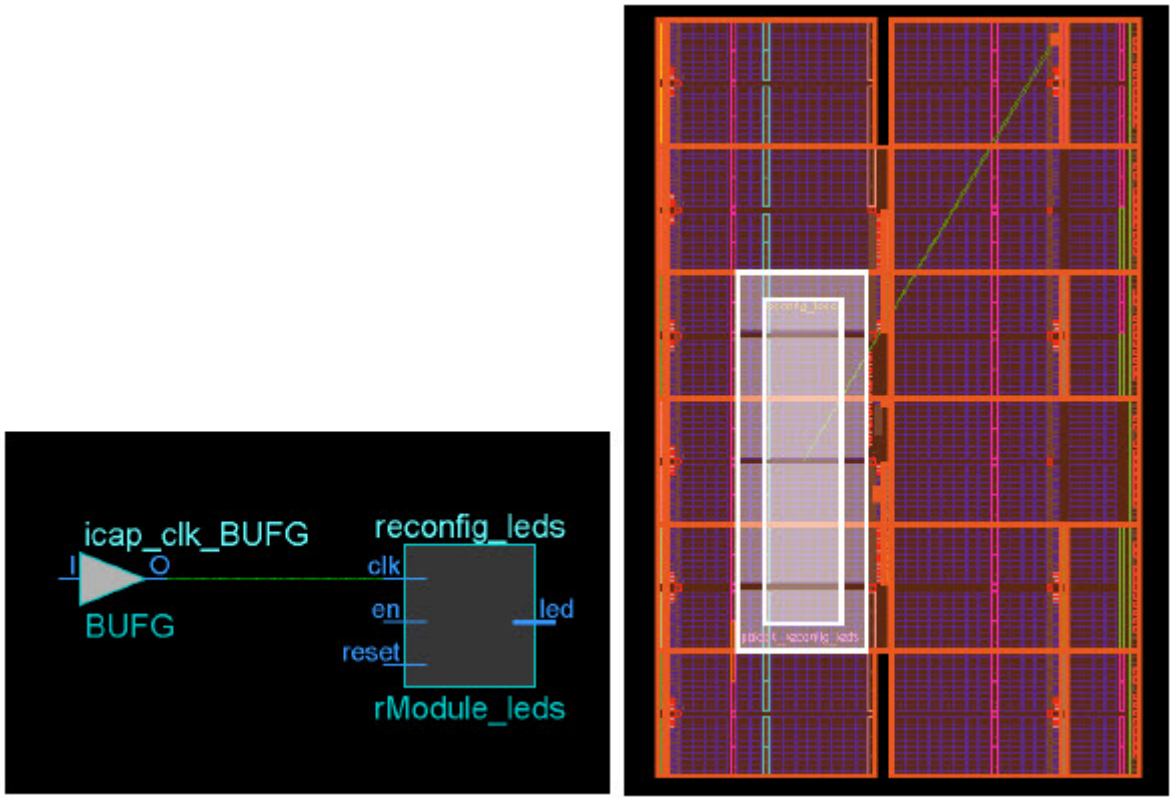
Global Clocking Rules

Because the clocking information for every RM for a particular Reconfigurable Partition (RP) is not known at the time of the first implementation, the DFX tools pre-route each BUFG output driving a partition pin on that RP to all clock regions that the Pblock encompasses. This means that clock spines in those clock regions might not be available for static logic to use, regardless of whether the RP has loads in that region.

In 7 series devices, up to 12 clock spines can be pre-routed into each clock region. This limit must account for both static and reconfigurable logic. For example, if 3 global clocks route to a clock region for static needs, any RP that covers that clock region can use the 9 global clocks available, collectively, in addition to those three top-level clocks.

In the example shown in the following figure, `icap_clk` is routed to clock regions X0Y1, X0Y2, and X0Y3 prior to placement, and static logic is able to use the other clock spines in that region.

Figure 72: **Pre-routing Global Clock to Reconfigurable Partition**



If there are a large number of global clocks driving an RP, create area groups that encompass complete clock regions to ease placement and routing of static logic. Global clocks can be downgraded to regional clocks (for example, BUFR, BUFH) for clocks with fewer loads or less demanding requirements. Shifting clocks from global to local resources allows for more flexibility in floorplanning when the RP requires many unique clocks.

Floorplanning for 7 Series Devices

As noted in Apply Reset After Reconfiguration, the height of the RP must align to clock region boundaries if `RESET_AFTER_RECONFIG` is to be used. Otherwise, any height can be selected for the RP.

The width of the RP must be set appropriately to make most efficient usage of interconnect and clocking resources. The left and right edges of Pblock rectangles should be placed between two resource columns (for example, CLB-CLB, CLB-block RAM or CLB-DSP) and not between two interconnect columns (INT-INT). This allows the placer and router tools the full use of all resources for both static and reconfigurable logic. Implementation tool DRCs provide guidance if this approach is not followed.

Related Information

Apply Reset After Reconfiguration

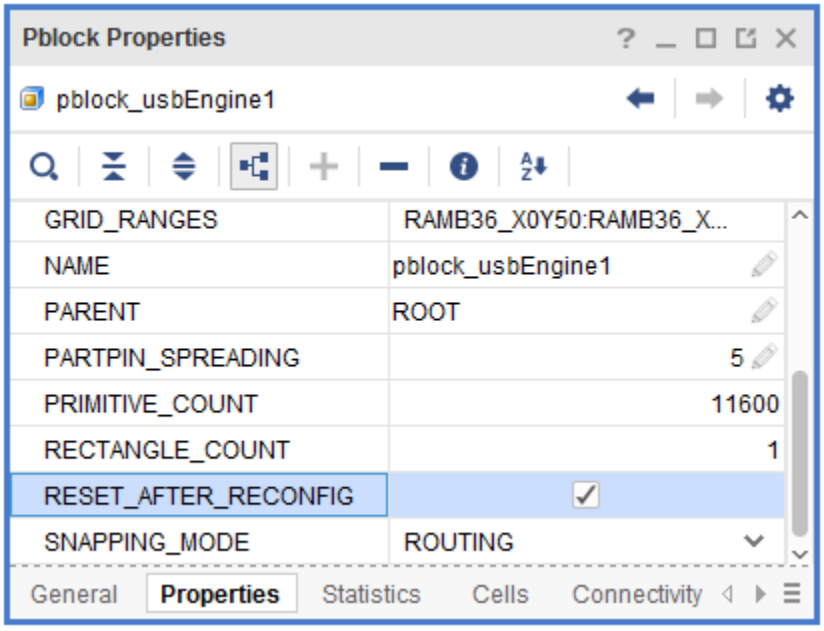
Automatic Adjustments for Reconfigurable Partition Pblocks

The Pblock `SNAPPING_MODE` property automatically resizes Pblocks to ensure no back-to-back violations occur for 7 series designs. When `SNAPPING_MODE` is set to a value of `ON` or `ROUTING`, it creates a new set of derived Pblock ranges that are used for implementation. The new ranges are stored in memory, and are not written out to the XDC. Only the `SNAPPING_MODE` property is written out, in addition to the normal Pblock constraints.

In 7 series devices the structure is such that the routing resources, called interconnect tiles, are placed adjacent, or back-to-back. When floorplanning for partial reconfiguration, it is important to understand where these back-to-back boundaries exist. If a Pblock splits these paired interconnect tiles, it is called a back-to-back violation. For more information on back-to-back interconnect please refer to Creating Reconfigurable Partition Pblocks Manually.

The original Pblock rectangle(s) are not modified when using `SNAPPING_MODE` and can still be resized, moved, or extended with additional rectangles. Whenever the original Pblock rectangle is modified, the derived ranges are automatically recalculated. The `SNAPPING_MODE` property is supported in batch mode, so there is no requirement to open the current Pblock in the Vivado IDE to set the `SNAPPING_MODE` value, although this option is available when performing interactive floorplanning, as shown in the following figure.

Figure 73: **Enabling the SNAPPING_MODE Property in the Vivado IDE**



When you set the SNAPPING_MODE property using the following syntax (or by selecting the Pblock Property as shown above), the implementation tools automatically see the corrected Pblock ranges.

```
set_property SNAPPING_MODE ON [get_pblocks <pblock_name>]
```

The following table shows SNAPPING_MODE property values for 7 series devices.

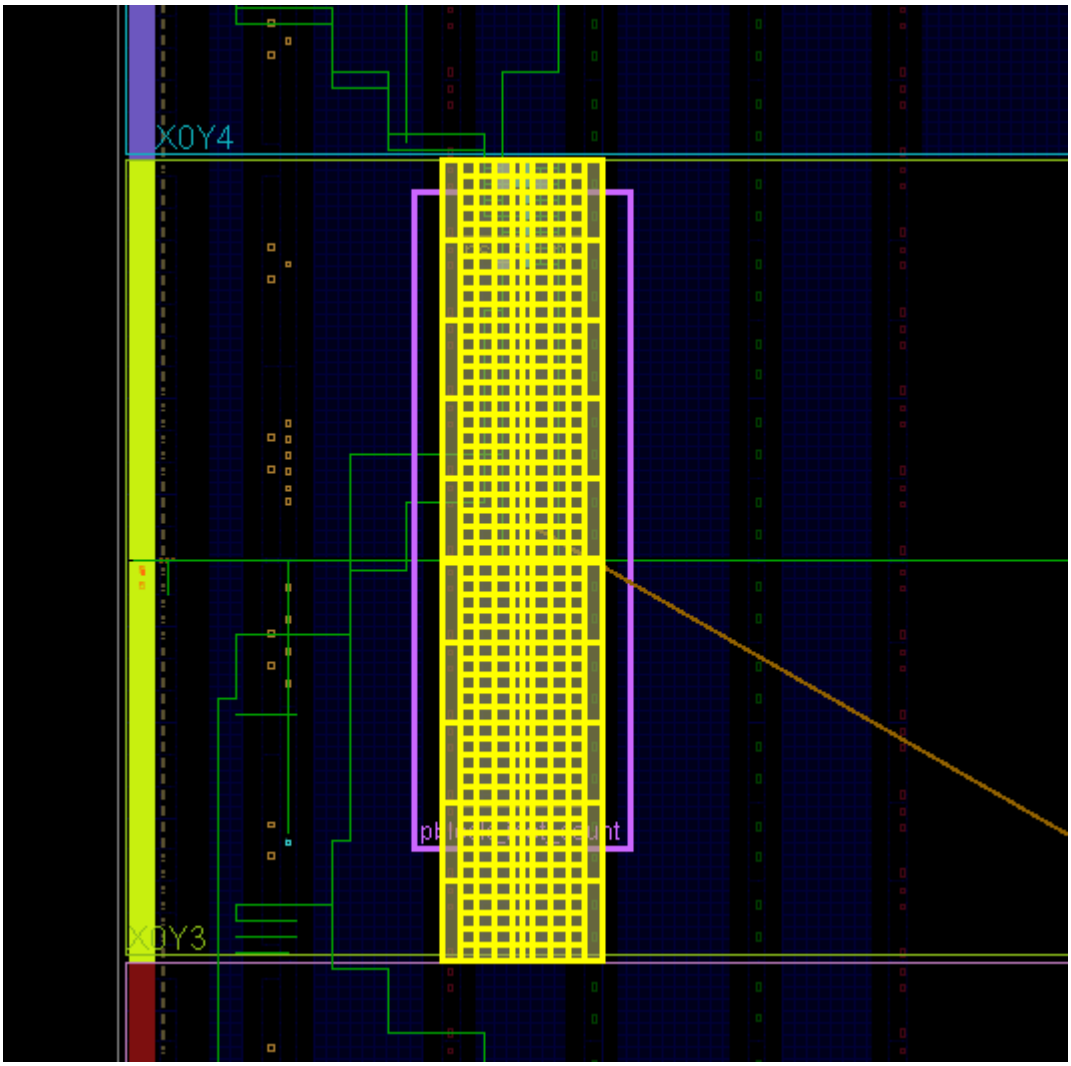
Table 12: **SNAPPING_MODE Property Values for 7 Series Devices**

Property	Value	Description
SNAPPING_MODE	OFF	Default for 7 series. No adjustments are made and DERIVED_RANGES == GRID_RANGES
	ON	Fixes all back-to-back violations.
	ROUTING	Same behavior as ON, except for the following exceptions: • Does not fix back-to-back violations across the center clock column to improve routing. • Grabs unbonded I/O and GT sites that are within or adjacent to the RP Pblock to improve routing. It can only use these resources for PR routing if the sites are unbonded and if the entire column (Clock Region in height) is included in the Pblock rectangle. This is the recommended value for 7 series and Zynq designs.

The SNAPPING_MODE property also works in conjunction with RESET_AFTER_RECONFIG. Using RESET_AFTER_RECONFIG requires Pblocks to be vertically frame (or clock region) aligned. When SNAPPING_MODE is set to ON or to ROUTING and RESET_AFTER_RECONFIG is set to TRUE, the derived ranges automatically include all sites necessary to meet this requirement.

The following figure shows the original user-created Pblock in purple. RESET_AFTER_RECONFIG has been enabled, and both left and right edges split interconnect columns. By applying SNAPPING_MODE, the resulting derived Pblock (shown in yellow) is *narrower* to avoid INT-INT boundaries, and *taller* to snap to the height of a clock region.

Figure 74: **Original and Derived Pblocks Using SNAPPING_MODE**



Related Information

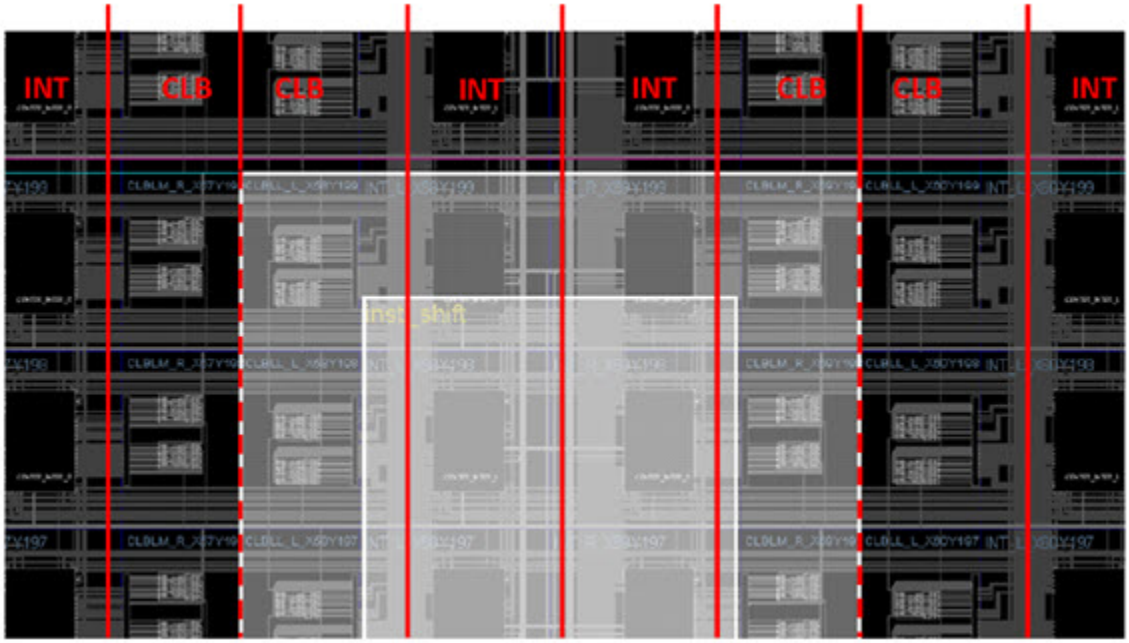
Creating Reconfigurable Partition Pblocks Manually

Creating Reconfigurable Partition Pblocks Manually

If automatic modification to the RP Pblock is not desired to fix back-to-back issues, you can create Pblock ranges manually to meet your needs. This is most useful when explicit control is needed for Pblocks that must span non-reconfigurable sites, such as configuration blocks or the center column, which contains clock buffer resources.

In the following figure the left and right edges are drawn between CLB columns for the Pblock highlighted in white. Visualization of the interconnect tiles as shown in this image requires that the routing resources are turned on, using this symbol in the Device View .

Figure 75: **Optimal - Reconfigurable Partition Pblock Splitting CLB-CLB on Both Left and Right Edges**



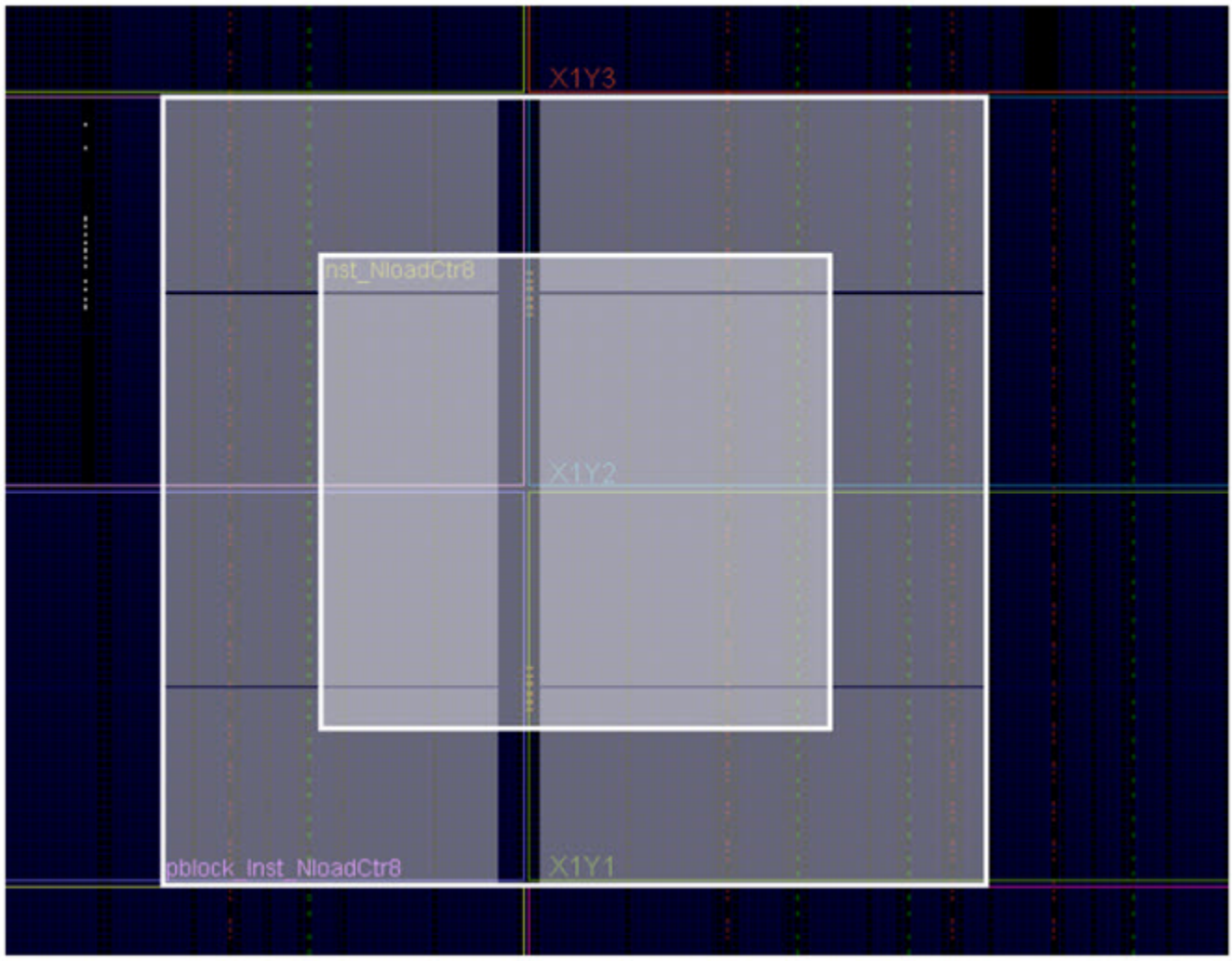
The RP Pblock must include all reconfigurable element types within the shape drawn. In other words, if the rectangle selected encompasses CLB (Slice), block RAM, and DSP elements, all three types must be included in the Pblock constraints. If one of these is omitted, a DRC is triggered with an alert that a split interconnect situation has been detected.

Other considerations must be taken if the RP spans non-reconfigurable sites, such as the center-column clocking resources or configuration components (ICAP, BSCAN, etc.), or abuts non-reconfigurable components such as I/O. If a Pblock edge splits interconnect columns for different resource types, implementation tools accept this layout, but restrict placement in the columns on each side of the boundary. If this prohibits sites that are needed for the design (such as the ICAP or BSCAN, for example), the Pblock must be broken into multiple rectangles to clearly define reconfigurable logic usage, or `SNAPPING_MODE` must be used.

The implementation tools automatically prevent placement on both sides of the back-to-back interconnect by creating `PROHIBIT` constraints. If the sites that are prohibited due to a back-to-back violation are not needed in the design, it is acceptable to leave the back-to-back violation in the design. Doing so allows an extra column of routing tiles to be included in the dynamic region, and can reduce congestion in a dynamic region that spans non-reconfigurable sites. In this case, a Critical Warning is issued by DRCs, but the warning can be safely ignored if you understand the trade-offs of placement versus routing resources.

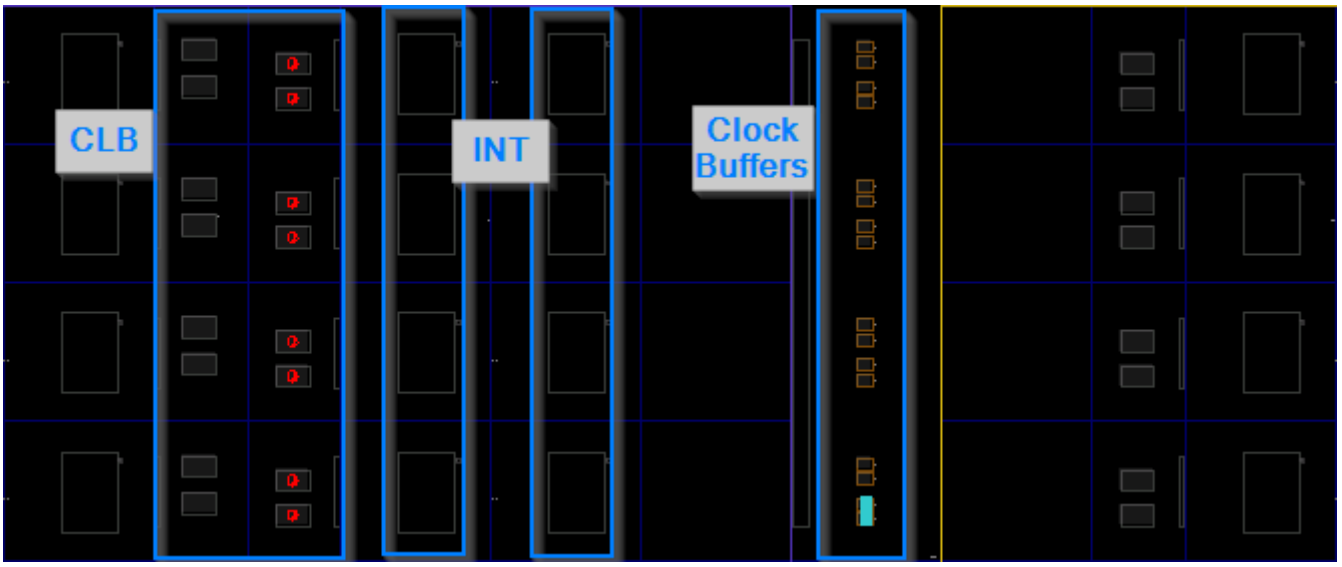
The one exception to this behavior is around the clock column. If a violation occurs at the clock column boundary, `PROHIBIT` constraints are generated for the RM side of the violation (typically SLICE prohibits), but the clocking resources do not get prohibit constraints and are still available to the static logic. The `SNAPPING_MODE` property has a value of `ROUTING`, which takes advantage of this special exception. For example, the initial floorplan shown in Figure 75 spans the center column, which contains clock buffer resources (BUFHCE/BUFGCTRL). These resources have not been included in the Pblock, as they are not highlighted in the following figure. There is violation caused by spanning this clock column but the resources can still be used by the static logic.

Figure 76: **Pblock Spanning Non-Reconfigurable Sites**



Prohibited sites appear in placed or routed checkpoints as sites with a red circle with a slash, as shown in the following figure. With this automatic prohibit feature, the routing interconnect associated with reconfigurable sites (CLBs) can still be used for the RM even though the CLBs themselves are not used. In the following figure, the column of INT on the left is available for the RM, but the column of INT on the right is only available for static logic because these are part of the clock tile, which is not reconfigurable for 7 series devices.

Figure 77: **Prohibited Sites in a Checkpoint**



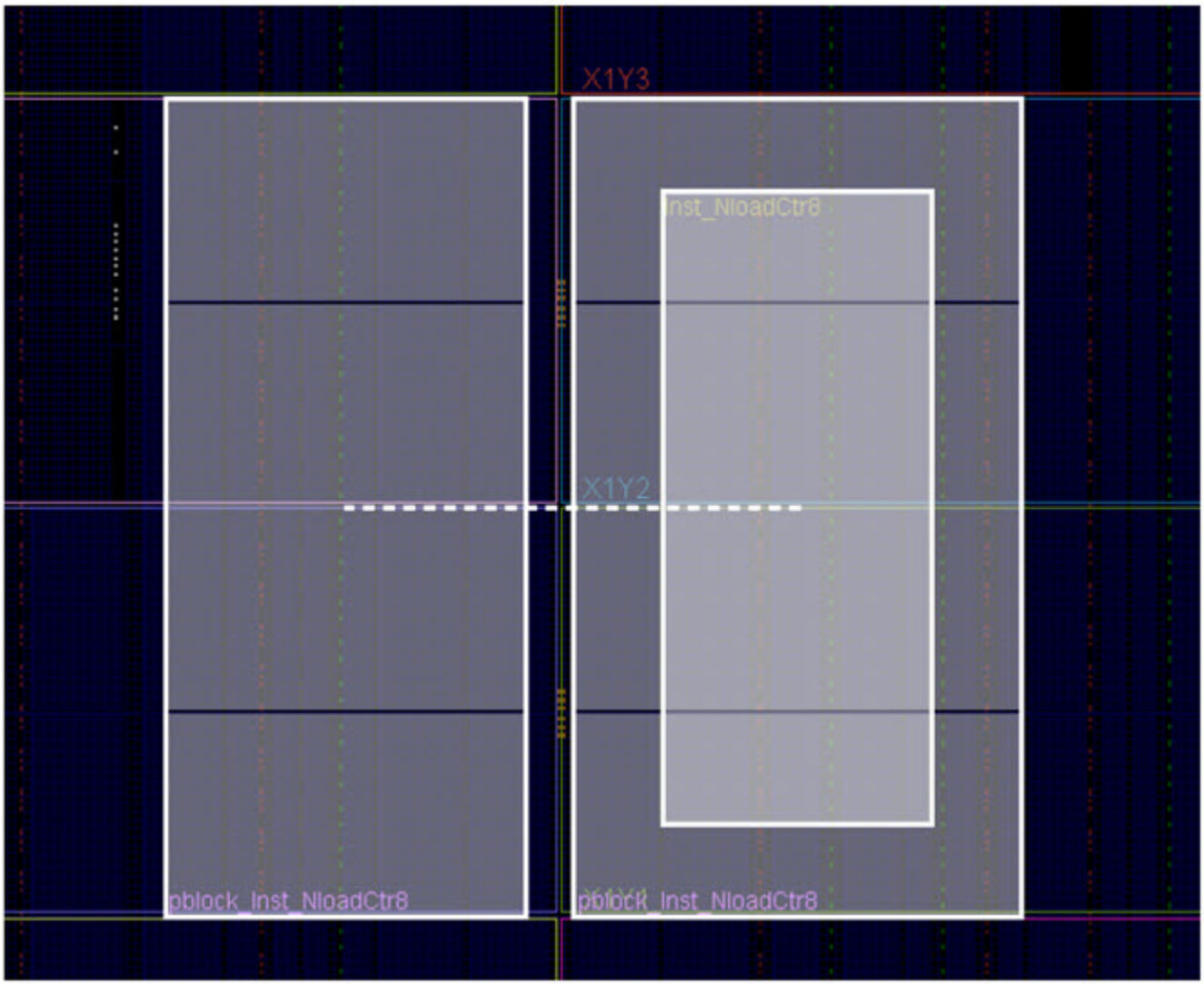
If a back-to-back violation prohibits sites that are needed for the design (that is, ICAP or BSCAN sites), a placement error is issued, stating that not enough sites are available in the device.

```
ERROR: [Common 17-69] Command failed: Placer could not place all instances
```

To avoid this restriction, create multiple Pblock rectangles that avoid splitting interconnect columns, as shown in the following figure, or use the Pblock `SNAPPING_MODE` property.

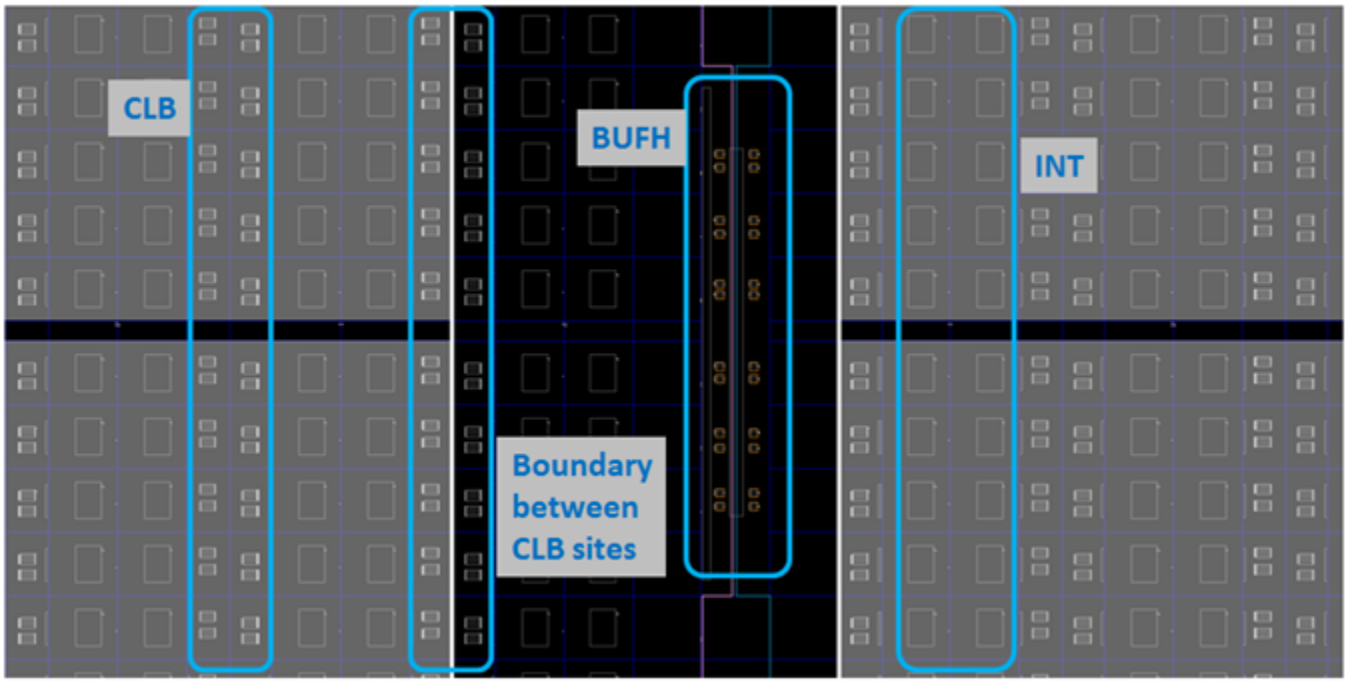
In general, spanning non-reconfigurable site types (such as IOB, configuration, or clocking columns) should be avoided whenever possible. If the Pblock must span one of these, the clocking column is the least risky choice, owing to its special nature (described previously). Use `SNAPPING_MODE ROUTING` to cross this boundary as efficiently as possible.

Figure 78: **Multiple Pblock Rectangles that Avoid Non-Reconfigurable Resources**



The following figure is a close-up of this split, showing Slice (CLB) and Interconnect (INT) resource types. The gap between the two Pblock rectangles gives full access to the BUFHCE components to route completely using static resources. This also leaves one column of CLBs available for the static design to use. Although routing resources exist that can cross these gaps, the overall routability of such structures is notably reduced. This approach is more challenging and should be avoided if possible. When spanning other static boundaries, such as IOB or configuration tiles, the routing gap for the dynamic region becomes two INT resources, and routing becomes difficult.

Figure 79: **Close-up Showing Columns Reserved for Clock Routing Usage**



Irregular shaped Partitions (such as a T or L shapes) are permitted, but you are encouraged to keep overall shapes a simple as possible. Placement and routing in such regions can become challenging because routing resources must be entirely contained within these regions. Boundaries of Partitions can touch, but this is not recommended, as some separation helps mitigate potential routing restriction issues. Nested or overlapping RPs (partitions within partitions) are not permitted.

Finally, only one RP can exist per physical Reconfigurable Frame. A Reconfigurable Frame is the smallest size physical region that can be reconfigured, and aligns with clock region boundaries. A Reconfigurable Frame cannot contain logic from more than one RP. If it were to contain logic from more than one RP, it would be very easy to reconfigure the region with information from an incorrect RM, thus creating contention. The Vivado tools are designed to avoid that potentially dangerous occurrence.

Using High Speed Transceivers for 7 Series Devices

AMD high speed transceivers (GTP, GTX, GTH, GTZ) are not reconfigurable in 7 series devices, and must remain in static logic. However, settings for the transceivers can be updated during operation using the DRP ports. For more information on the transceiver settings and DRP access, see *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476), or *7 Series FPGAs GTP Transceivers User Guide* (UG482).

Dynamic Function eXchange Design Checklist (7 Series)

AMD highly encourages the following items for a 7 series FPGA design using Dynamic Function eXchange:

Recommended Clocking Networks

Are you using global clock buffers, regional clock buffers, or clock modifying blocks (MMCM, PLL)?

These blocks must be in static logic.

See Design Elements Inside Reconfigurable Modules for more information, and Global Clocking Rules for complete details on global clock implementation.

Configuration Feature Blocks

Are you using device feature blocks (BSCAN, CAPTURE, DCIRESET, FRAME_ECC, ICAP, STARTUP, USR_ACCESS)?

These featured blocks must be in static logic.

See Design Elements Inside Reconfigurable Modules for more information.

High Speed Transceiver Blocks

Do you have high speed transceivers in your design?

High speed transceivers must remain in the static partition.

See Using High Speed Transceivers for 7 Series Devices for specific requirements.

System Generator DSP Cores, HLS Cores, or IP Integrator Block Diagrams

Are you using System Generator DSP cores, HLS cores, or IP integrator block diagrams in your Dynamic Function eXchange design?

Any type of source can be used as long as it follows the fundamental requirements for Dynamic Function eXchange. Any code processed by System Generator, HLS, or Vivado IP integrator (or other tools) is eventually synthesized. The resulting design checkpoint or netlist must be made up entirely of reconfigurable elements (CLB, block RAM, DSP) for it to be legally included in an RP.

Packing I/Os into Reconfigurable Partitions

Do you have I/Os in RMs?

All I/Os must reside in static logic.

See Design Elements Inside Reconfigurable Modules for more information.

Packing Logic into Reconfigurable Partitions

Is all logic that must be packed together in the same RP?

Any logic that must be packed together must be in the same RP and RM.

See Packing Logic for more information.

Packing Critical Paths into Reconfigurable Partitions

Are critical paths contained within the same partition?

RP boundaries limit some optimization and packing, so critical paths should be contained within the same partition.

See Packing Logic for more information.

Floorplanning

Can your RPs be floorplanned efficiently?

See Creating Pblocks for 7 Series Devices for more information.

Recommended Decoupling Logic

Have you created decoupling logic on the outputs of your RMs?

During reconfiguration the outputs of RPs are in an indeterminate state, so decoupling logic must be used to prevent static data corruption.

See Decoupling Functionality for more information.

Recommended Reset after Reconfiguration

Are you resetting the logic in an RM after reconfiguration?

After reconfiguration, new logic might not start at its initial value. If the Reset After Reconfiguration property is not used, a local reset must be used to ensure it comes up as expected when decoupling is released. Clock and other inputs to the RP can also be disabled during reconfiguration to prevent initialization issues. Alternatively, the Reset After Reconfiguration property can be applied. This option holds internal signals steady during reconfiguration, then issues a masked global reset to the reconfigured logic.

See Apply Reset After Reconfiguration for more information.

Debugging with Logic Analyzer Blocks

Are you using the Vivado Logic Analyzer with your Dynamic Function eXchange design?

Vivado Logic Analyzer (ILA/VIO debug cores) can be used in your Dynamic Function eXchange design, but care must be taken when connecting these cores to debug hubs. Use the automatic inference solution shown in Using Vivado Debug Cores.

Efficient Reconfigurable Partition Pblocks

Have you created efficient RP Pblock(s) for your design?

The height of the RP Pblock must align with the top and bottom of a clock region boundary, if the `RESET_AFTER_RECONFIG` property is to be used. Otherwise, any height can be selected for the RP Pblock.

See Creating Pblocks for 7 series Devices for more information.

Validating Configurations

How do you validate consistency between configurations?

The `pr_verify` command is used to make sure all configurations have matching imported resources.

See Verifying Configurations for more information.

Configuration Requirements

Are you aware of the particular configuration requirements for Dynamic Function eXchange for your design and device?

Each device family has specific configuration requirements and considerations.

See Configuring the Device for more information.

Effective Pblock Recommendations

Does an RP Pblock extend over the center clock column or the configuration column in the device?

Due to the back-to-back INT tile requirement for 7 series devices, coupled with the `CONTAIN_ROUTING` requirement, extending a Pblock over these specialized blocks in the device can make routing very difficult or impossible. Avoid extending an RP Pblock across these areas whenever possible.

See Automatic Adjustments for Reconfigurable Partition Pblocks and Creating Reconfigurable Partition Pblocks Manually for more information on back-to-back requirements.

Related Information

Design Elements Inside Reconfigurable Modules
Global Clocking Rules
Using High Speed Transceivers for UltraScale and UltraScale+ Devices
Packing Logic
Decoupling Functionality
Floorplanning for 7 Series Devices
Apply Reset After Reconfiguration
Using Vivado Debug Cores
Verifying Configurations
Configuring the Device
Automatic Adjustments for Reconfigurable Partition Pblocks
Creating Reconfigurable Partition Pblocks Manually

Chapter 7

Design Considerations and Guidelines for UltraScale and UltraScale+ Devices

This chapter explains design requirements that are unique to Dynamic Function eXchange (DFX), and are specific to AMD UltraScale™ and AMD UltraScale+™ devices.

To take advantage of the Dynamic Function eXchange capability of AMD devices, you must analyze the design specification thoroughly, and consider the requirements, characteristics, and limitations associated with DFX designs. This simplifies both the design and debug processes, and avoids potential future risks of malfunction in the design.

Design Elements Inside Reconfigurable Modules

In UltraScale and UltraScale+ devices, nearly all component types can be partially reconfigured.

Logic that can be placed in a reconfigurable module includes:

- All logic components that are mapped to a CLB slice in the FPGA. This includes LUTs (look-up tables), FFs (flip-flops), SRLs (shift registers), RAMs, and ROMs.
- Block RAM and FIFO: RAMB18E2, RAMB36E2, FIFO18E2, FIFO36E2
- DSP blocks: DSP48E2
- PCIe® (PCI Express), CMAC (100G MAC), and ILKN (Interlaken MAC) blocks
- UltraRAM blocks: URAM288
- SYSMON (XADC and System Monitor)
- Clocks and Clock Modifying Logic: Includes BUFG, BUFGCE, BUFGMUX, MMCM, PLL, and similar components
- I/O and I/O related components (ISERDES, OSERDES, IDELAYCTRL, etc.)
- Serial transceivers (MGTs) and related components

- HSADC blocks in Zynq RFSoC devices for RF-ADC and RF-DAC data conversion

  ***Note*:** DNA_PORT -- the Device DNA Access Port (DNA_PORTE2) is the only configuration element in the CONFIG_SITE that is reconfigurable. Any other use of CONFIG_SITE elements is not permitted if the DNA_PORT is to be reconfigurable.

Only configuration components must remain in the static part of the design. These components are:

- AXI32
- BSCAN
- DCIRESET
- EFUSE_USR
- FRAME_ECC
- ICAP
- MASTER_JTAG
- STARTUP
- USR_ACCESS

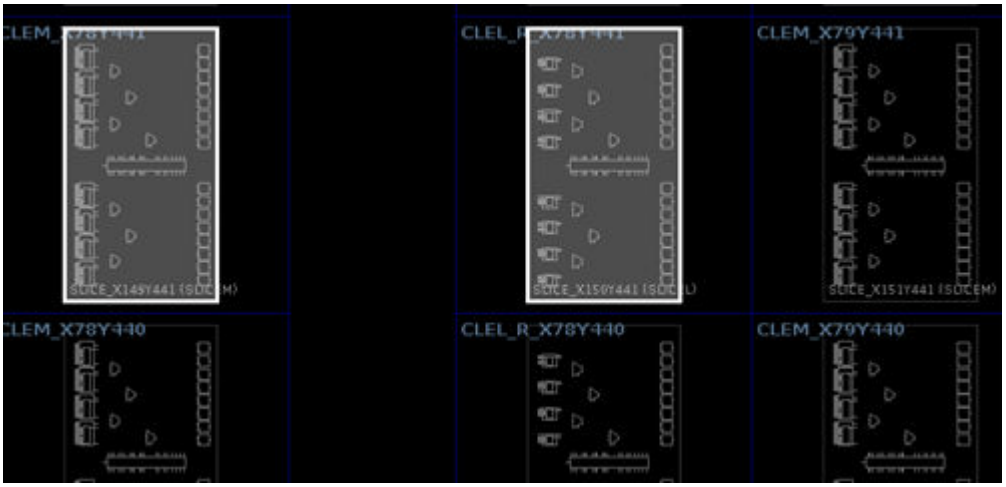
Floorplanning for UltraScale and UltraScale+ Devices

In the UltraScale architecture, the smallest unit that can be reconfigured is smaller than in previous architectures. The minimum required resources for reconfiguration varies based on the resource type and are referred to as a programmable unit (PU). Because adjacent sites share a routing resource (or interconnect tile) in UltraScale devices, a PU is defined in terms of pairs.

Following are examples of some of the minimum PU that can be reconfigured based on the site types:

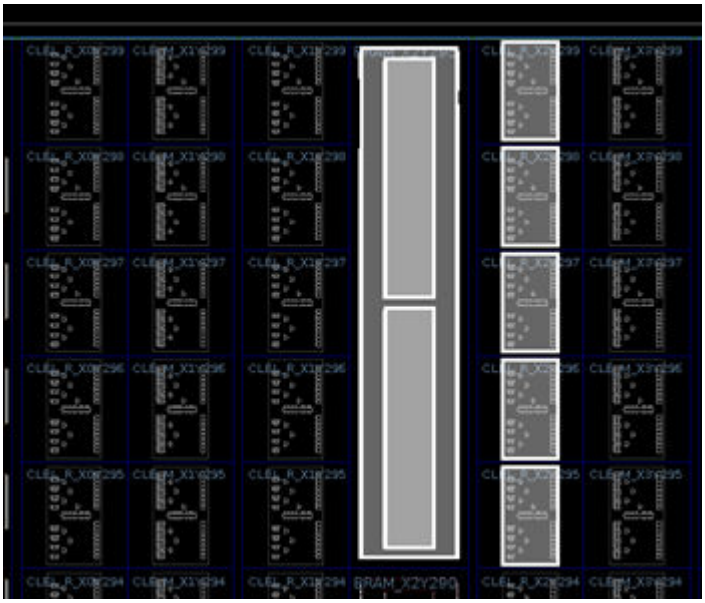
- **CLB PU:** Includes two adjacent CLBs and the shared interconnect.

Figure 80: **CLB PU**



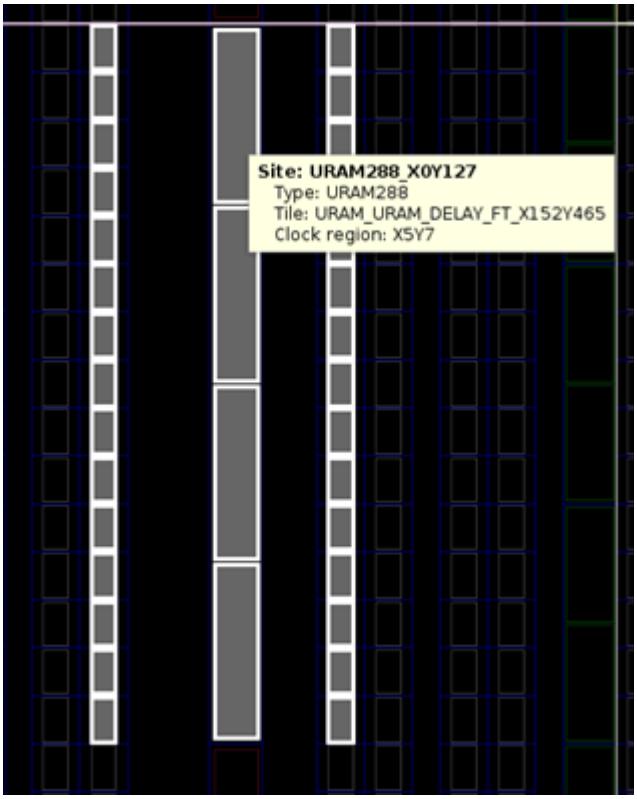
- **Block RAM PU:** Includes one RAMBFIFO36, two RAMB18, five adjacent CLBs, and the shared interconnect. The PU is the corresponding block RAM tile. One block RAM tile includes RAMBFIFO18, FIFO18_0, RAMB180, RAMB181, RAMBFIFO36, FIFO36, and RAMB36. Adjacent CLE sites are part of the block RAM PU.

Figure 81: **Block RAM PU**



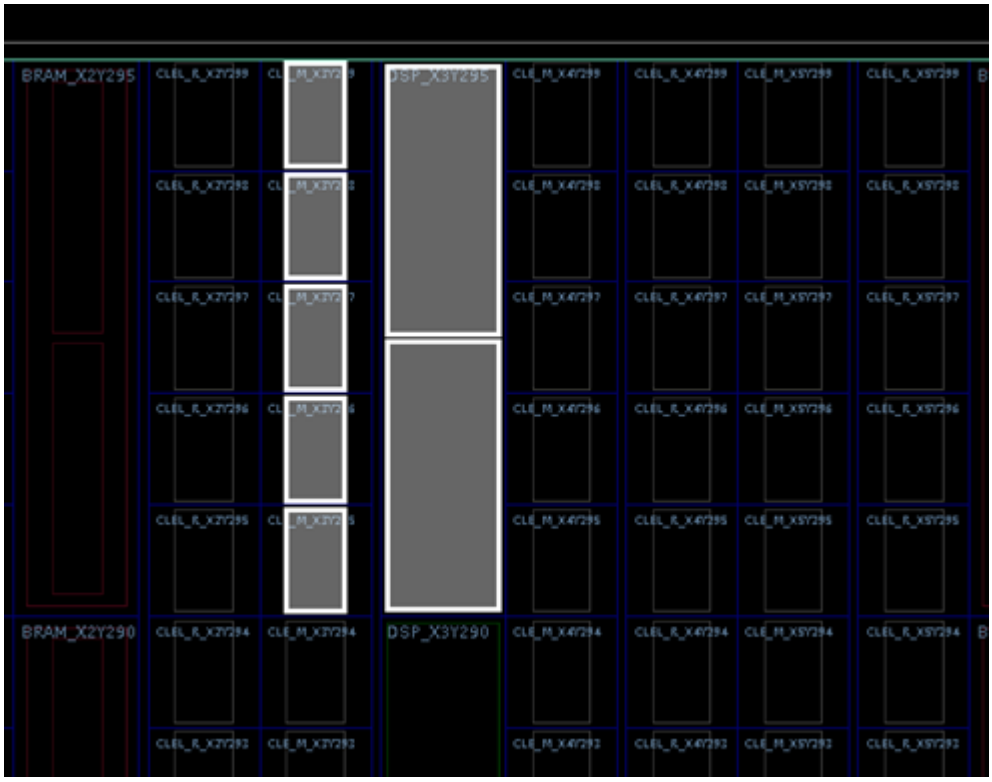
- **URAM PU:** Includes one URAM, 30 adjacent CLBs (15 on each side), and the shared interconnect.

Figure 82: **URAM PU**



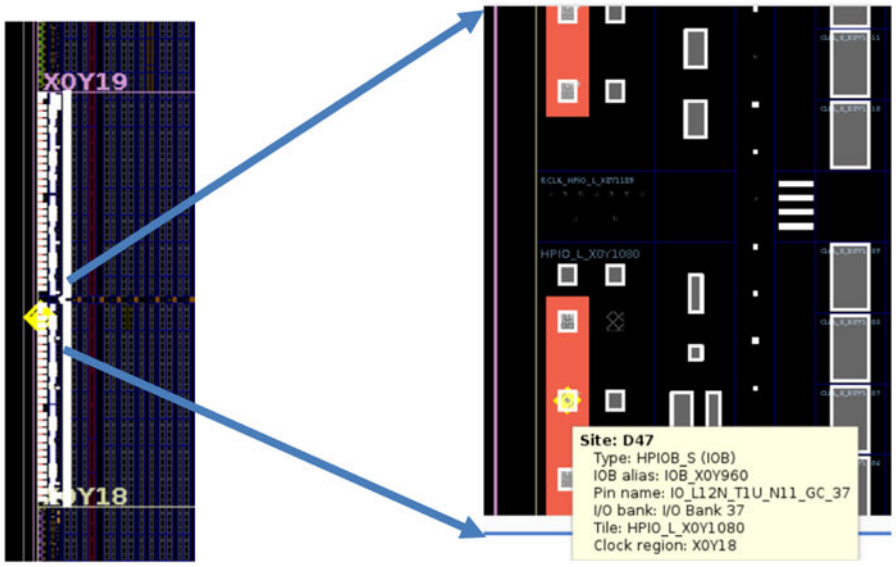
- **DSP PU:** Includes one DSP, the five adjacent CLBs, and the shared interconnect. The PU is the corresponding DSP tile. One DSP tile includes two DSP sites.

Figure 83: **DSP PU**



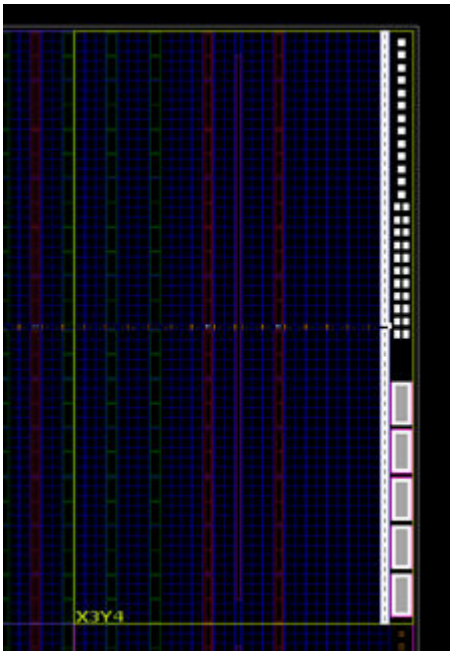
- **IO PU:** The I/O of the full height of the clock_region includes BITSLICE_CONTROL, BITSLICE_RX_TX, BITSLICE_TX, BUFGCE, BUFGCE_DIV, BUFGCTRL, IOB, MMCME3_ADV, PLLE3_ADV, PLL_SELECT_SITE, RIU_OR, HBM_REFCLK etc., the adjacent 60 CLBs and the shared interconnect.

Figure 84: **IO PU**



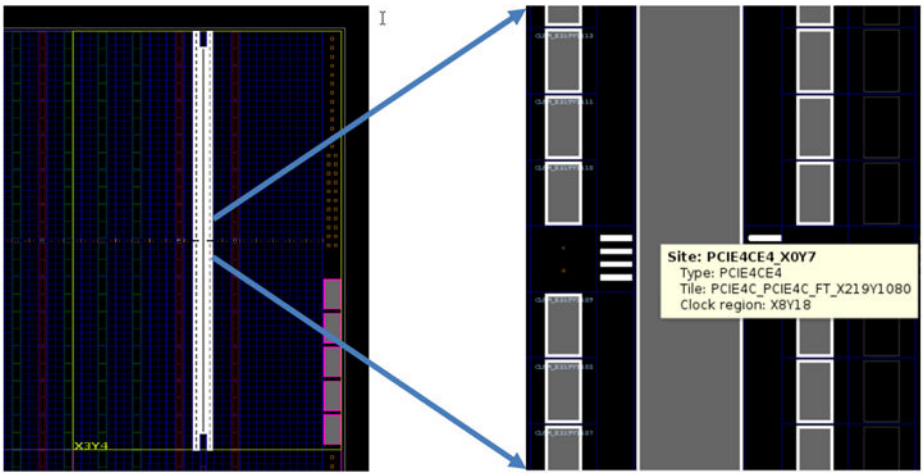
- **GTY PU:** Includes a full GT quad (four GT_CHANNEL and one GT_COMMON), the adjacent 60 CLBs, 24 BUFG_GTs, 15 BUFG_GT_SYNCs, and the shared interconnect. The PU is the corresponding GTY_R_X147Y20 tile. Sites included in the tile are BUFG_GT, BUFG_GT_SYNC, GTYE4_CHANNEL, and GTYE4_COMMON. The adjacent INTF_GT tiles are automatically pulled into the routing footprint of the reconfigurable Pblock.

Figure 85: **GTY PU**



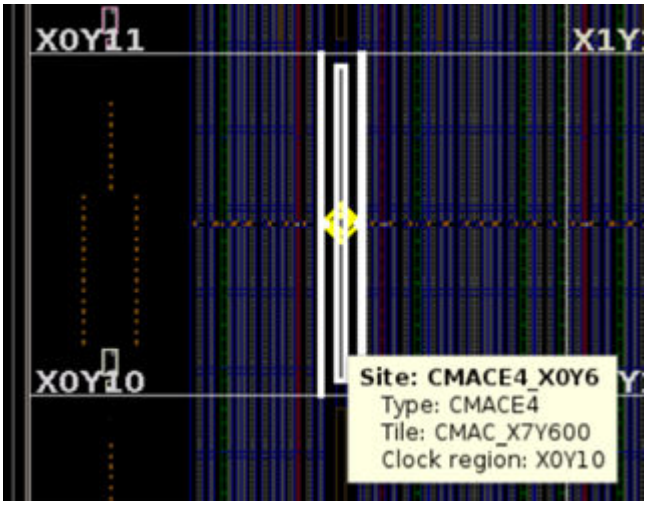
- **GTM_DUAL PU:** Includes GTM_DUAL, GTM_REFCLK, 24 BUFG_GTs, 15 BUFG_GT_SYNCs, 60 CLB tiles and shared interconnect.
- **PCIe® PU:** Includes one PCIE40E4 or PCIE4CE4, 120 adjacent CLBs (60 on each side) and the shared interconnect. The PU is the corresponding PCIe tile. It includes PCIE40E4.

Figure 86: **PCIe PU**



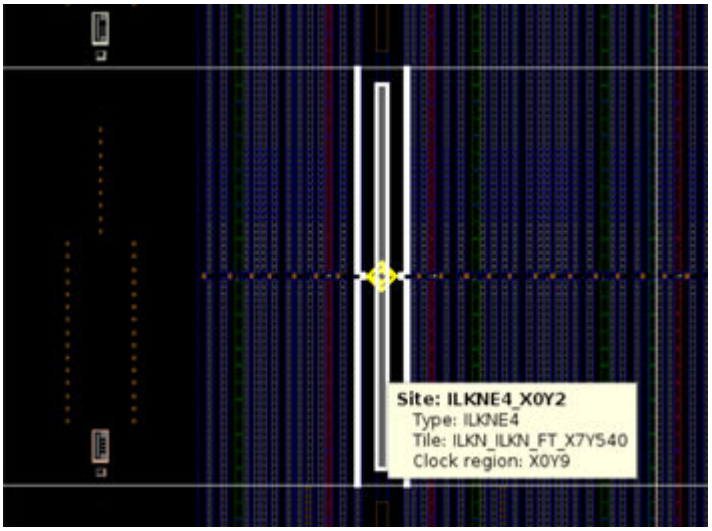
- **CMAC PU:** Includes one CMACE4, 120 adjacent CLBs (60 on each side) and the shared interconnect.

Figure 87: **CMAC PU**



- **Interlaken PU:** Includes one ILKNE4, 120 adjacent CLBs (60 on each side) and the shared interconnect.

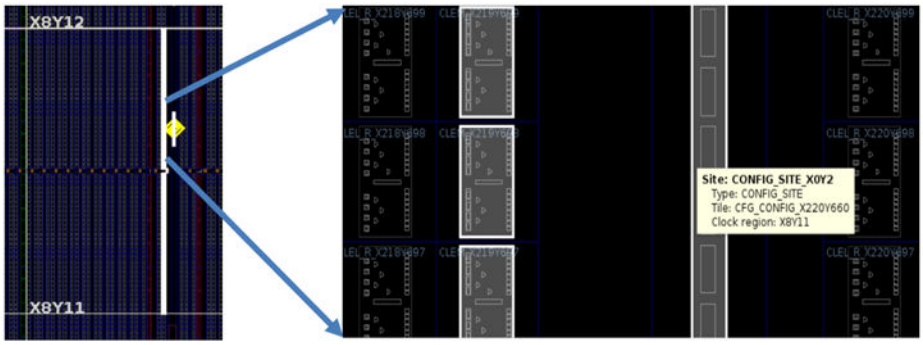
Figure 88: **Interlaken PU**



- **CONFIG PU:** Includes one CONFIG_SITE, 120 adjacent CLBs (60 on each side) and the shared interconnect.

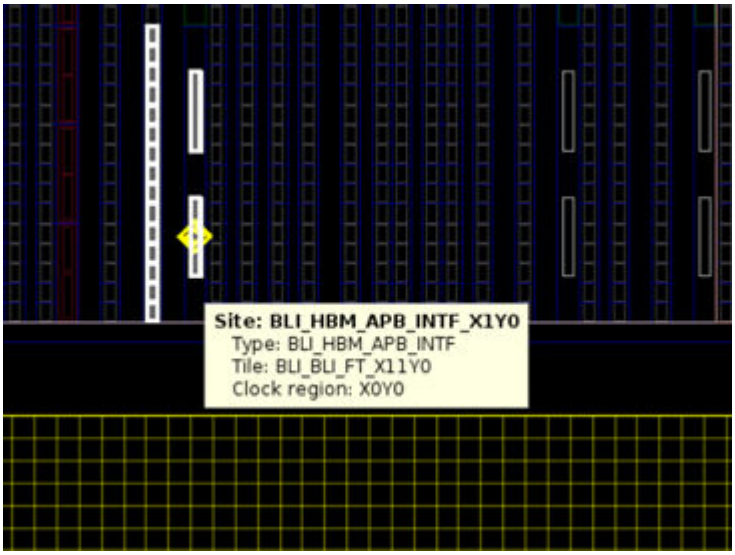
  ***Note*:** CONFIG_SITE elements cannot be reconfigured. However, you can add the CONFIG_SITE to a reconfigurable partition to get access to the CLBs. The CONFIG_SITE contains many single site resources including ICAP, STARTUP, BSCAN, FRAME_ECC, DNA_PORT, EFUSE_USR and MASTER_JTAG, and cannot be broken up further.

Figure 89: **CONFIG PU**



- **HBM BLI PU:** Includes one BLI_HBM, 15 adjacent CLBs and the shared interconnect.

Figure 90: **HBM BLI PU**

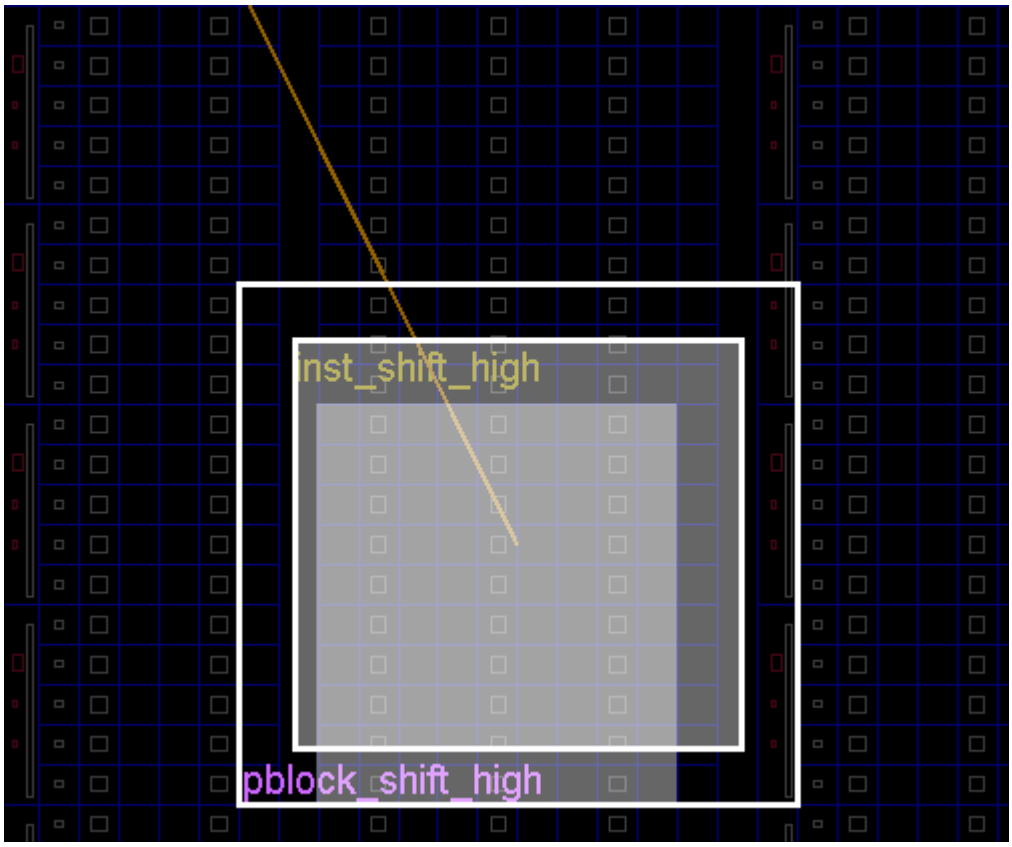


Automatic Adjustments for PU on Pblocks

In UltraScale and UltraScale+ devices, there is no `RESET_AFTER_RECONFIG` option. Instead, GSR is always issued at the end of a partial reconfiguration, and there are no Pblock size/shape requirements to enable this like there are in 7 series devices. However, to ensure that the Pblock does not violate any rules for minimum PU sizes, the `SNAPPING_MODE` property is also always on by default, and automatically adjusts the Pblock to make sure it is valid for DFX.

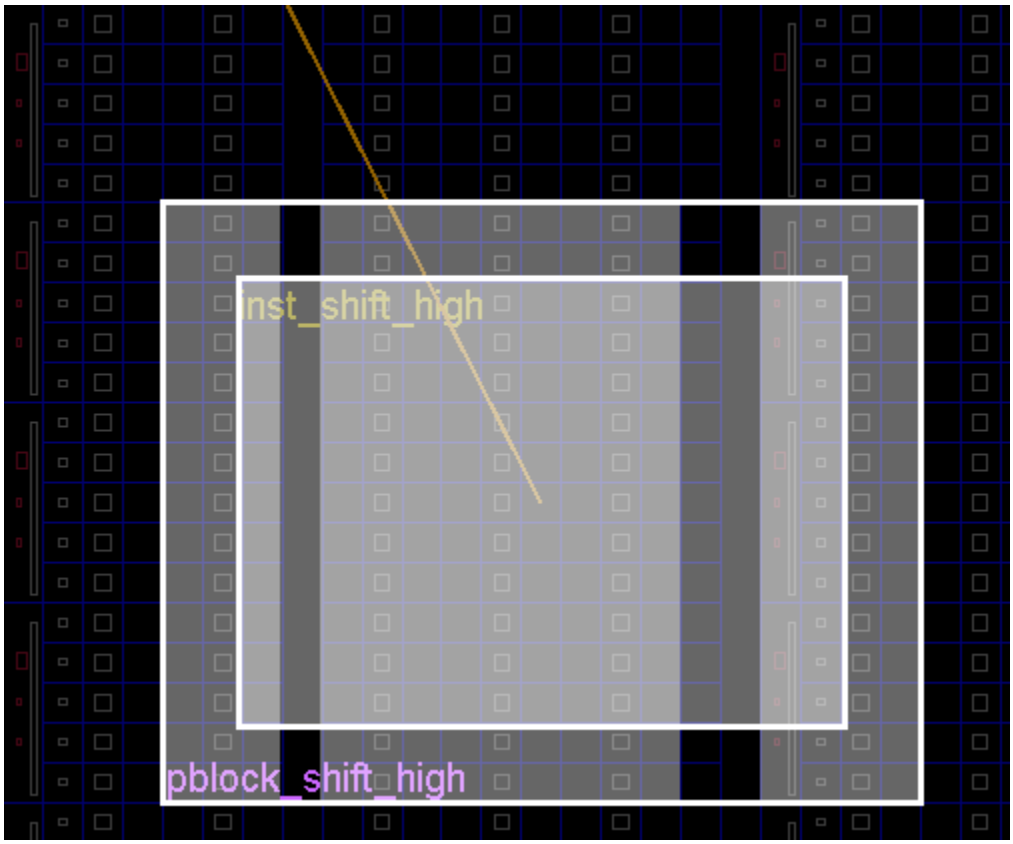
The following figures provide an example of how `SNAPPING_MODE` adjusts the Pblock for PU alignment. Despite the larger outer rectangle, only the selected tiles belong to the RP Pblock. The upper block RAM and DSP sites are not included because they are not fully contained in the Pblock, and the associated CLB sites are not included either, based on the PU rules. There are also CLB sites on both the left and right edge that are not included in the Pblock because the adjacent CLBs are not owned by the original rectangle.

Figure 91: **SNAPPING_MODE Example - UltraScale**



While `SNAPPING_MODE` made the above Pblock legal for the RP, it is possible that the intent was to include all of these sites. By making a small adjustment to the original Pblock rectangle, you can prevent `SNAPPING_MODE` from removing sites that are intended for the dynamic region. The Pblock has been expanded by one CLB on the left, right, and top edges. The shaded tiles that are owned by the RP Pblock now match the outer rectangle.

Figure 92: **PU Aligned Pblock**



While shading shows what is included in a reconfigurable partition (RP), you can best visualize the sites owned by an RP by using the `get_dfx_footprint` utility. The following steps can be used for debugging/verifying Pblocks:

1. Create or make an adjustment to an RP. The cell assigned to the Pblock must have the `HD.RECONFIGURABLE` property set.
2. Use the `get_dfx_footprint` utility to see the placement tiles for a given pblock.

```
highlight_objects -color <color> [get_dfx_footprint -place -of_objects
[get_cells <RP_cell>]]
```

Note:

The `get_dfx_footprint` Tcl utility is available for UltraScale and UltraScale+ devices and is a much more powerful and efficient visualization tool compared to hd_visual scripts. For more information on `get_dfx_footprint`, see Floorplanning Visualization. Versal has expanded capabilities for DFX as compared to UltraScale and UltraScale+, so the options that are available to UltraScale and UltraScale+ devices are limited to the following:

`-place`: placement footprint of a reconfigurable Pblock

`-route`: routing footprint of a reconfigurable Pblock

`-illegal_nodes`: set of nodes that cannot be used for an RP

Related Information

Floorplanning for Versal Devices

Sharing Configuration Frames between RP and Static Logic

Even though UltraScale and UltraScale+ devices Pblocks are not required to be frame aligned (that is, the height of the clock region), Dynamic Function eXchange still programs the entire configuration frame. This means that logic outside of the RP is overwritten. This does not cause any issues in DFX, but in previous architectures there were some limitations about what kind of static logic could be in the same frame as reconfigurable logic.

For UltraScale and UltraScale+ devices, any static logic can be placed in the same configuration frame as the RM, in any sites not owned by the RP Pblock. This includes block RAM, DSP, and LUT RAM. There is still a restriction however, that there can be only one RP per configuration frame. That means you cannot vertically stack two RPs in the same clock region.

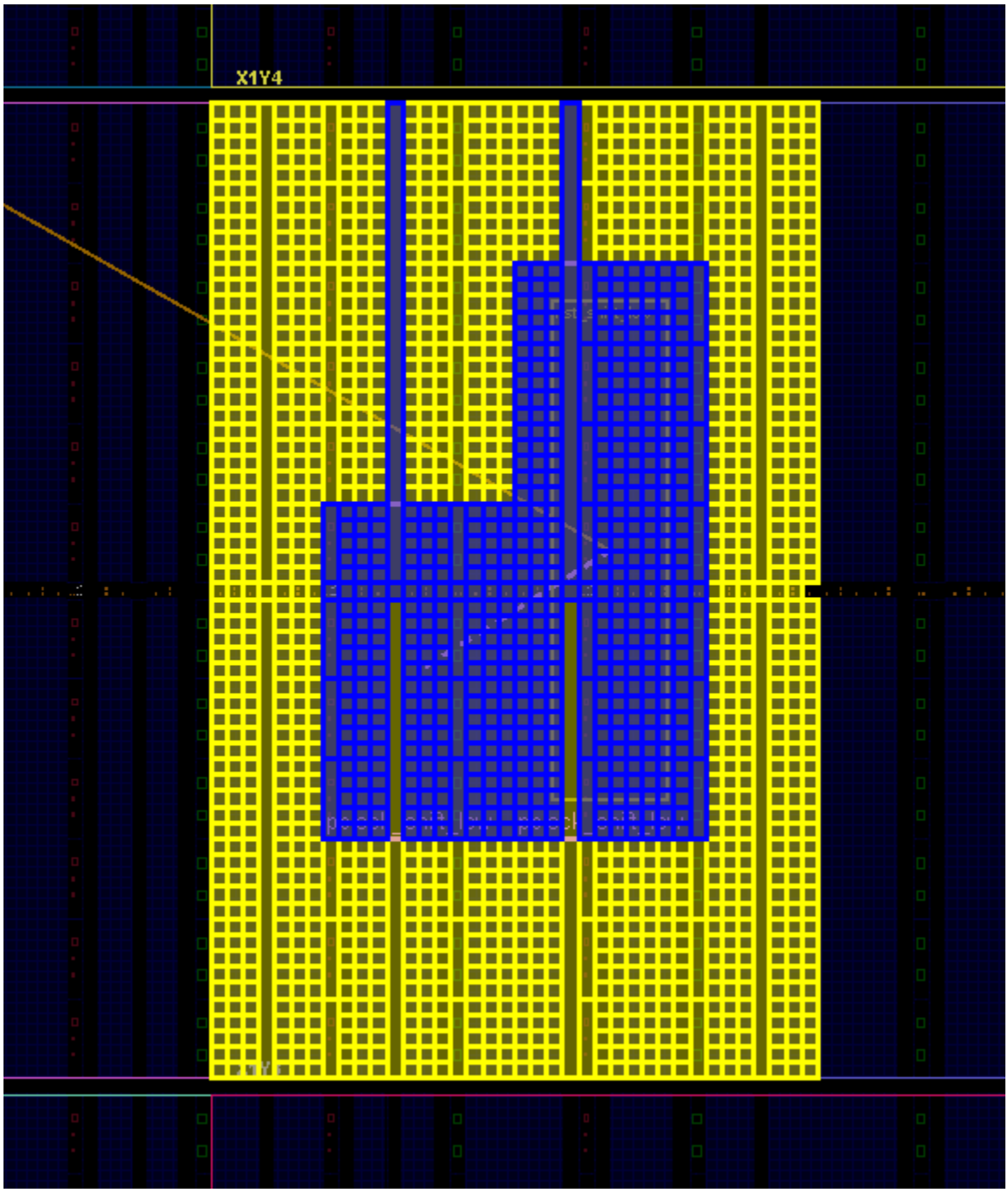
Expansion of CONTAIN_ROUTING Area

The contained routing requirement of RP Pblocks for UltraScale and UltraScale+ devices has been relaxed to allow for improved routing and timing results. Instead of routing being confined strictly to the resources owned by the Pblock, the routing footprint is expanded. This includes resources that are within the Pblock boundary, but not necessarily owned by the Pblock, as well as resources beyond the Pblock rectangle. This means there might be RM nets and partition pins outside of the Pblock boundary. However, any partition pin or contained net is still within the expanded routing footprint.

The expanded routing footprint can be visualized by using the `get_dfx_footprint` Tcl utility. To see the expanded routing footprint, use this utility from the Tcl Console after opening a post-synthesis configuration or a fully routed design. This selects all tiles available to the router, and then selected tiles can be highlighted or marked as desired. In the following figure, the user-defined Pblock that bounds placement is shown in blue, and the expanded routing zone is shown in yellow. Highlight the routing footprint first, followed by placement, as the routing footprint will always be larger.

```
highlight_objects -color yellow [get_dfx_footprint -route -of_objects
[get_cells <rm_inst>]]
highlight_objects -color blue [get_dfx_footprint -placee -of_objects
[get_cells <rm_inst>]]
```

Figure 93: **Highlighted Pblock and Expanded Routing Footprint**



Any frames that are used by the RM must be contained with the partial bit files, so one effect of expanding the routing footprint is larger partial bit files. The increase in size depends on the original Pblock size and shape. Pblocks that are already rectangular can still expand. However, the expansion cannot go beyond a clock region boundary in the vertical direction; it can extend into a new clock region to the left or right. It may help the routability of the RP if the Pblock boundaries stop short of internal clock region edges, especially in the vertical direction. Pblock

edges that align to the device edges, such as left or bottom edges, should not be pulled in just to allow for expanded routing. This causes placement issues if the static region now has access to small pockets of resources along the edges. AMD recommends keeping this routing expansion enabled, but if the partial bitstream size is more critical than the performance of the design, then this feature can be disabled by setting the following parameter:

```
set_param hd.routingContainmentAreaExpansion false
```

IMPORTANT! *The expanded routing footprint is not supported for 7 series devices. The* `get_dfx_footprint` *utility is also not supported for 7 series devices.*

In Vivado, the algorithms that determine the device and design resources included in the expanded routing region have been updated. The expansion region is now slightly smaller than in prior tool versions, resulting in smaller partial bitstreams with minimal impact on design routability. These changes are necessary to support the Abstract Shell feature for UltraScale+ targets.

For DFX designs brought into Vivado for the purpose of generating new partial bitstreams, the original routing expansion is maintained to preserve bitstream compatibility. In other words, if the parent configuration (the run that establishes the static design results) was implemented in an earlier version of Vivado and is not reimplemented in the current version, the partial bitstreams for all new RMs continue to use the older style routing expansion, ensuring compatibility with existing deployed static platforms.

All designs that are new in the current version of Vivado or whose static designs are reimplemented in this release use the new routing expansion solution. No user intervention is required beyond simply implementing the static design through place and route. According to DFX methodology rules, all child configurations used to create results for the remaining RMs must also be implemented, maintaining compatibility for all partial bitstreams.

IMPORTANT! *The use of the Abstract Shell requires the enhanced expanded routing footprint. All DFX designs intending to use this feature must be implemented in the current version of Vivado, the first production release for Abstract Shell, to ensure correct generation of the abstract shells. Abstract shells cannot be generated from pre-current version design checkpoints.*

UltraRAM Behavior

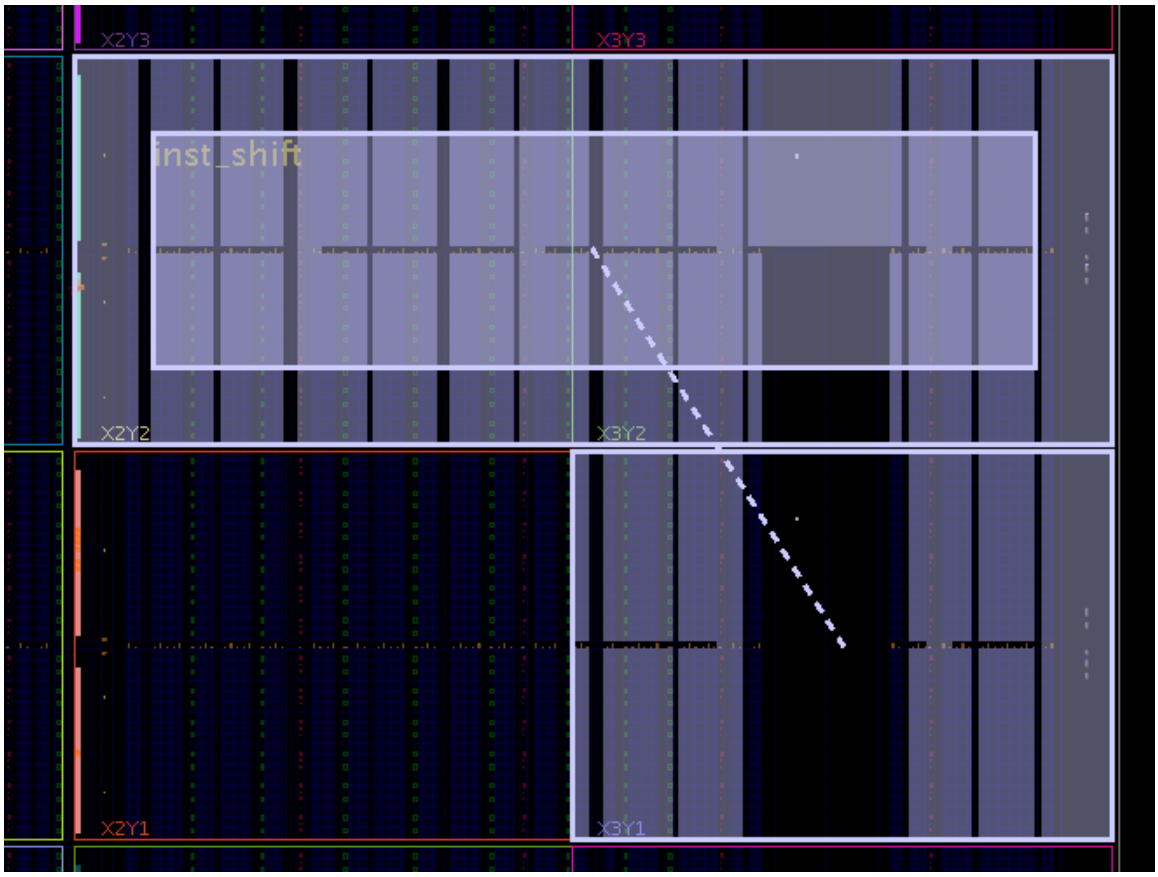
Just as with a full device configuration, UltraRAM memory is initialized to all 0's during partial reconfiguration. There is no user defined INIT attribute and therefore the content of the SRAM array cannot be initialized to user defined values.

Floorplanning Rules for Clocks Inside an RP

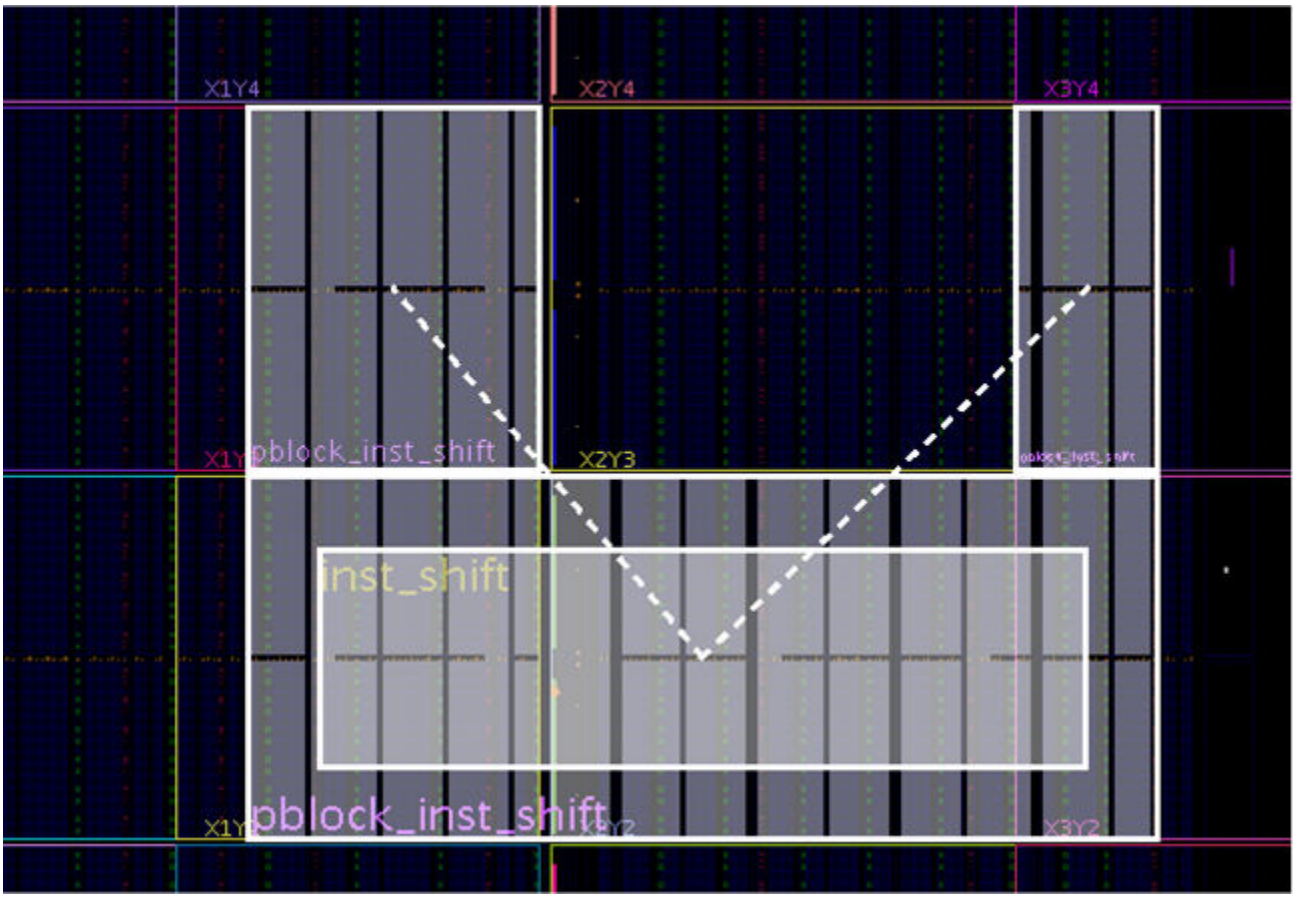
UltraScale and UltraScale+ devices support clocking resources within the RP such as BUFG_*, PLL, and MMCM. Designs incorporating this feature should follow the general design restrictions described in Global Clocking Rules as well as the additional floorplanning rules below. These rules are required to ensure that the clocks internal to the RP can reach the necessary routing resources within the frames owned by the RP Pblock.

1. Create rectangular Pblocks whenever possible. If the Pblock is made up of multiple rectangles, the tallest column of the Pblock must be `clock_region` aligned.
2. The CLOCK_ROOT property of the internal RM clock should be set as one of the tallest columns in the Pblock. The tools attempt to pick the correct columns for the CLOCK_ROOT automatically, but in some cases this cannot be done.
   a. If a USER_CLOCK_ROOT property exists on the clock net, then the tools will not automatically select the CLOCK_ROOT. If the USER_CLOCK_ROOT property is set to a column that is not the full height of the Pblock, unroutable connections might occur.
   b. Certain configurations of BUFG_GT require that the CLOCK_ROOT be in the same region as the BUFG_GT. If this is not the tallest column of the Pblock, unroutable connection might occur. To resolve this, consider splitting the clock net into two BUFG_GT (one for user logic, and the other for the direct GT connections). This way each clock can have its own CLOCK_ROOT.

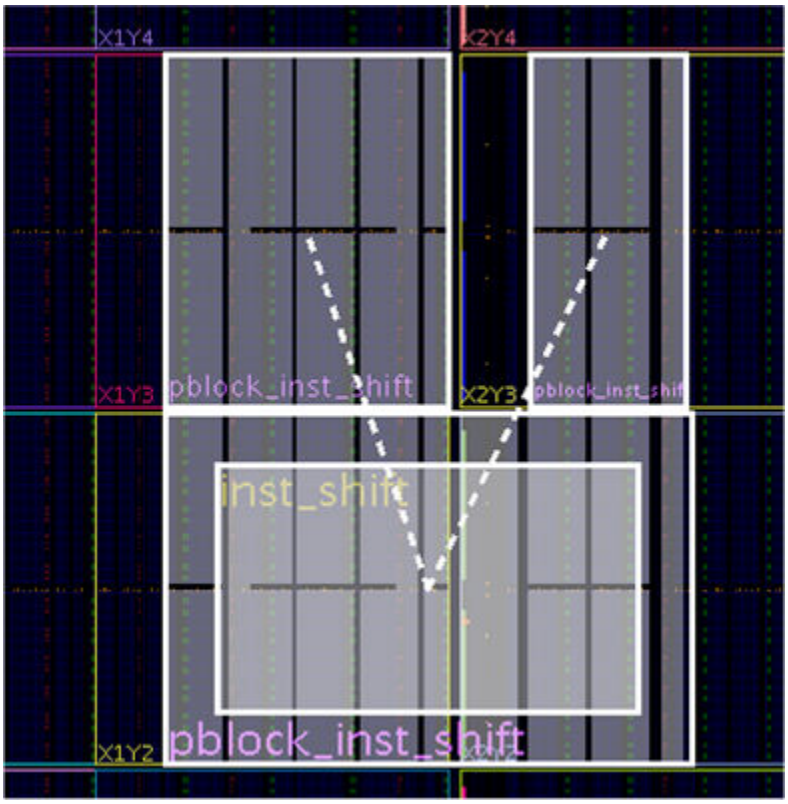
      As shown in the following figure, a CLOCK_ROOT defined in region `X2Y2` (top-left of the Pblock) would prevent routing to any loads in region `X3Y1` (bottom-right of the Pblock), because the region `X2Y1` is not available to the clock. Conversely, if the CLOCK_ROOT were defined in either `X3Y2` or `X3Y1`, no clock routing restrictions would apply.



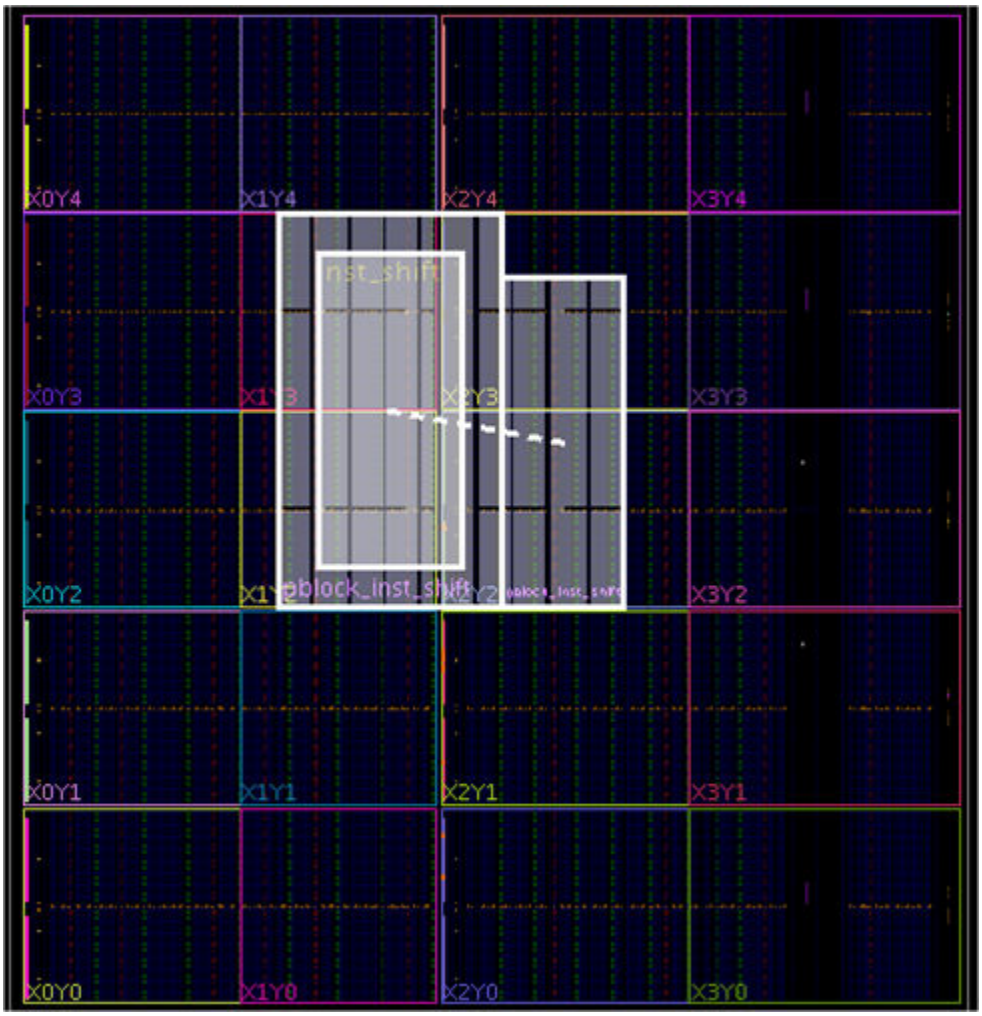
3. If a CLOCK_ROOT cannot be set to the tallest column, the loads of the clock can be contained to regions accessible by the clock using nested Pblocks within the RP region. The nested Pblock will prevent the placer from putting a load in a region that is not accessible by the clock due to irregularly shaped Pblocks.
4. Do not create U or H shaped Pblocks with large gaps that span an entire clock region, as shown in the following figure.



Small static gaps, such as an IOB column, are permitted in the row of an RP Pblock as shown in the following figure. However, AMD recommends avoiding these gaps when possible, as they are a potential source of routing congestion because RM routes need to route over these gaps.



Small stair-step shaped Pblocks are sometimes necessary, as shown in the following figure. While they are supported, they can also lead to routing congestion around the corners.



5. RP Pblocks with clocking resources cannot share any part of a clock region with any other RP. It can share a clock region with static logic. This is true regardless of where the logic driven by these clock resources exist—inside or outside of the given RP.

Related Information

Global Clocking Rules

Global Clocking Rules

As with architectures previous to UltraScale, all unique clocks driving the RP are pre-routed to every clock region in which the RP owns sites. Effectively, this means that the total number of global clocks driving the RP (regardless of size) is a maximum of 24. Higher clock utilization is possible when the clock source is in the RM, since these do not need to be pre-routed to every clock region. For this reason it is always important to carefully consider the RP Pblock size and shape. However, one difference in the UltraScale architecture is that there are now 24 global clocks available per clock region instead of the 12 available in 7 series devices.

***Note*:** For BUFGCTRL components, the `PRESELECT_I0` and `PRESELECT_I1` properties are ignored during partial reconfiguration, even with `RESET_AFTER_RECONFIG` enabled. The clock source selected depends only on the select and clock enable inputs of the `BUFGCTRL` instance.

Clock sources that exist within RM can drive logic to the static design or other RM. Vivado handles many of the low-level details, such as consistent clock spine usage. Design rule checks ensure that fundamental rules are applied. There are, however, a few considerations to be aware of:

- Clock resource must be consistent from one RM to the next for a given RP. While you may change characteristics of the clock such as MMCM parameters, the clock driver type (BUFG, etc.) must remain fixed so routing resources can remain consistent in the locked static design.
- Clock behavior is indeterminate during reconfiguration, so consider decoupling. Because clocks will be using high-fanout routing resources, a basic 2-to-1 MUX or register will not be appropriate like it would be for standard interfaces. Instead, a BUFGMUX can be used to block the clock source during reconfiguration. Alternately, synchronous elements in static or other RMs can be disabled or held in reset while the clock source is reconfigured.

I/O Rules

In UltraScale and UltraScale+ devices, I/O logic and buffers can be included in an RP. While the I/O can be modified from one RM to another, there are some rules that must be followed.

The following checks are done between all configurations that use the I/O sites. If an I/O site changes from being used to unused, or vice versa, then these checks are not done for those configurations. If an I/O is unused in a particular configuration, make sure the appropriate property for the design is set on these ports via the PULLTYPE attribute. For more information on setting PULLTYPE, see *Vivado Design Suite Properties Reference Guide* (UG912).

- The I/O direction and I/O standard for any given pin must be the same between all RMs whenever the I/O is used. I/O pins can swap between used and unused, but the usage must be consistent.

- The voltage rail for any I/O bank must not change from one RM to the next. Even if the sets of I/O pins within a bank are completely different for different RMs, the voltage required for the I/O standards within that bank cannot be modified on the fly.
- For `DCI_CASCADE`, the member bank assignments between RMs cannot overlap.
  - Legal example: In Configuration 1, `DCI_CASCADE` has banks 12, 13. In Configuration 2, `DCI_CASCADE` has banks 14, 15, and 16. They do not have overlapped banks.
  - Illegal example: In Configuration 1, `DCI_CASCADE` has banks 12 and 13. In Configuration 2, `DCI_CASCADE` has banks 13, 14, 15, and 16. In this case bank 13 overlaps.
- For `DCI_CASCADE`, member banks must be fully contained within the reconfigurable region. All of the member banks for the same `DCI_CASCADE` must be in either the same RP Pblock, or completely in static. `DCI_CASCADE` usage must remain consistent between different RM.
- DCI calibration is automatically done for any IO bank included in a RP at the end of partial reconfiguration, in the same way it is done at the initial configuration of the device. DCIRESET or any user intervention is not necessary.

Changes to the IOB from one configuration to another are limited by the rules above. This means that the following I/O characteristics can be modified through Dynamic Function eXchange:

- Usage (used vs. unused, per I/O)
- Drive Strength (12 mA, 8 mA, etc.)
- Driver Output Impedance (40Ω, 48Ω, etc.)
- Driver Input Impedance (40Ω, 48Ω, etc.)
- Driver Slew Rate (slow, fast, etc.)
- ODT Termination (RTT_40, RTT_60, etc.)

Adding the I/O sites into the RP requires that the entire PU (encompassing the I/O bank, BITSLICE, MMCM, PLL, and one column of CLBs plus shared interconnect) be added. All components in this fundamental region are reconfigured and reinitialized, and adding these other site types to the reconfigurable region can be beneficial in some cases for these reasons:

- Adding I/O sites allows use of the routing resources of the I/O, which reduces congestion (instead of increasing congestion, as it could if the I/O sites were in Static, and caused a gap in the reconfigurable region).
- Allows reconfiguration of other clocking resources like the MMCM and PLL.
- Allows reconfiguration of other I/O logic sites such as BITSLICE and BITSLICE_CONTROL.

Regardless of whether or not the I/O usage or characteristics change during reconfiguration, the entire bank is reconfigured. During reconfiguration, all I/O in the banks defined by the RP Pblock is held with the dedicated global tri-state (GTS) signal, which is released at the end of reconfiguration.

If the RM contains an MMCM or PLL component, the size of the partial bitstream is at its smallest when the lock cycle of these components are set to "no wait." Similarly, IO with DCI matching requirements have minimal bitstream sizes when the lock cycle is set to “no wait.” Set these options using this commands:

```
set_property BITSTREAM.STARTUP.LCK_CYCLE NoWait [current_design]
set_property BITSTREAM.STARTUP.MATCH_CYCLE NoWait [current_design]
```

During the Dynamic Function eXchange flow, RMs are carved out using `update_design -black_box`. During this command any embedded IO buffers, and the associated constraints, such as `PACKAGE_PIN` and `IOSTANDARD`, are removed. When the black box RP is filled in with a new RM, these IOB constraints need to be reapplied to the design.

Using High Speed Transceivers for UltraScale and UltraScale+ Devices

AMD high speed transceivers (GTH, GTY) are supported within an RP. As with other reconfigurable site types, the entire PU must be included. For the UltraScale GT transceivers, the PU includes:

- 4 GT_CHANNEL sites (GT Quad)
- Associated GT_COMMON site
- Associated BUFG_GT_SYNC sites
- Associated BUFG_GT sites
- Associated Interconnect and CLB sites

The required GT PU is the entire height of a clock region. As with previous architectures, it is also possible to leave the GT components in static logic and change the functionality through the DRP. For more information on using UltraScale and UltraScale+ transceivers, see the *UltraScale Architecture GTH Transceivers User Guide* (UG576) or the *UltraScale Architecture GTY Transceivers User Guide* (UG578).

Virtex UltraScale+ High Bandwidth Memory (HBM) devices support Dynamic Function eXchange just like any other UltraScale+ device. Users have the choice of where the HBM Controllers are placed: in the static region or in the dynamic region. If the AXI High Bandwidth Memory Controller IP is kept in the static region with an active HBM reference clock, the memory interface will remain active and all memory contents are retained. Self-refresh mode can be used for power savings when the memory is not needed. If this IP is placed in a RM, it will be reconfigured and memory contents will be reinitialized per its Vivado customization. For more information on the HBM Controller IP, consult the documentation at https://www.xilinx.com/products/intellectual-property/hbm.html.

Dynamic Function eXchange Checklist for UltraScale and UltraScale+ Device Designs

AMD highly encourages the following for an UltraScale and UltraScale+ device design using Dynamic Function eXchange:

Recommended Clocking Networks

Are you using Global Clock Buffers or Clock Modifying Blocks (MMCM, PLL)?

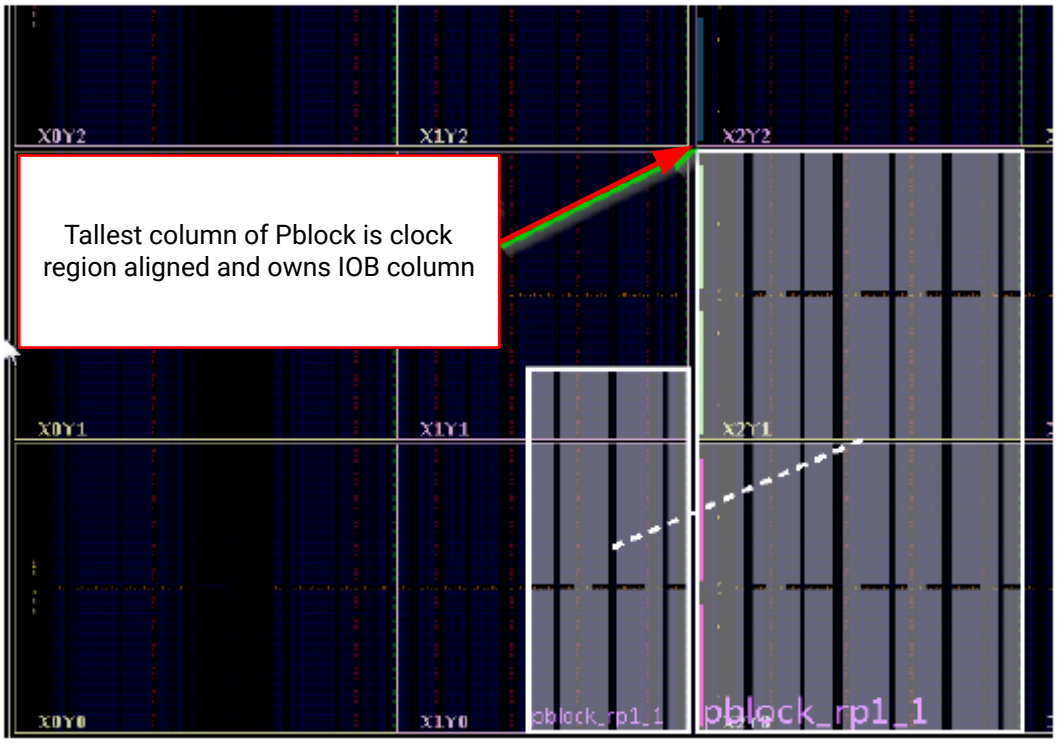
These blocks can be reconfigured, but all elements in this frame type must be reconfigured. This includes an entire I/O bank and all clocking elements in that shared region, plus one column of CLBs that share the interconnect.

See Design Elements Inside Reconfigurable Modules for more information, and Global Clocking Rules for complete details on global clock implementation.

In addition, the following restrictions are currently enforced by Vivado Design Suite DRC rules. The use of clocking resources `BUFGCTRL`, `BUFG_CE` and `BUFG_GT` is supported with the following restrictions:

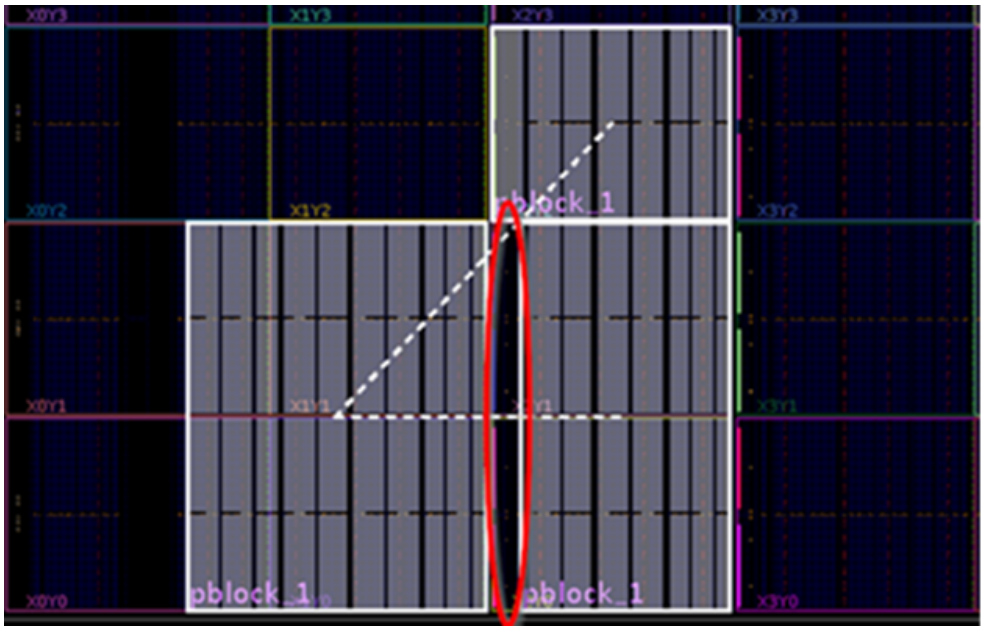
- AMD recommends using rectangular Pblock shapes. Non-rectangular shapes are also supported for RPs with clocking logic, as long as the tallest column of the Pblock is aligned vertically and horizontally with the clock region. The tallest column of the RP Pblock must also range the IOB, and this range must cover the full height of all the rectangles that define the RP Pblock, as shown in the following figure. In other words, this vertical column of IOB ranges must be able to access all rows of the Pblock. Pblock shapes like a sideways "L" are not supported unless the vertical section of the shaped includes the IOB range.

Figure 94: **Tallest Column of Pblock Clock Region Aligned**



- A gap is defined as an unranged site type with ranged sites on both sides of it. The following gaps are not allowed:
  - Gaps in the IOB/XIPHY ranges, such as the gap in the IOB column shown in the following figure.

Figure 95: **Unsupported Gap in the IOB Column**



  - Gaps in the DSP ranges.

- A clock region cannot be shared by two RP Pblocks if:
  - At least one of them has a global clock source.
  - The other has ranged a global clock source.

Configuration Feature Blocks

Are you using device feature blocks (BSCAN, DCIRESET, FRAME_ECC, ICAP, STARTUP, USR_ACCESS)?

These featured blocks must be in static logic.

See Using High Speed Transceivers for UltraScale and UltraScale+ Devices for more information.

Pblock Boundaries

Have you set the Pblock boundaries?

For UltraScale and UltraScale+ devices, the X-axis boundary of a dynamic region can be set by a PU, including CLB, Block RAM, DSP, and others. The tool adjusts the Pblock automatically for a valid placement. The Y-axis boundary of a PR region can be a clock region and IO bank. However, if BUFGCTRL/BUFG_CE/BUFG_GT are used in the RP, a full clock region must be used.

SSI Technology

Does the Pblock span an SLR of an SSI device?

If using an SSI device it is recommended to keep a dynamic region within a single SLR. However, for UltraScale and UltraScale+ devices, if a RP Pblock must span an SLR, the necessary Laguna sites must be included to allow for routing across this boundary. This requires that at least one full clock region belongs to the dynamic region on both sides of the SLR boundary.

For more information on SSI Technology devices and Laguna, see Devices using Stacked Silicon Interconnect (SSI) Technology in the *UltraScale Architecture Configurable Logic Block User Guide* (UG574).

High Speed Transceiver Blocks

Do you have high speed transceivers in your design?

High speed transceivers can be reconfigured. An entire quad, including all component types (GT_CHANNEL, GT_COMMON, BUFG_GT) must be reconfigured together.

See Using High Speed Transceivers for UltraScale and UltraScale+ Devices for specific requirements.

System Generator DSP Cores, HLS Cores, or IP Integrator Block Diagrams

Are you using System Generator DSP cores, HLS cores, or IP integrator block diagrams in your Dynamic Function eXchange design?

Any type of source can be used as long as it follows the fundamental requirements for Dynamic Function eXchange. Any code processed by System Generator, HLS, or IP integrator (or other tools) is eventually synthesized. The resulting design checkpoint or netlist must be comprised entirely of reconfigurable elements in order for it to be legally included in an RP.

Packing I/Os into Reconfigurable Partitions

Do you have I/Os in RMs?

I/Os can be partially reconfigured. An entire I/O bank, along with all I/O logic (XiPhy) and clocking resources, must be reconfigured at once. IOSTANDARD and direction cannot change and DCI Cascade rules must be followed. But other I/O characteristics may change from one RM to the next.

See Design Elements Inside Reconfigurable Modules for more information.

Packing Logic into Reconfigurable Partitions

Is all logic that must be packed together in the same RP?

Any logic that must be packed together must be in the same RP and RM.

See Packing Logic for more information.

Packing Critical Paths into Reconfigurable Partitions

Are critical paths contained within the same partition?

RP boundaries limit some optimization and packing, so critical paths should be contained within the same partition.

See Packing Logic for more information.

Floorplanning

Can your RPs be floorplanned efficiently?

See Creating Pblocks for UltraScale and UltraScale+ Devices for more information.

Recommended Decoupling Logic

Have you created decoupling logic on the outputs of your RMs?

During reconfiguration the outputs of RPs are in an indeterminate state, so decoupling logic must be used to prevent static data corruption.

See Decoupling Functionality for more information.

Recommended Reset After Reconfiguration

Are you resetting the logic in an RM after reconfiguration?

Reset After Reconfiguration is always enabled for UltraScale and UltraScale+ devices. This capability cannot be disabled.

See Apply Reset After Reconfiguration for more information.

Debugging with Logic Analyzer Blocks

Are you using the Vivado Logic Analyzer with your Dynamic Function eXchange design?

Vivado logic analyzer (ILA/VIO debug cores) can be used in your Dynamic Function eXchange design, but care must be taken when connecting these cores to debug hubs. Use the automatic inference solution shown in Using Vivado Debug Cores.

Efficient Reconfigurable Partition Pblocks

Have you created efficient RP Pblock(s) for your design?

A RP Pblock can be any height, but multiple RPs cannot be stacked vertically within a single clock region.

See Creating Pblocks for UltraScale and UltraScale+ Devices for more information.

Validating Configurations

How do you validate consistency between configurations?

The `pr_verify` command is used to make sure all configurations have matching imported resources.

See Verifying Configurations for more information.

Configuration Requirements

Are you aware of the particular configuration requirements for Dynamic Function eXchange for your design and device?

Each device family has specific configuration requirements and considerations.

See Configuring the Device for more information.

Recommended Floorplanning Constraints

These recommendations apply to both AMD UltraScale™, AMD UltraScale+™ devices.

Reduce the Number of Partition Pins

In a DFX design, signals between the reconfigurable module (RM) and static region are called *boundary signals*. All RM pins must have a partition pin location (PPLOC) deposited on the boundary signal by the placer. The only exceptions are dedicated paths between hard primitives. The partition pin is the physical interface on fabric that separates the static and reconfigurable portions of a boundary signal.

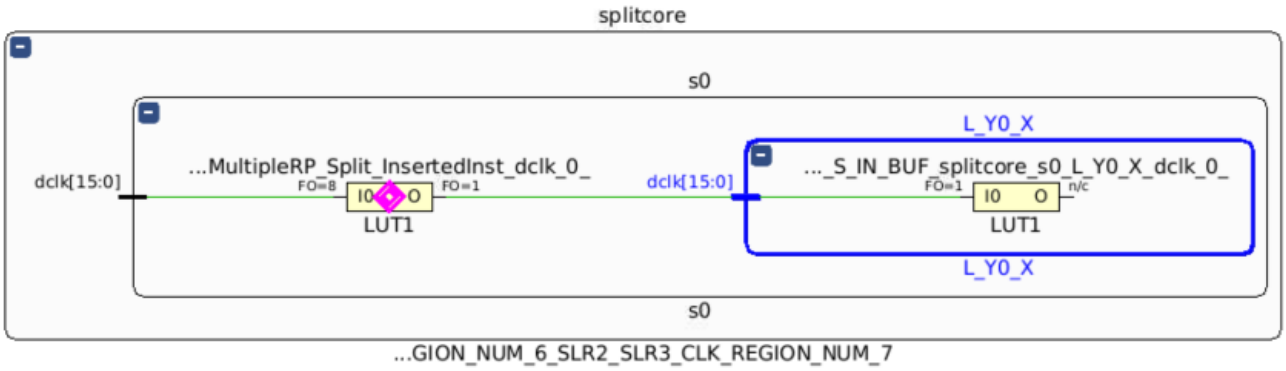
The presence of partition pins reduces the solution space for the router, because the corresponding boundary net is always forced to route through the partition pin. To alleviate this issue, the DFX flow includes expanded routing. Expanded routing is the additional routing footprint for a reconfigurable partition (RP) that can include routing tiles from the static region.

The boundary nets of an RM have fanout in the static region as well as within the RM. In a DFX design, the loads of a boundary net in the static region are static boundary leaf cells. When a static boundary leaf cell is placed in the expanded routing footprint of an RP, the PPLOC is not needed and the router will have more flexibility routing to the static cell in subsequent RM implementations. AMD recommends using the expanded routing feature to reduce the dependency of the router on PPLOCs. This feature is on by default.

***Note*:** PPLOC reduction occurs only on single fanout boundary signals. Therefore, AMD recommends avoiding multiple fanouts for boundary signals in a DFX design.

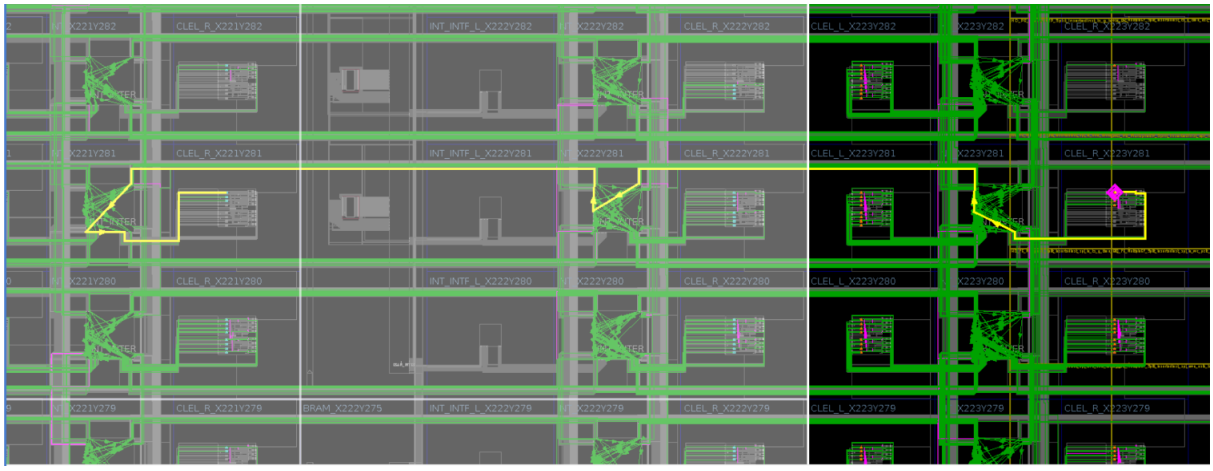
If PPLOC reduction is triggered for a boundary signal when a blackbox is created for the RM after initial implementation, the boundary net is removed up to the SITE or BEL pin of the static boundary leaf pin. Subsequent RM implementations route the boundary signal from the static boundary leaf pin to the RM logic. The following figure shows the PPLOC reduction in a boundary signal. LUT1 is the boundary static leaf cell and L_Y0_X is the RM.

Figure 96: **Schematic of a Boundary Signal**



The following figure shows the Device window for the boundary net after `route_design`. The boundary signal is shown in yellow, and the boundary static leaf cell is shown in magenta. In this example, the boundary static leaf cell is placed in the expanded routing footprint of an RM. After initial implementation is complete and a blackbox is created for the RM, the boundary net is removed up to the static boundary leaf pin.

Figure 97: **Routed Device Window for Boundary Net**

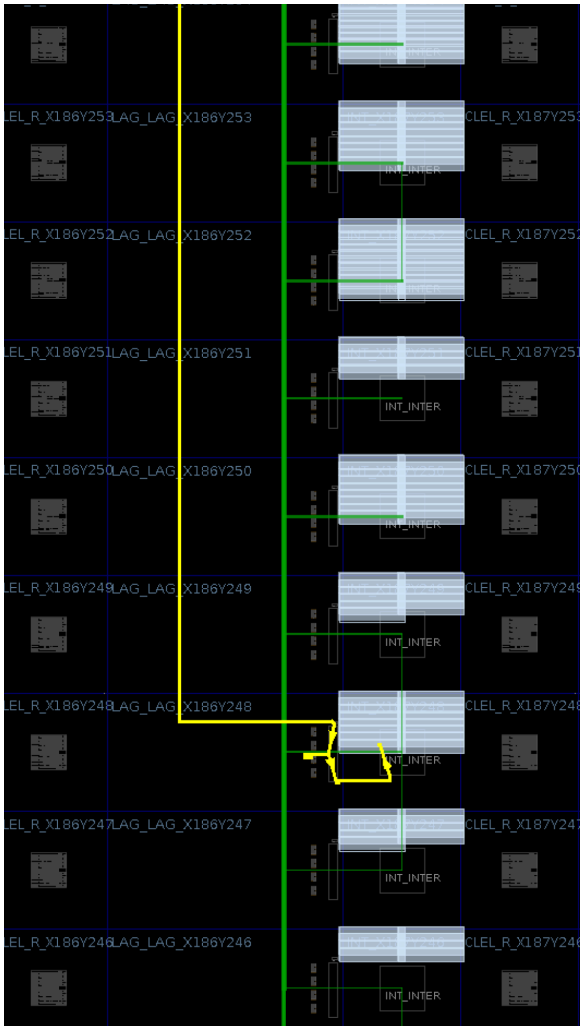


If PPLOC reduction is *not* triggered for a boundary signal when the blackbox is created for the RM after initial implementation, the static boundary net segment is preserved from the static leaf pin to the PPLOC. During subsequent RM implementations, boundary nets are routed only from the PPLOC to the RM logic. This reduces the solution space for the router due to the following:

- The static segment of the boundary net (from the static boundary leaf pin to the PPLOC) is locked down during all subsequent implementations. The router must obey the IS_FIXED_ROUTE constraint on the static segment of the boundary signal and cannot reroute during subsequent RM implementations.
- The presence of locked static nets (IS_FIXED_ROUTE TRUE) at the boundary of the reconfigurable Pblock causes the tool to exclude some sites from placement, because access to these sites might be blocked by the locked static nets.
- The PPLOC deposit occurs only on single or double interconnect nodes. This approach has lower connectivity than using site pins and is equivalent to having no PPLOCs.

The following figure shows the Device window for the static boundary net segment that terminates at the PPLOC when the static boundary leaf cell is not placed in the expanded routing footprint.

Figure 98: **Device Window for Static Boundary Net Segment**

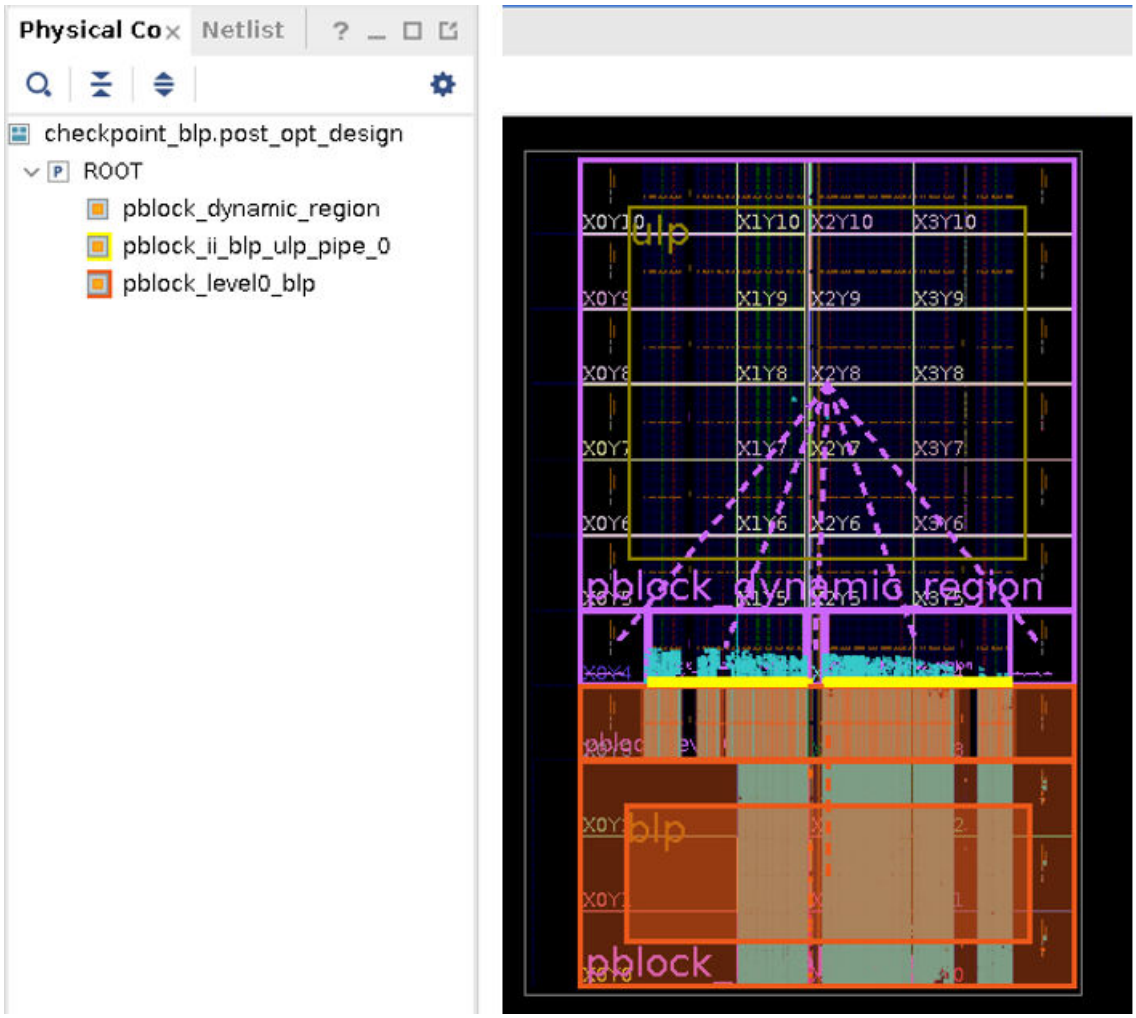


The routing footprint of the reconfigurable Pblock (pblock_dynamic_region) is the same as the reconfigurable Pblock size (CLOCKREGION_X0Y4: CLOCKREGION_X4Y10). All of the static region logic is assigned to the static Pblock region (pblock_static_region), which is outside the routing footprint of the reconfigurable Pblock. Therefore, PPLOC reduction is not triggered, and the reconfigurable Pblock contains a large number of PPLOCs after `route_design`.

In the following example, static boundary leaf cells to the reconfigurable Pblock (pblock_dynamic_region) are assigned to a thin static Pblock (pblock_ii_blp_ulp_pipe_0), which is defined in the expanded routing footprint of the RP Pblock. There are no PPLOCs remaining after `route_design`.

The following figure shows the static boundary leaf cells assigned to a thin static Pblock defined in the expanded routing footprint of the RP.

Figure 99: **Static Boundary Leaf Cells in the Expanded Routing Footprint of the RP**



To achieve maximum PPLOC reduction, AMD recommends that you guide the placer to keep static boundary leaf cells in the expanded routing footprint of the reconfigurable Pblock. One way to achieve this is to use thin static Pblocks defined in the expanded routing footprint of the reconfigurable Pblock.

TIP: *To highlight the tiles in the routing footprint of a reconfigurable Pblock, use* `get_dfx_footprint -route` *to select the tiles owned for routing by a reconfigurable partition.*

Recommended Netlist Structure at the DFX Boundary for Maximum PPLOC Reduction

Avoid RP to RP Direct Paths

AMD recommends avoiding direct timing paths between multiple reconfigurable partitions (RPs) for the following reasons:

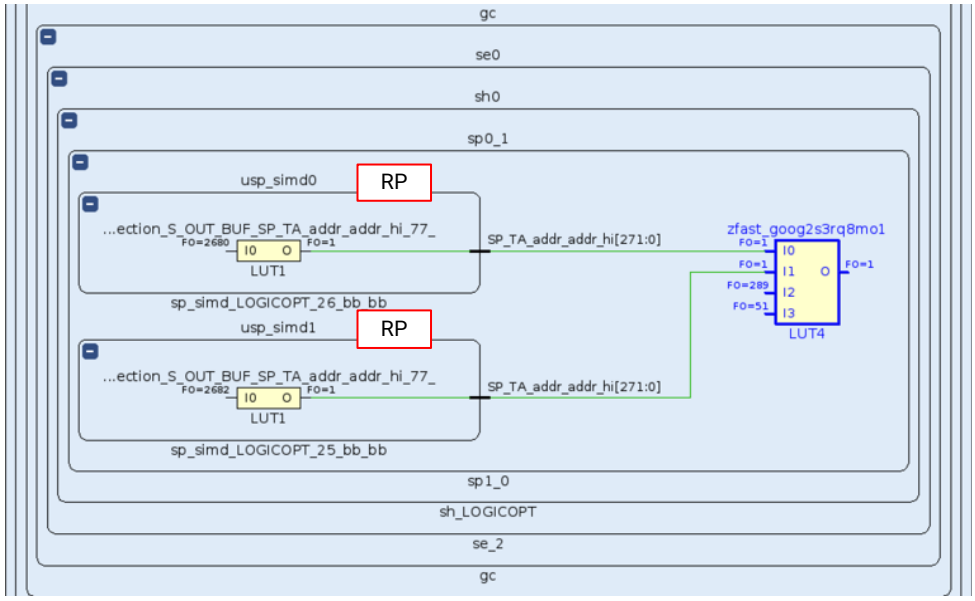
- If the boundary signal between the RPs does not have a static boundary leaf cell, the DFX flow must deposit a PPLOC on both RPs. As a result, tool capabilities like expanded routing with PPLOC reduction cannot be used. The presence of PPLOCs also causes routability challenges in subsequent reconfigurable module (RM) implementations, also known as child implementations in Vivado Project Mode.
- If the timing paths across an RP do not have a static boundary leaf cell, there might be a combination of RMs in two RPs that do not meet timing. The omission of a synchronous timing point in the static portion of the design can also lead to timing and hardware failures depending on the RMs that are currently loaded. The HDPR-34 and HDPR-35 DRCs flag this issue.

Avoid Multiple RPs Driving Same Static Leaf Cell

Ensure that a static boundary leaf cell is connected to only one reconfigurable partition (RP). PPLOC reduction is triggered for a reconfigurable module (RM) pin only if the static boundary leaf cell is placed in the expanded routing footprint of the RP. A leaf cell with individual leaf pins connected to multiple RPs can be placed in the expanded routing footprint of only one RP. Therefore, PPLOC reduction cannot occur on two RPs at the same time.

In the following example, two RPs are connected to the same static boundary leaf cell. This approach is *not* recommended, because it negatively impacts routability.

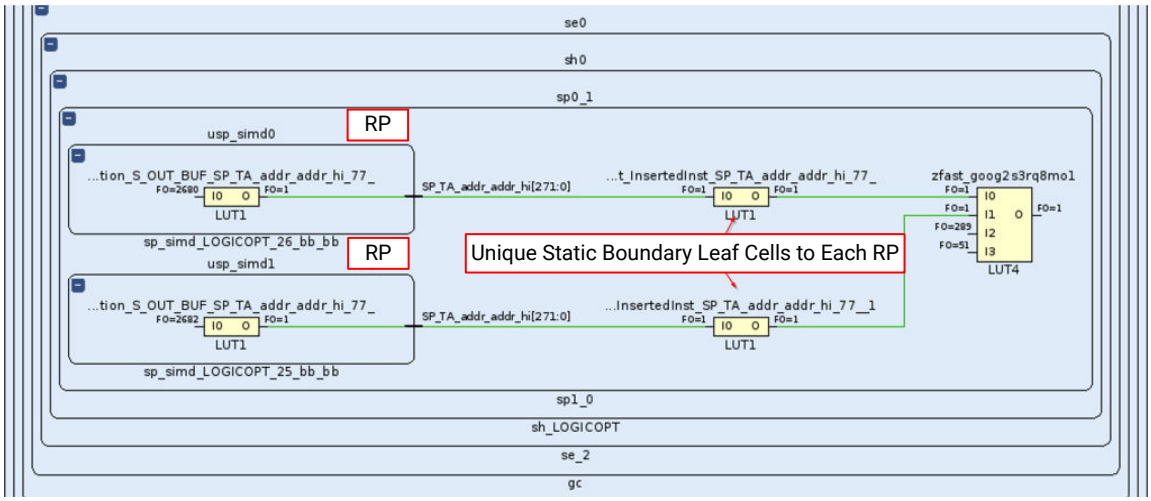
Figure 100: **Two RPs Driving the Same Static Boundary Leaf Cell**



X25536-070821

In the following example, two RPs each drive different static boundary leaf cells. LUT1 is inserted to separate the static boundary leaf cells. This approach is recommended for improved routability.

Figure 101: **Two RPs Driving Unique Static Boundary Leaf Cells**



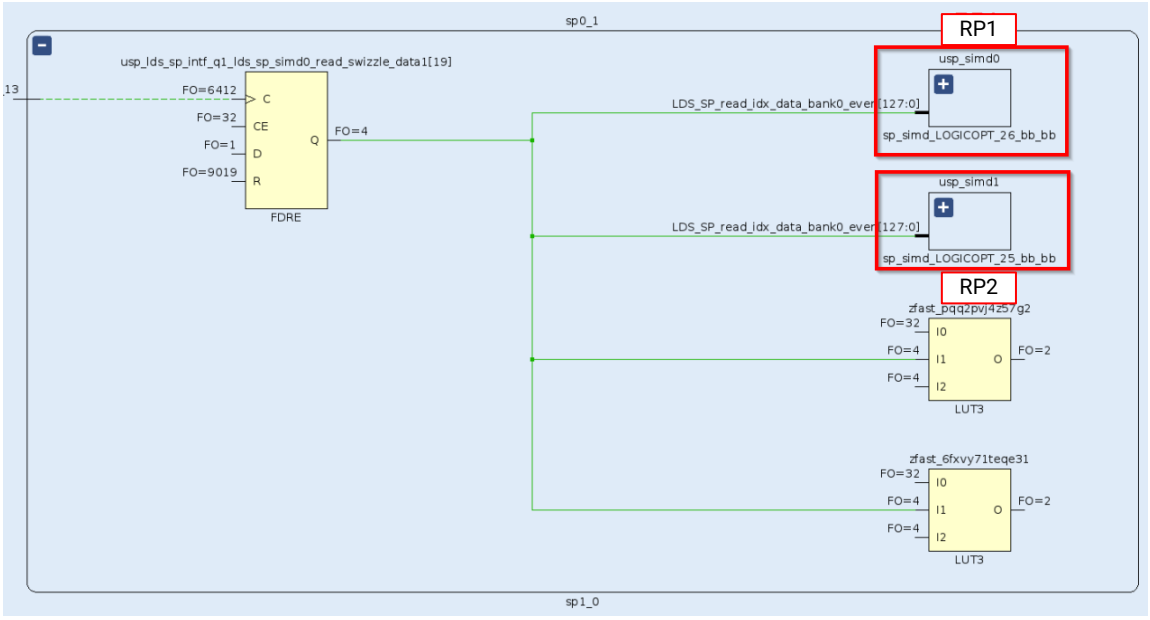
X25535-071221

Replicate Static Register Driving Multiple RPs

To ensure that a static boundary leaf cell is not shared by multiple reconfigurable partitions (RPs), avoid single registers that drive multiple RPs. The DFX-1 methodology check (`report_methodology`) flags this violation.

In the following example, a single static register drives multiple RPs as well as static logic. This approach is *not* recommended in the DFX flow.

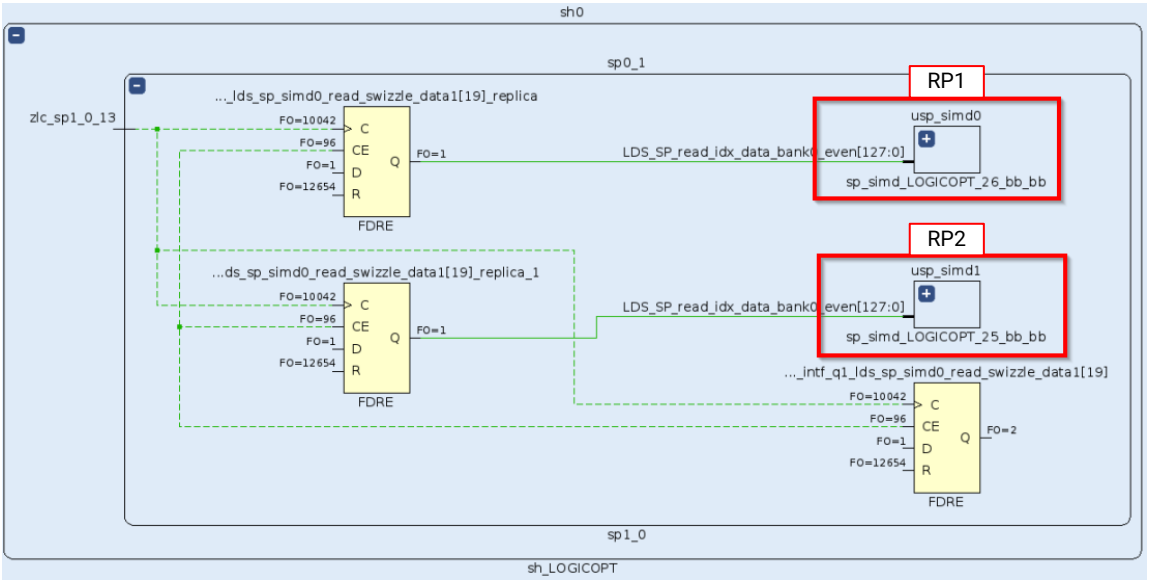
Figure 102: **One Static Register Driving Multiple RPs and Static Logic**



X25534-070821

In the following example, two separate but equivalent registers drive two RPs. This is the recommended approach when using the DFX flow.

Figure 103: **Two Registers Driving Two RPs**



X25533-070821

Register Inputs and Outputs of RMs

AMD recommends registering inputs and outputs of reconfigurable modules (RMs) in a DFX design for multiple reasons.

In the parent implementation, the RM used for the partition does not need to be the actual design. Instead, the RM can be training logic used as a placeholder while you define the platform. If the training logic is sub-optimal and there is combinatorial logic in the static portion of the boundary timing paths, it is likely that the static portion of the path consumes a significant amount of timing budget of the path. During child implementation, this can cause timing closure issues for the signals in the RM connected to this boundary signal.

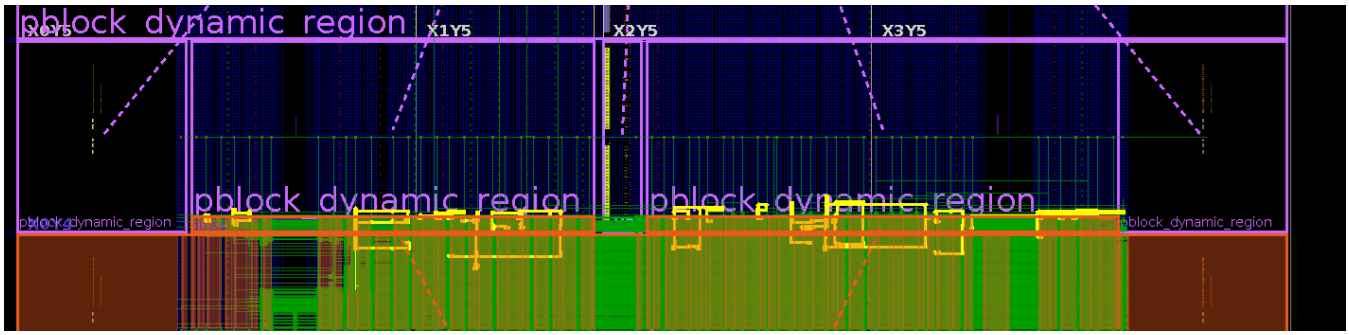
In addition, creating an abstract shell of a reconfigurable partition (RP) prunes most of the static region and keeps only the logic up to the first sequential cell in the static region. Registering the input and output pins of the RMs enables maximum abstraction, thereby reducing the size of the abstract shell.

Reduce Bleed Over of Static Nets to the Reconfigurable Pblocks

By default, nets in the static region of a DFX design can use any routing resources in the device. However, this might cause the static nets to bleed over to the dynamic region. Although allowed from a functional perspective, this approach reduces the solution space of the router for reconfigurable modules (RMs) inside the reconfigurable partition (RP) Pblock. After the first implementation is complete and static routes and placement are locked using the `lock_design` command, the static nets are locked with some of the nets in the RP Pblock. During subsequent child implementations, the DFX flow identifies these locked static nets during RM implementation and attempts to perform place and route in a reduced solution space to avoid unroutability. To avoid the bleed over of static nets to RP Pblocks, AMD recommends containing the static nets inside a static Pblock by setting the CONTAIN_ROUTING TRUE constraint.

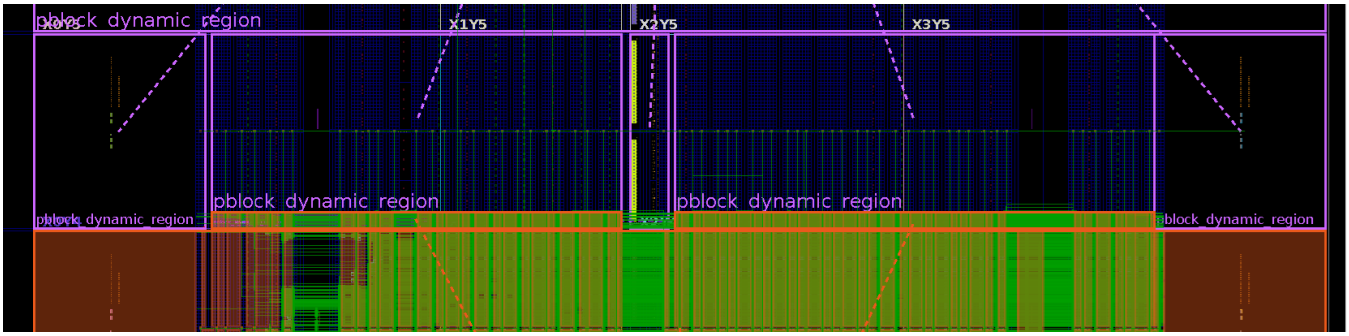
In the following example, the static region Pblock does not have the CONTAIN_ROUTING constraint enabled. The bleed over from the static nets the to RP Pblock is highlighted in yellow. This approach is *not* recommended, because it negatively impacts routability during RM compile.

Figure 104: **Static Region Pblock without CONTAIN_ROUTING Constraint**



In the following example, the CONTAIN_ROUTING constraint is enabled on the static Pblock, and there is no bleed over of static nets to the RP Pblock. This is the recommended approach for improved routability during RM compile.

Figure 105: **Static Region Pblock with CONTAIN_ROUTING Constraint**

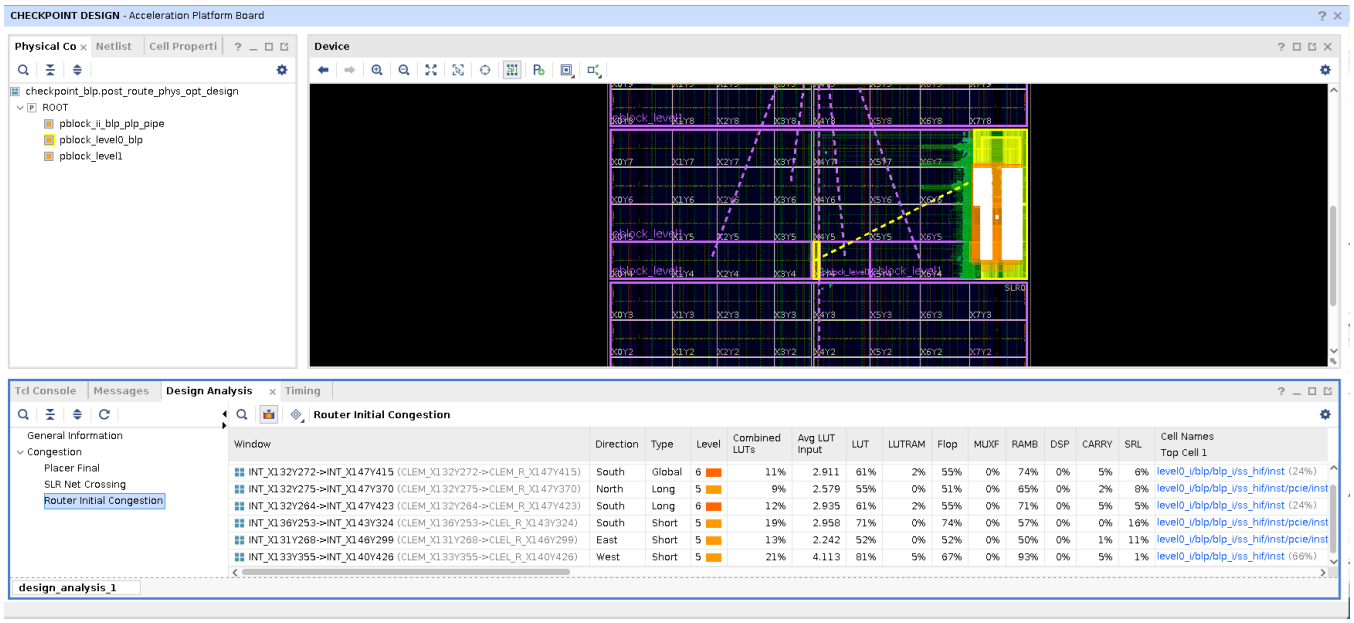


Make Pblocks as Rectangular as Possible to Avoid Unroutability at the Edges

For Pblocks with CONTAIN_ROUTING enabled, the router has less solution space at the corners of the Pblock. To avoid unroutable situations, AMD recommends making Pblocks as rectangular as possible. Using another shape results in more corners for the Pblock, which can introduce congestion at those corners.

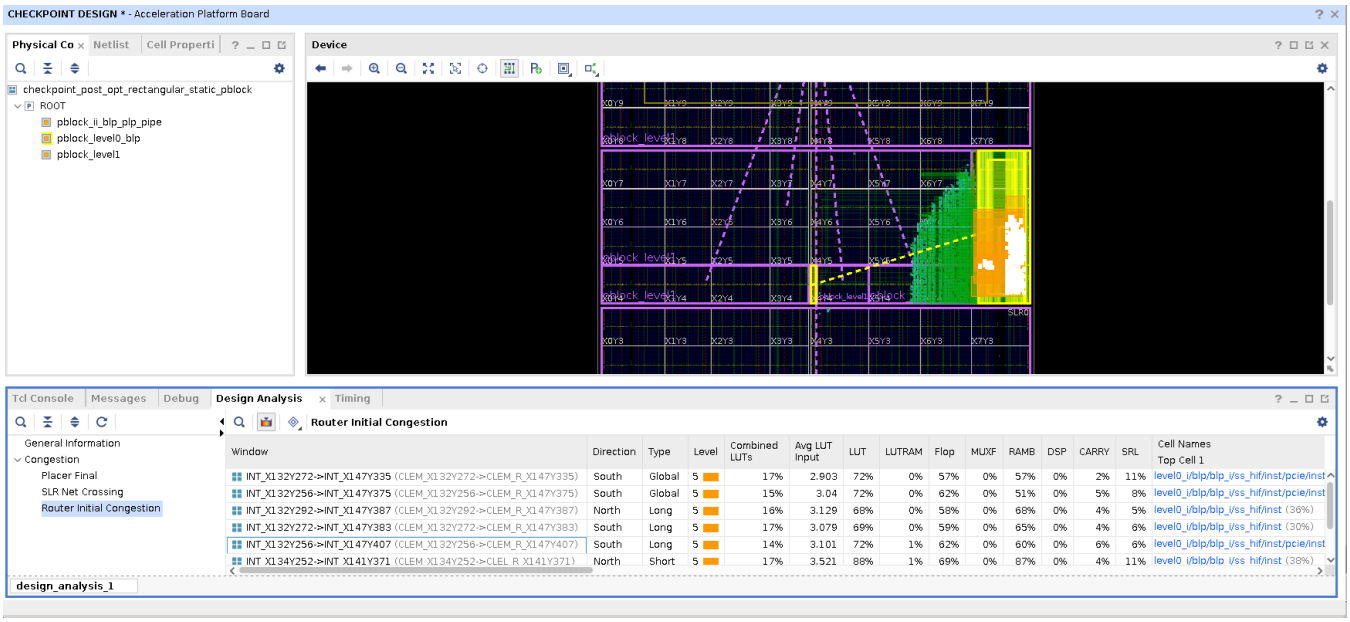
The following figure shows a non-rectangular static Pblock (shown in yellow) with an overlapping congestion area (shown in white). In this example, `report_design_analysis` reports a congestion of level 6 at the edges of the Pblock. This approach is *not* recommended.

Figure 106: **Non-Rectangular Static Pblock with Congestion at the Corners**



The following figure shows the same design but with a rectangular Pblock (shown in yellow) and a reduced congestion area (shown in white). With this approach, `report_design_analysis` reports a reduced congestion level of 5 with no congestion reported at the edge of the rectangular Pblock. This is the recommended approach.

Figure 107: **Rectangular Static Pblock with Reduced Congestion**



Considerations for Static Pblocks with CONTAIN_ROUTING Enabled

Keep Routability as a Factor in Utilization

When defining Pblocks, you might look at utilization from a placement perspective only. However, for static Pblocks with the CONTAIN_ROUTING constraint enabled, routability is also an important factor. Unlike Pblocks without a containment requirement, the router must find a solution using the available routing tiles inside the static Pblock. Therefore, it is important to stay within the recommended utilization value. This approach provides more solution space for the router and allows you to converge to a solution more quickly. See the utilization recommendations for flat designs using:

```
report_qor_assessment -full_assessment_details
```

Reduce the Number of Unique Control Sets

CLB packing restrictions caused by unique control sets can introduce sub-optimal placement and higher net delay. On a Pblock that already has a CONTAIN_ROUTING requirement, the additional restriction of unique control sets puts more constraints on the router, which might lead to an unroutable situation. Therefore, it is very important to reduce the unique control sets on static logic that is assigned to a Pblock, especially if the Pblock has CONTAIN_ROUTING enabled. For techniques to reduce unique control sets, see the *Versal Adaptive SoC System Integration and Validation Methodology Guide* (UG1388).

Reduce the Detour Due to Hold Violations

The router gives priority to fixing hold violations by detouring through longer paths. However, this adds more restrictions if the violations are inside the CONTAIN_ROUTING static Pblocks. There might not be enough solution space for detouring inside the static Pblock that includes a CONTAIN_ROUTING requirement. Therefore, AMD strongly recommends having a good post-place timing summary for such logic. For techniques to reduce unique control sets, see the *Versal Adaptive SoC System Integration and Validation Methodology Guide* (UG1388).

Exclude Containment Requirement for Nets

If you have routability challenges inside a static Pblock that has CONTAIN_ROUTING enabled, try the following options:

- Increase the Pblock size and reduce the overall utilization.
- Allow bleed over of certain static nets to the reconfigurable partition (RP) by setting the following property on those nets:

```
set_property HD.NO_ROUTE_CONTAINMENT 1 [get_nets <net_name>]
```

Avoid Disjoint Pblocks Whenever Possible for UltraScale and UltraScale+ Devices

AMD highly recommends contiguous floorplanning for static regions and reconfigurable Pblocks for many reasons, including the following:

- Disjoint Pblocks for either a reconfigurable partition or the static region can result in related logic being placed in separate areas, which will have a negative impact on routability and timing closure. Without additional guidance, it is possible for the placer to create a situation that is unroutable if related logic is not kept together.
- Although static nets can cross through reconfigurable Pblocks to allow communication between two static islands, it is best to limit this approach, because locked static nets can block routing during subsequent reconfigurable module implementations. Building contiguous regions and minimizing overlap is the best layout strategy for DFX designs.

The Vivado tools determine that two portions of a Pblock are disjoint if the corresponding expanded routing footprint for a collection of resources has a gap between it and another routing footprint section for the same Pblock. The reconfigurable Pblock does not own general routing resources within this gap. This means that finding a solution that connects these disjoint sections is either difficult or impossible, depending on how the Pblock is created. Use the `get_dfx_footprint` Tcl utility (for example, `get_dfx_footprint -route`) to see the routing footprint of a reconfigurable partition Pblock.

Chapter 8

Design Considerations and Guidelines for Versal Devices

This chapter describes the design requirements that are unique to DFX and are specific to AMD Versal™ devices.

To take advantage of the DFX capability of AMD devices, you must analyze the design specification thoroughly and consider the requirements, characteristics, and limitations associated with DFX designs. This simplifies both the design and debug processes, and avoids potential future risks of malfunction in the design.

DFX Design Migration from Standard Flow to Advanced Flow

AMD Vivado™ uses the Advanced Flow architecture for Versal devices. The Advanced Flow provides place and route features designed to improve compile time and performance through more advanced technologies and parallelization techniques. When targeting Versal technologies, the Advanced Flow is always run. 7 series and AMD UltraScale™ architectures do not use the Advanced Flow.

Versal DFX designs created before 2024.2 can be migrated to the Advanced Flow. Upgrading such designs requires more consideration than a typical Vivado version upgrade because of the implementation flow change. You might choose not to migrate designs that are already in production, designs nearing completion, designs for which significant timing-closure effort has already been invested, or designs that depend on features such as Intelligent Design Runs that are not supported by Advanced Flow.

One limitation that makes upgrades from pre-2024.2 releases more restrictive is that placement and routing data cannot be migrated because the Advanced Flow uses different internal data structures than earlier releases. This means any pre-2024.2 design that has run past the Place Design step is incompatible with Vivado post-2024.2. Therefore, platforms released prior to 2024.2 cannot be imported into current versions of Vivado to update reconfigurable modules or create new ones.

IMPORTANT! *It is critical to never mix full or partial PDI images that have been compiled with different implementation tools. You must generate all programming images entirely using Standard Flow, or entirely using Advanced Flow. The `pr_verify` utility flags errors if this is attempted.*

For Vivado projects, automatic migration occurs when the project is opened in Vivado 2024.2 or later. The migration process updates the project's implementation runs so that only Advanced Flow-supported commands are used. After a Versal project has been migrated to Advanced Flow, it cannot be opened in earlier Vivado versions. Therefore, AMD strongly recommends backing up the project under revision control before migration. For non-project users, minor script modifications are required.

After migration, rerun all parent and child configuration runs to establish a new static design image and a new collection of reconfigurable modules. For most designs, synthesis does not need to be rerun. The exception is designs with a single reconfigurable partition targeting devices that use SSI Technology; in these cases, synthesis must also be rerun to ensure the correct NoC data is established and propagated through the flow.

After migration, review implementation results carefully to determine whether additional adjustments are needed. One expected change is in Reconfigurable Partition Pblock handling: the base Pblock snaps outward to include all covered programmable units rather than inward, which can cause more resources to be identified as dynamic instead of static. In some cases, this can create unroutable conditions if the static design no longer retains the resources needed to complete routing.

For more information on the Advanced Flow, refer to *Vivado Design Suite User Guide: Implementation* (UG904) and Answer Record 000036830.

Design Elements Inside Reconfigurable Modules

Versal devices support partial reconfiguration for almost all component types. Logic that can be placed in an RM includes:

- NoC master units (NMUs) and NoC slave units (NSUs)
- Boundary logic interface (BLI) flip-flops
- XPIO and HDIO banks, including XPHY, ISERDES, OSERDES, and IDELAYCTRL
- Memory controllers: DDRMC and DDRMC_RIU
- Serial transceivers (MGTs) and related components: GTYE5_QUAD, MRMAC, PCIE40E5, and GTM_DUAL
- All logic components that are mapped to a CLB slice, including LUTs (look-up tables), LUTRAMs, FFs (flip-flops), SRLs (shift registers), and LOOKAHEAD.

- Block RAM: RAMB18E5 and RAMB36E5
- DSP blocks: DSP58 (DSP58_PRIMARY and DSP58_CPLX)
- High-speed channelized cryptography engines (HSC)
- PCIe® (PCI Express), CMAC (100G MAC), and ILKN (Interlaken MAC) blocks
- UltraRAM blocks: URAM288E5 and URAM288E5_BASE
- Clocks and clock modifying logic, including BUFG_FABRIC, BUFGCE, BUFG_GT, BUFG_GT_SYNC, BUFGMUX, MMCM, DPLL, XPLL, and MBUFG
- AI Engines

  Note: Versal AI Engine inclusion in RMs is supported through Vitis platform flows only.

I/O Rules

In Versal devices, I/O logic and buffers can be included in an RP. While the I/O can be modified from one RM to another, there are some rules that must be followed.

The following checks are done between all configurations that use the I/O sites. If an I/O site changes from being used to unused, or vice versa, then these checks are not done for those configurations. If an I/O is unused in a particular configuration, make sure the appropriate property for the design is set on these ports via the PULLTYPE attribute. For more information on setting PULLTYPE, see *Vivado Design Suite Properties Reference Guide* (UG912).

- The I/O direction and I/O standard for any given pin must be the same between all RMs whenever the I/O is used. I/O pins can swap between used and unused, but the usage must be consistent.
- The voltage rail for any I/O bank must not change from one RM to the next. Even if the sets of I/O pins within a bank are completely different for different RMs, the voltage required for the I/O standards within that bank cannot be modified on the fly.
- For `DCI_CASCADE`, the member bank assignments between RMs cannot overlap.
  - Legal example: In Configuration 1, `DCI_CASCADE` has banks 12, 13. In Configuration 2, `DCI_CASCADE` has banks 14, 15 and 16. They do not have overlapped banks.
  - Illegal example: In Configuration 1, `DCI_CASCADE` has banks 12 and 13. In Configuration 2, `DCI_CASCADE` has banks 13, 14, 15 and 16. In this case bank 13 overlaps.
- For `DCI_CASCADE`, member banks must be fully contained within the reconfigurable region. All of the member banks for the same `DCI_CASCADE` must be in either the same RP Pblock, or completely in static. `DCI_CASCADE` usage must remain consistent between different RM.
- DCI calibration is automatically done for any IO bank included in a RP at the end of partial reconfiguration, in the same way it is done at the initial configuration of the device. DCIRESET or any user intervention is not necessary.

Changes to the IOB from one configuration to another are limited by the rules above. This means that the following I/O characteristics can be modified through Dynamic Function eXchange:

- Usage (used vs. unused, per I/O)
- Drive Strength (12 mA, 8 mA, etc.)
- Driver Output Impedance (40Ω, 48Ω, etc.)
- Driver Input Impedance (40Ω, 48Ω, etc.)
- Driver Slew Rate (slow, fast, etc.)
- ODT Termination (RTT_40, RTT_60, etc.)

Adding the I/O sites into the RP requires that the entire PU (encompassing the I/O bank, BITSLICE, MMCM, PLL, and one column of CLBs plus shared interconnect) be added. All components in this fundamental region are reconfigured and reinitialized, and adding these other site types to the reconfigurable region can be beneficial in some cases for these reasons:

- Adding I/O sites allows use of the routing resources of the I/O, which reduces congestion (instead of increasing congestion, as it could if the I/O sites were in Static, and caused a gap in the reconfigurable region).
- Allows reconfiguration of other clocking resources like the MMCM and PLL.
- Allows reconfiguration of other I/O logic sites such as BITSLICE and BITSLICE_CONTROL.

Regardless of whether or not the I/O usage or characteristics change during reconfiguration, the entire bank is reconfigured. During reconfiguration, all I/O in the banks defined by the RP Pblock is held with the dedicated global tri-state (GTS) signal, which is released at the end of reconfiguration.

If the RM contains an MMCM or PLL component, the size of the partial bitstream is at its smallest when the lock cycle of these components are set to "no wait." Similarly, IO with DCI matching requirements have minimal bitstream sizes when the lock cycle is set to “no wait.” Set these options using this commands:

```
set_property BITSTREAM.STARTUP.LCK_CYCLE NoWait [current_design]
set_property BITSTREAM.STARTUP.MATCH_CYCLE NoWait [current_design]
```

During the Dynamic Function eXchange flow, RMs are carved out using `update_design -black_box`. During this command any embedded IO buffers, and the associated constraints, such as `PACKAGE_PIN` and `IOSTANDARD`, are removed. When the black box RP is filled in with a new RM, these IOB constraints need to be reapplied to the design.

Special Considerations for XPIO Usage

When including XPIO within a reconfigurable partition, the reconfigurable module with the greatest amount of usage must be included in the parent configuration. Any XPIO sites that remain unused in that initial configuration are tied to ground to meet silicon requirements. These tie-offs are considered static and remain for subsequent configurations, which means these sites are unavailable for RMs in child configurations. The Vivado tools automatically insert PROHIBIT constraints on unusable sites. Attempts to assign I/Os to these sites result in an error. For example:

```
ERROR: [DRC HDPR-29] Reconfigurable logic illegally placed: Reconfigurable
logic 'design_top/pl_top/instance_1' is placed at site 'IOB_X52Y0' outside
reconfigurable Pblock 'my_dynamic_pblock'.
```

The recommended strategy is to build the worst case (greatest usage), which is declared in the parent configuration so all subsequent RMs have equal or lesser usage. In hardware, partial images can be delivered in any order.

Floorplanning for Versal Devices

As part of improvements to the Versal architecture, the smallest unit that can be reconfigured is much smaller than in previous architectures. The minimum required resources for reconfiguration varies based on the resource type, and are referred to as a Programmable Unit (PU). Many site types have improved PU requirement making granularity of reconfigurable Pblocks significantly improved compared to previous architecture.

TIP: *Although the fundamental building blocks are shown in the following images, in real design scenarios these building blocks are part of a larger collection of resources, creating a comprehensive floorplan for each dynamic region.*

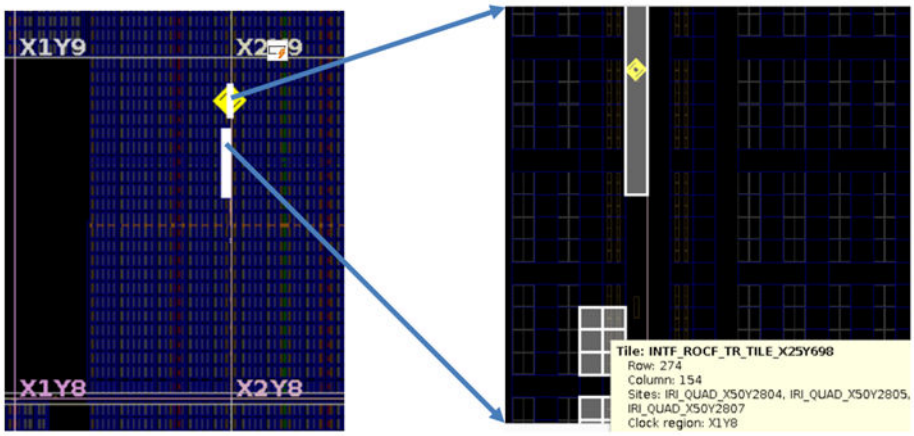
You can use `get_dfx_footprint -pu -of_objects [get_tiles <tile_name>]` to get the PU of any tile in a Versal device. For information, enter `get_dfx_footprint -help`. The PUs are *not* design dependent. You can load any device with `link_design -part <device_name>` or create an I/O planning project and use the `get_dfx_footprint` command with the `-pu` switch to get the PU of any tile. Following are the details provided for each site type.

Programmable Logic (PL) NoC NMU and NSU

The PU is the corresponding NOC_NMU or NOC_NSU tile and includes 38 INTF and the shared INT tiles.

***Note*:** The platform management controller (PMC) NOC tiles in Versal SSI technology devices are not reconfigurable and must be part of the static region.

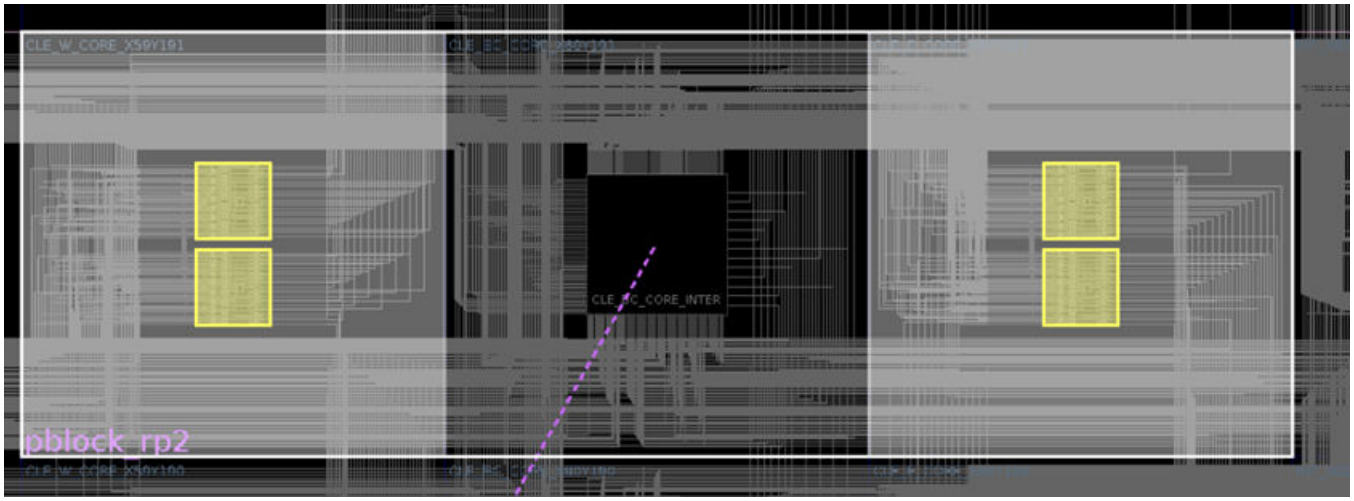
Figure 108: **PL NoC NMU and NSU**



CLE

Two adjacent CLE tiles share a routing resource (interconnect tile). The PU is the two CLE tiles (four SLICE sites) with shared interconnect.

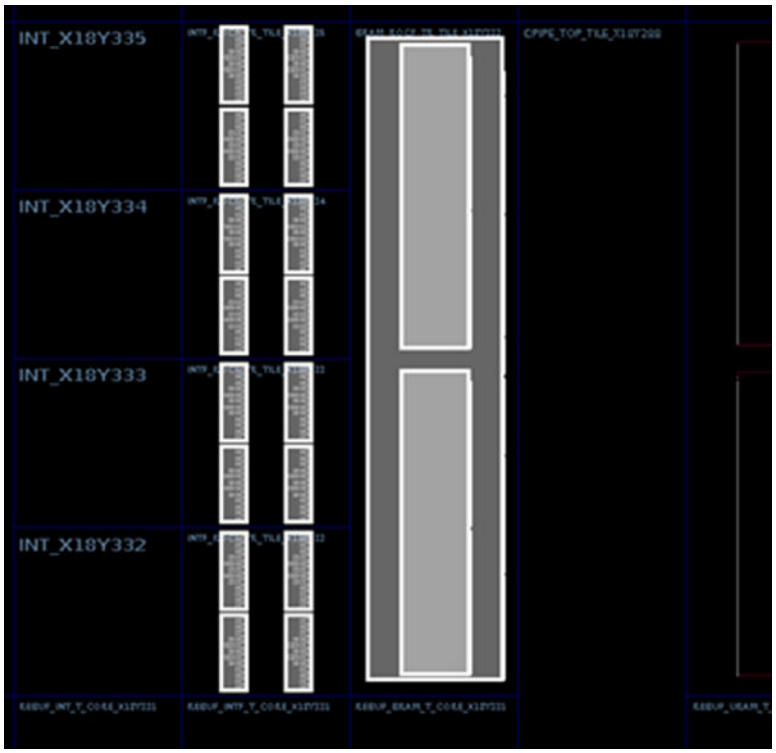
Figure 109: **CLE PU**



Block RAM

The PU is the corresponding block RAM tile. One block RAM tile includes two RAMB18s and one RAMB36. Adjacent INTF and INT tiles, including IRI_QUAD_EVEN and IRI_QUAD_ODD, are automatically pulled into the routing footprint if it is not covered by the Pblock. Unlike previous architectures, adjacent CLE sites are not part of the block RAM PU.

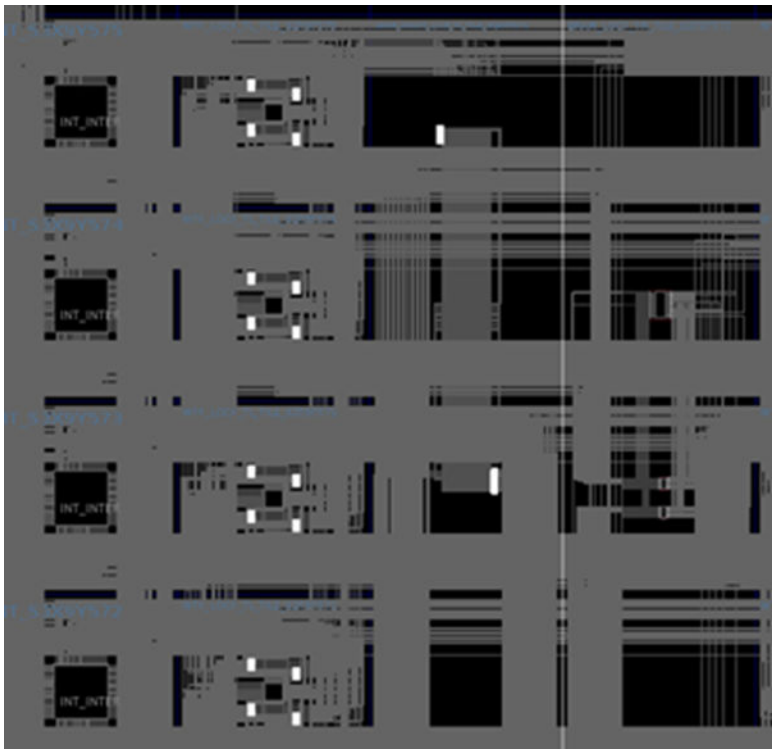
Figure 110: **Block RAM PU: RAMB18s and RAMB36 of One Block RAM Tile**



URAM

The URAM PU includes 1 URAM288, 1 URAM_CAS_DLY, IRI_QUAD_EVEN, and IRI_QUAD_ODD. The adjacent INTF and INT tiles are automatically pulled into the routing footprint if it is not covered by the Pblock.

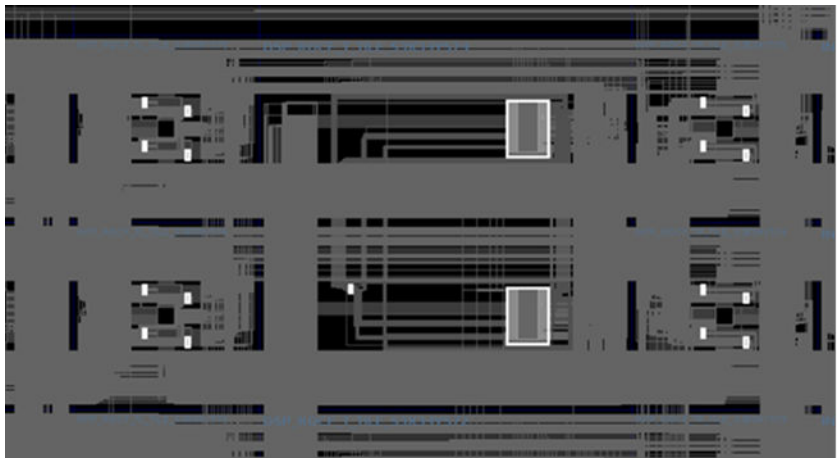
Figure 111: **URAM PU: URAM Tile**



DSP

The PU is the corresponding DSP tile. One DSP tile includes two DSP sites: DSP58_PRIMARY, DSP58_CPLX and IRI_QUAD_EVEN, IRI_QUAD_ODD.

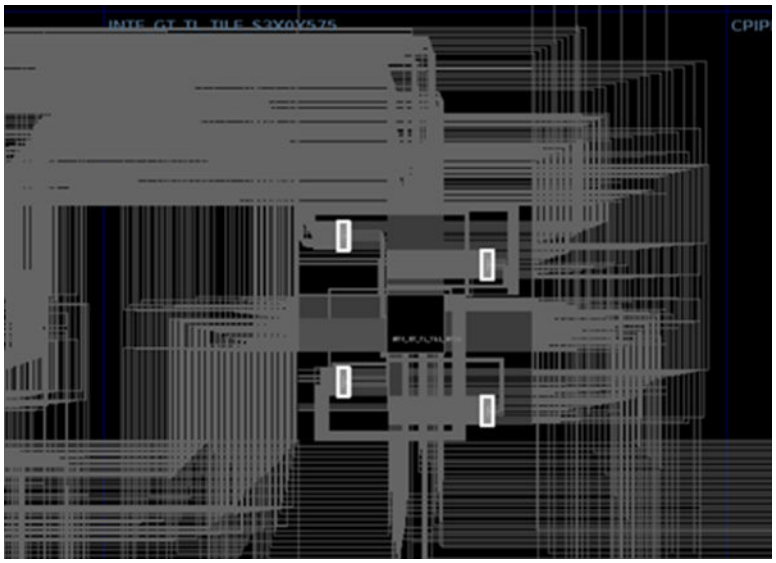
Figure 112: **DSP PU: DSP Tile**



IRI_QUAD (ODD/EVEN)

The PU is the corresponding INTF_ROCF_TL_TILE. One tile includes four IRI Quads. The INTF at the center of the IRI quads is automatically pulled into the routing footprint. Although IRI_QUADs are user range-able, the adjacent IRI_QUADs of the RP Pblock are automatically pulled into the routing footprint, because the expanded routing footprint is always a two INT tile expansion.

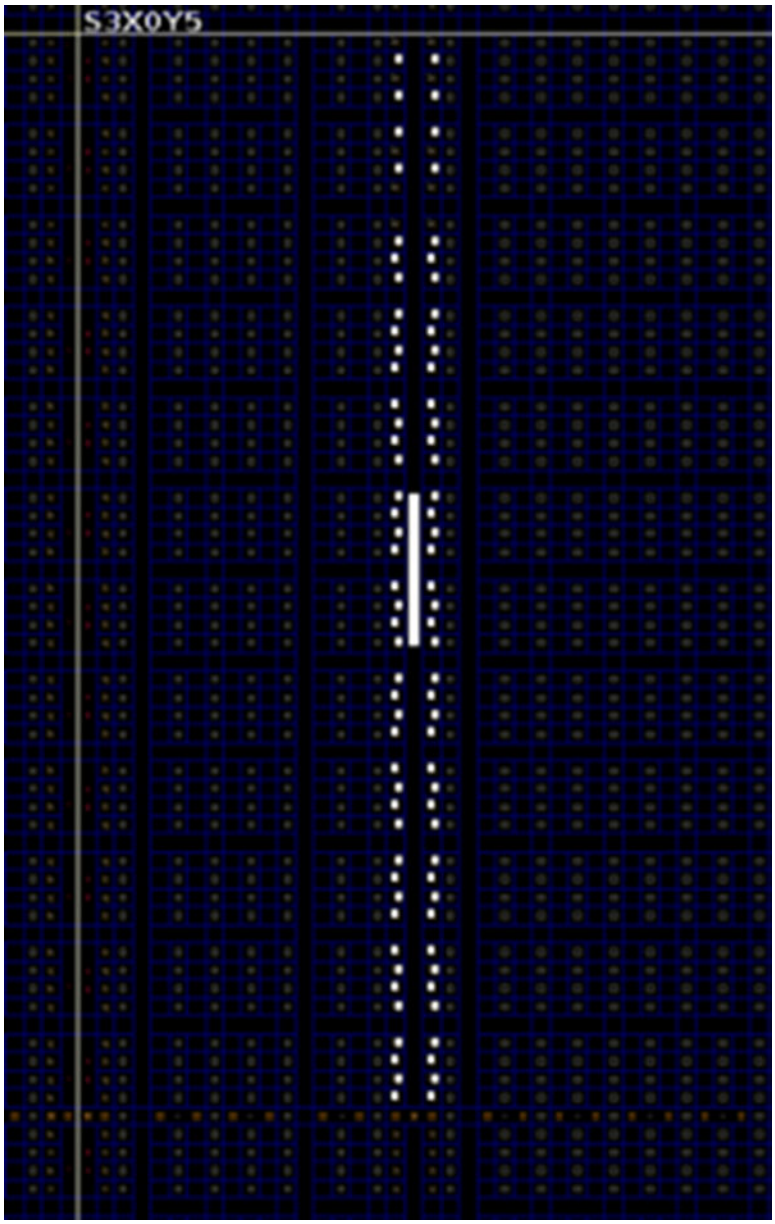
Figure 113: **IMUX Register Interface Quad: PU is INT_ROCF_TL Tile**



PCIe

The PU is the IRI_QUAD_EVEN, IRI_QUAD_ODD, and PCIE50. The adjacent INTF tiles are automatically included in the routing footprint of the reconfigurable Pblock.

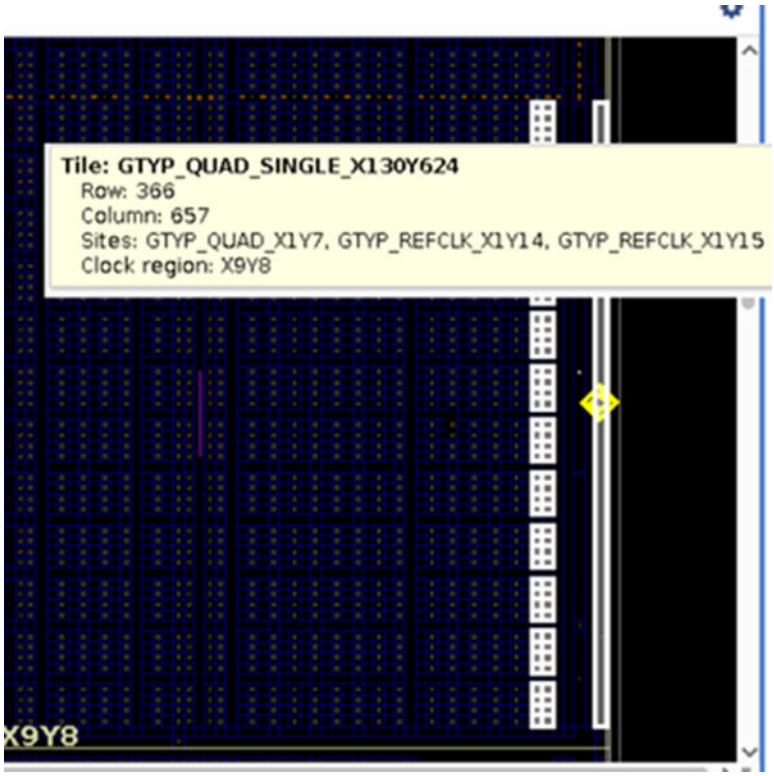
Figure 114: **PCIe PU is PCIe Tile**



GTY_QUAD

The PU is the corresponding GTY_QUAD_SINGLE tile. Sites included in the tile are GTY_QUAD, GTY_REFCLK, and IRI_QUAD. The adjacent INTF_GT tiles are automatically pulled into the routing footprint of the reconfigurable Pblock.

Figure 115: **GTY_QUAD PU is GTY_QUAD Tile**

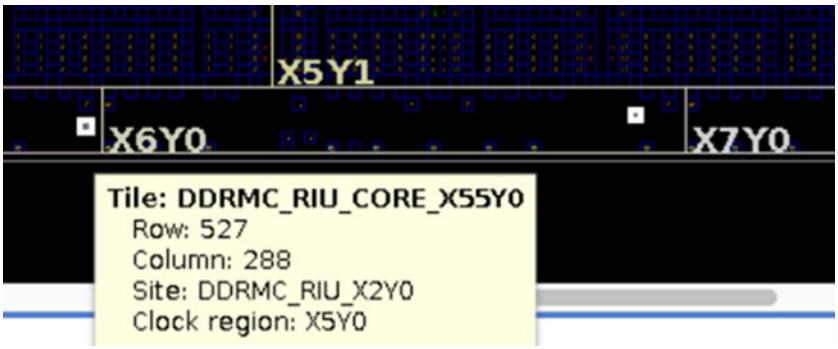


DCMAC and DDRMC_RUI

The PUs are the corresponding DDRMC_DMC_CORE and DDRMC_RIU_CORE tiles respectively. For information, run the following command:

```
get_dfx_footprint -pu -of_objects [get_tiles -of_objects [get_sites
DDRMC_RIU_X2Y0]]
DDRMC_DMC_CORE_X65Y0 DDRMC_RIU_CORE_X55Y0
```

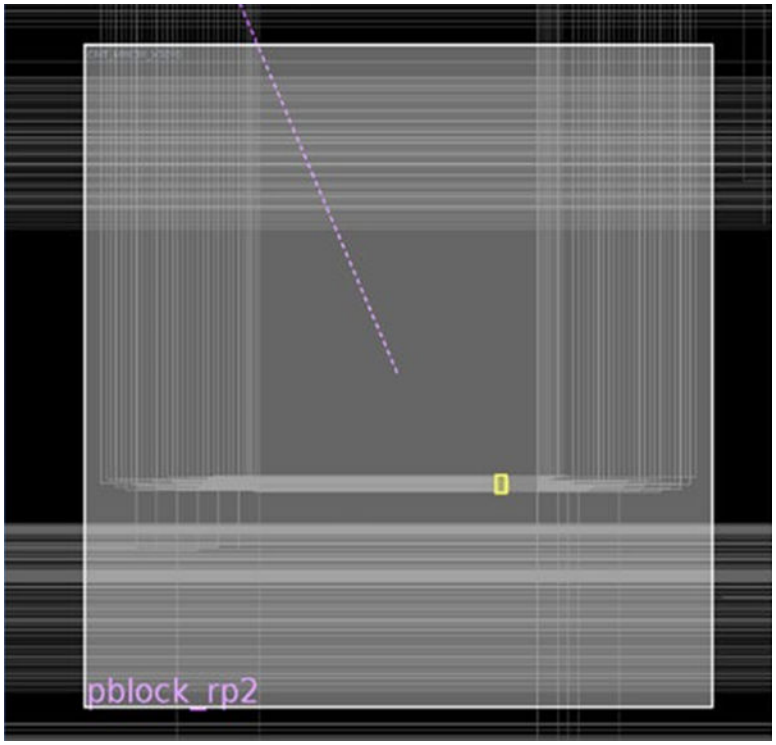
Figure 116: **DDRMC PU**



MMCM

The PU is a corresponding CMT_MMCM tile.

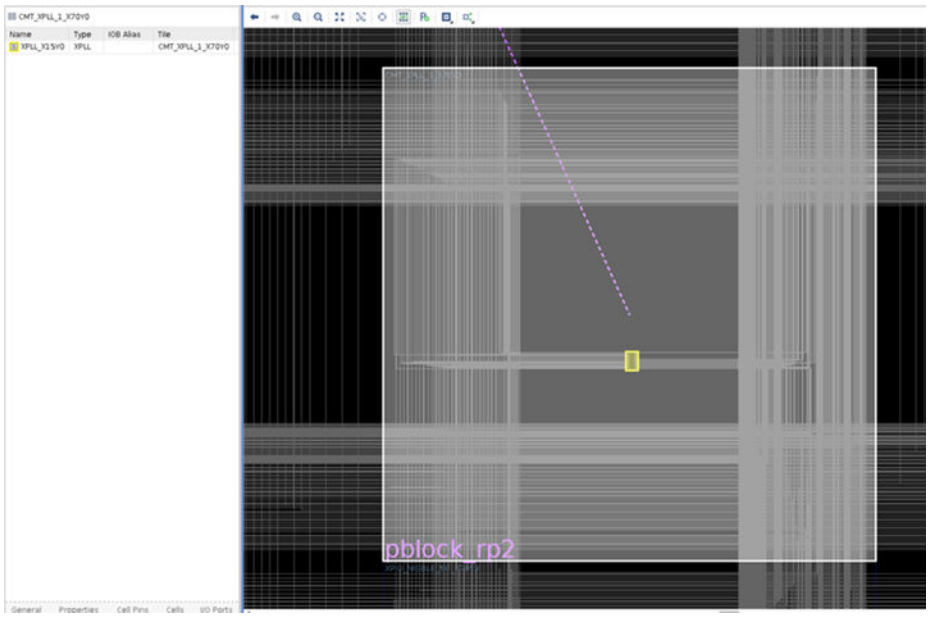
Figure 117: **MMCM PU is CMT_MMCM Tile**



XPLL

The PU is the corresponding CMT_XPLL tile.

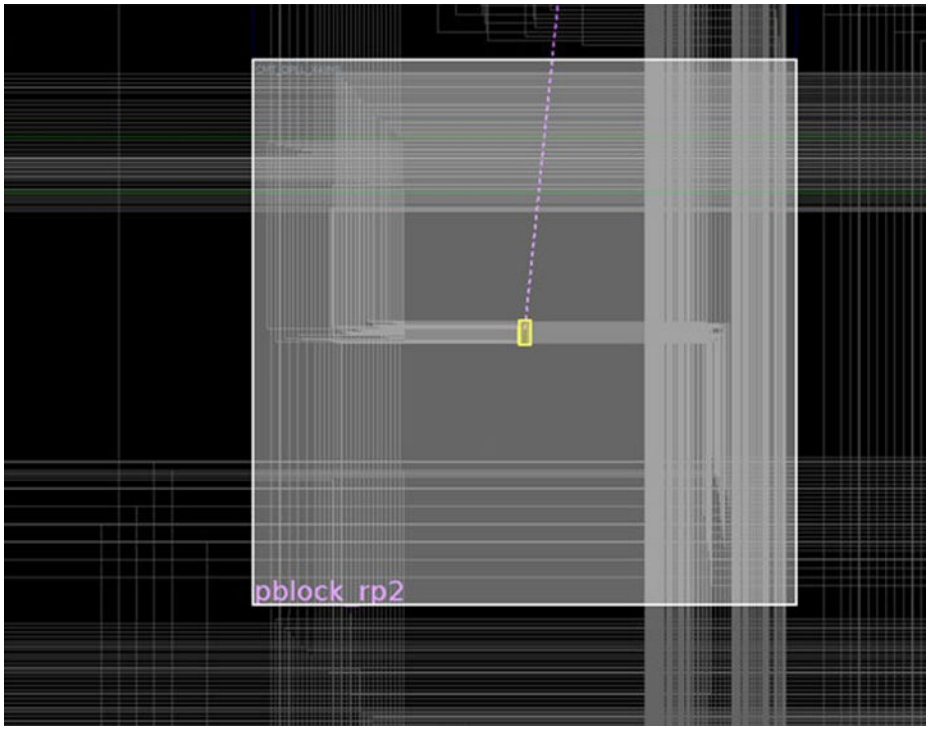
Figure 118: **XPLL PU is CMT_XPLL Tile**



DPLL

The PU is the corresponding CMT_DPLL tile.

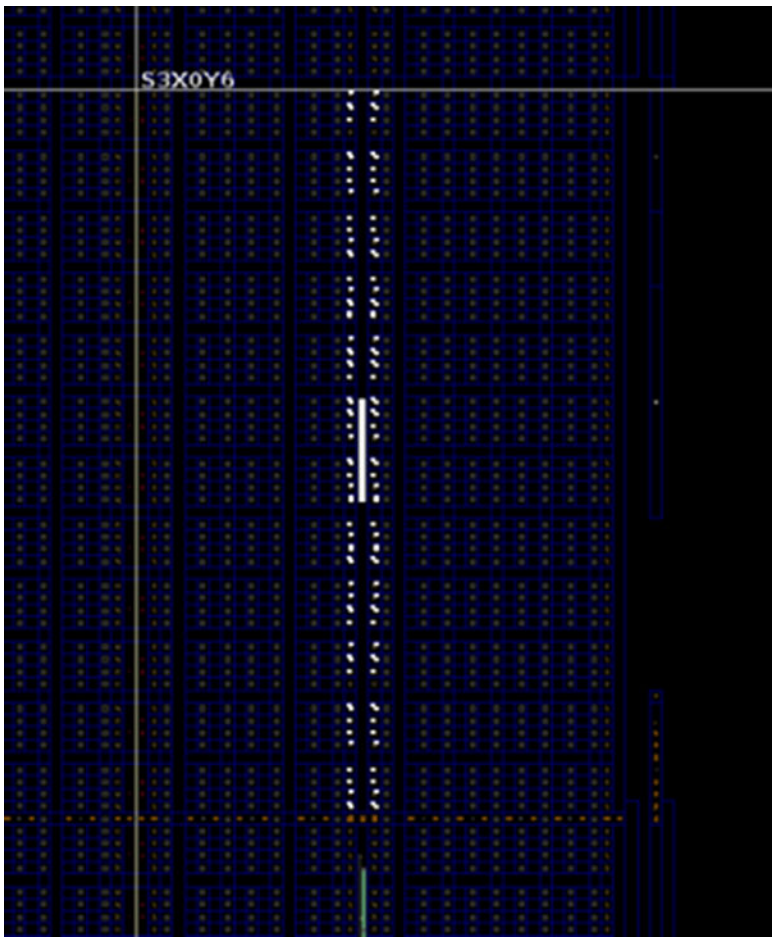
Figure 119: **DPLL PU is CMT_DPLL Tile**



MRMAC

The PU includes the MRMAC_BOT_TILE and the IRI_QUAD_EVEN and IRI_QUAD_ODD sites. The adjacent INT and INTF tiles are automatically included in the routing footprint of the reconfigurable Pblock.

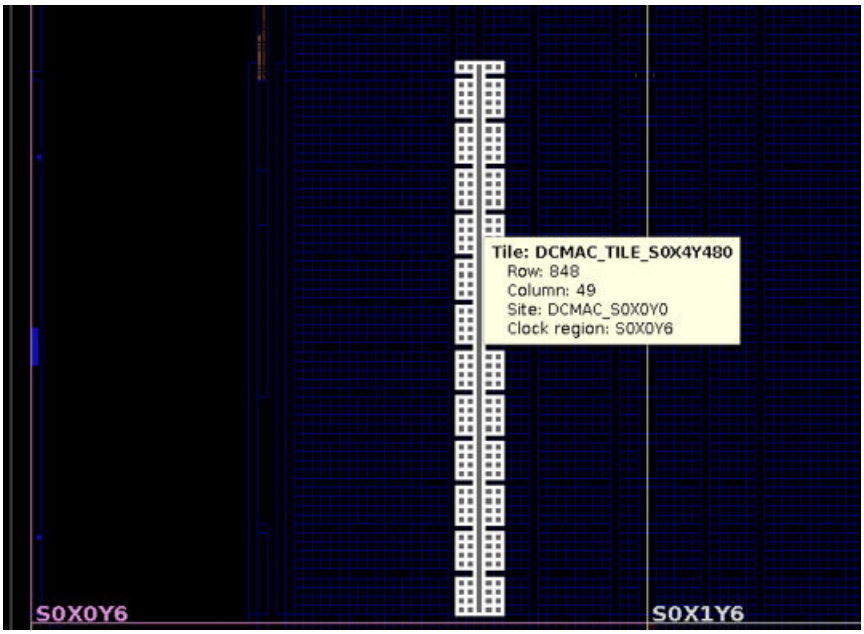
Figure 120: **MRMAC PU**



DRMAC

The PU corresponds to the associated DCMAC tile. If the Pblock does not cover it, adjacent INT and INTF tiles are automatically included in the routing footprint.

Figure 121: **DCMAC PU**



BUFG_FABRIC, BUFG_PS and GCLK_DELAY

These three site types are included in same CLK_VNOC tile. BUFG_PS is present only in the VNOC column adjacent to CIPS. Other VNOC tiles include only BUFG_FABRIC and GCLK_DELAY. The PU requirement is the CLK_VNOC tile and IRI_QUADS.

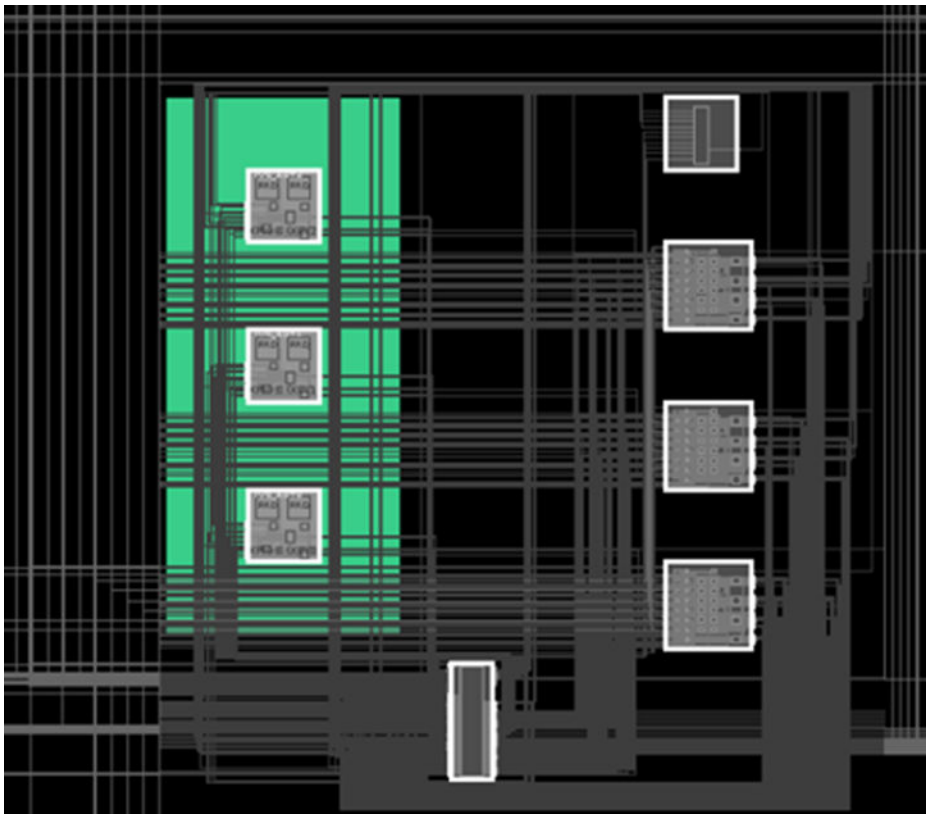
Figure 122: **BUFG_FABRIC, BUFG_PS, and GCLK_DELAY Shares Same Tile As PU**



XPHY, XPIO, and IOB

An I/O bank in Versal devices cannot be shared by static and reconfigurable partitions. All XPIO_NIBBLE tiles of one I/O bank must be used by one partition only.

Figure 123: **XPHY, XPIO and IO PU**



BUFG_GT, BUFG_GT_SYNC, and GCLK_DELAY

The CLK_GT tile and the IRI_QUAD sites are included in the PU.

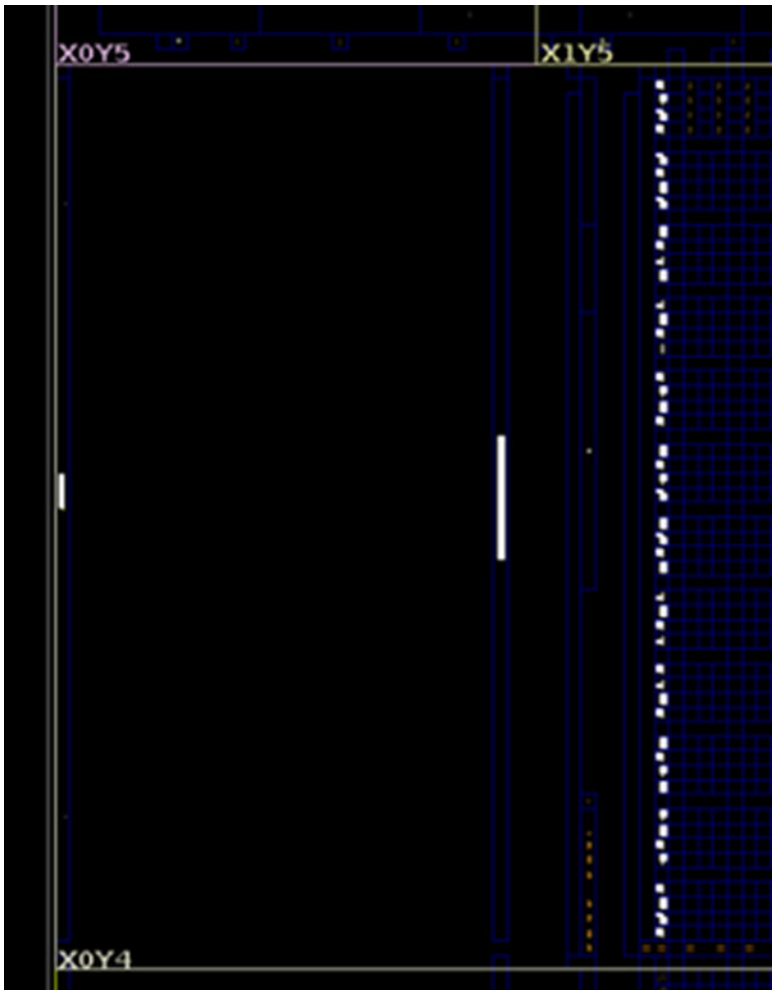
Figure 124: **BUFG_GT, BUFG_GT_SYNC Share Same PU: CLK_GT Tile**



XPIPE_QUAD

The XPIPE_QUAD tile is the PU,which includes IRI_QUAD_EVEN, IRI_QUAD_ODD,GTY_QUAD, GTY_REF_CLK, and XPIPE_QUAD.

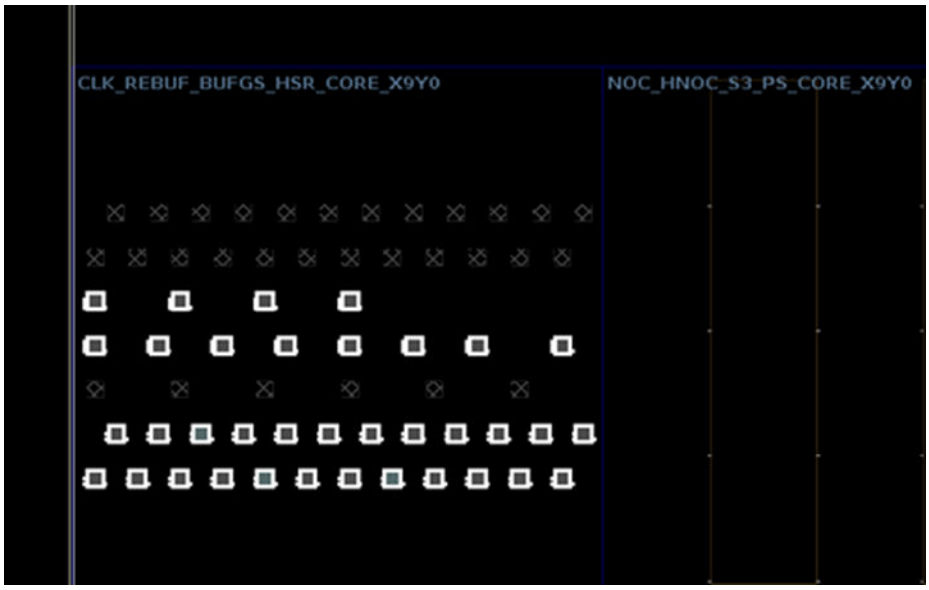
Figure 125: **XPIPE_QUAD PU**



BUFGCE, BUFGCTRL, and BUFGCE_DIV

For the BUFGCE elements in HSR, the PU is the CLK_REBUF_BUFGS_HSR_CORE tile.

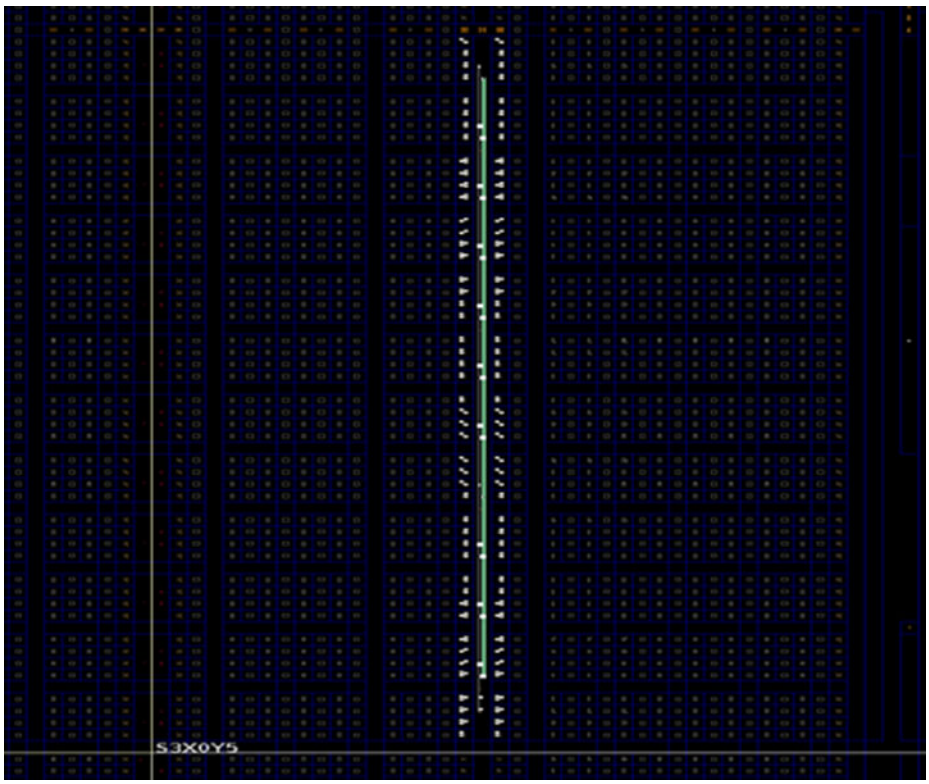
Figure 126: **Clocking Buffers in HSR Has CLK_REBUF_BUFG_HSR_CORE Tile As PU**



BUFGCE_HDIO, HDIO_BIAS, HDIO_LOGIC, and IOB

For these sites in HDIO, the PU is the HDIO_TILE tile and the IRI_QUADs.

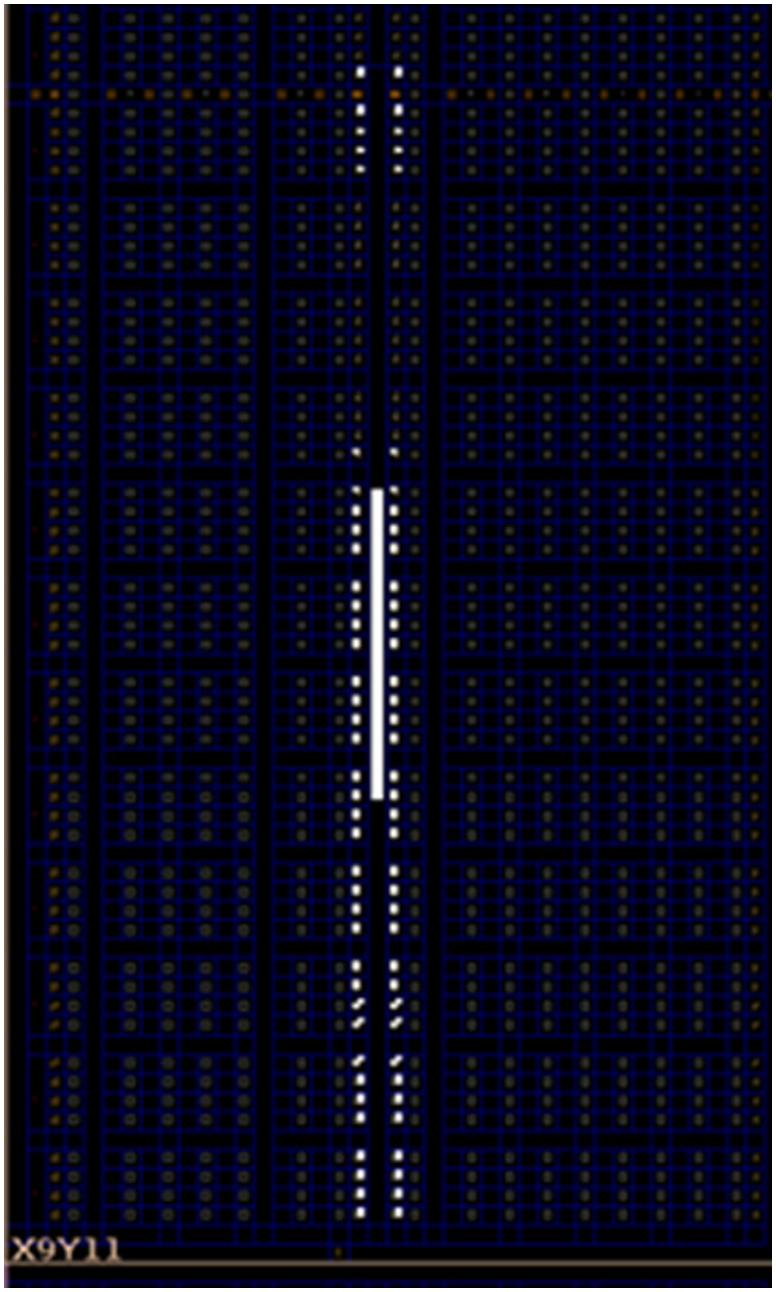
Figure 127: **Clock Buffers and IOB in HDIO Bank Share Same PU: HDIO_TILE**



High-speed Channelized Cryptography Engines (HSC)

The PU is the corresponding HSC_TILE tile and IRI_QUADs.

Figure 128: **HSC PU: HSC Tile**



Related Information

Floorplanning Rules for Clocks Inside an RP

Floorplanning with SNAPPING_MODE

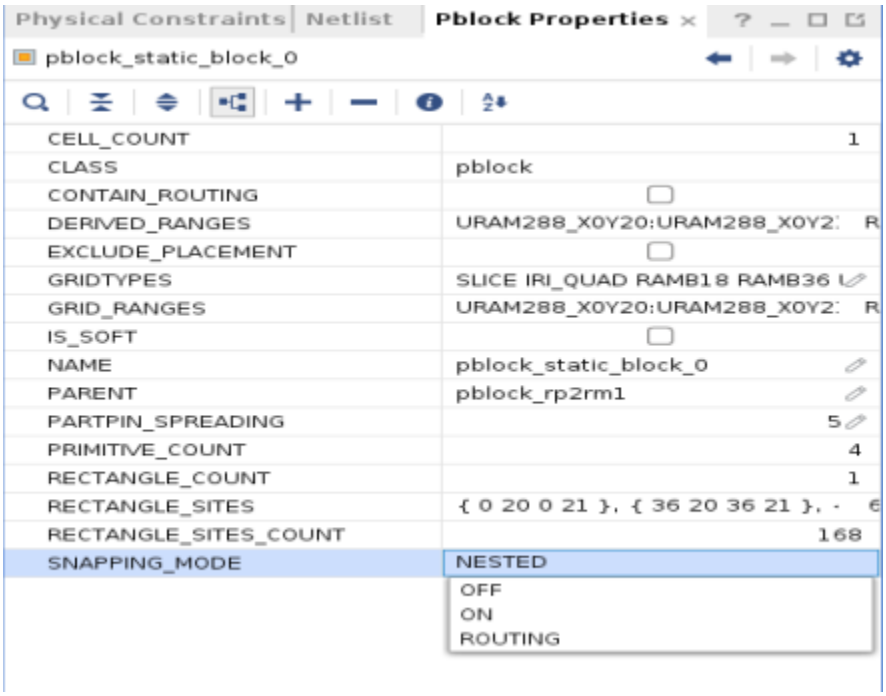
The Pblock SNAPPING_MODE property automatically resizes Pblocks to ensure the Pblocks align to programmable unit boundaries. The value of ON is selected by default, and a set of derived ranges are generated for the implementation tools to use. For Versal devices, a value of ROUTING is equivalent to ON.

TIP: *When adjusting Pblocks in the Vivado IDE Device view, you can temporarily set SNAPPING_MODE to OFF to prevent the tools from adjusting Pblocks with each mouse click. Make all adjustments to confirm the Pblock edges, and then set SNAPPING_MODE back to ON.*

For Versal devices, you can use the SNAPPING_MODE = NESTED property to ensure that child Pblocks are contained within the parent Pblock. If the parent Pblock is resized, the child Pblock with SNAPPING_MODE = NESTED is automatically resized to ensure all derived range sites are within the parent Pblock. The following figure shows the child Pblock Properties window with SNAPPING_MODE set to NESTED.

***Note*:** By default, the child Pblock automatically includes the SNAPPING_MODE = NESTED property when the parent Pblocks includes the SNAPPING_MODE= ON property.

Figure 129: **Snapping Mode in the Pblock Properties Window**



UltraRAM Behavior

As with a full device configuration, you can initialize UltraRAM memory with user-defined values during partial reconfiguration. This differs from prior architectures, which did not offer UltraRAM initialization.

Clock Region Sharing Between Multiple Reconfigurable Pblocks

Clocking resources in an RCLK row can be shared by two reconfigurable partitions. Hence, it is possible to share a clock region between two reconfigurable partitions (one RP above the RCLK row and one RP below the RCLK row). You can share more than two reconfigurable partitions in a clock region if only at most two of them have internal clocking resources. Sharing a clock region between more than two reconfigurable partitions, where all of them have clocking resources results in a DRC error. The error happens because clock routing expansion of all RPs tries to pull in clocking resources from the same RCLK row of that clock region which is not supported. RCLK row clock tracks sharing is allowed only with two RPs and anything more than that will result in DRC error.

In the following figure, two reconfigurable partitions `rp1` and `rp2` share the same clock region X3Y2. Both RPs have internal clock net (yellow and magenta color highlighted). Both the clocks share the same RCLK row.

Figure 130: **Sharing Clock Region Between Multiple Reconfigurable Pblocks**

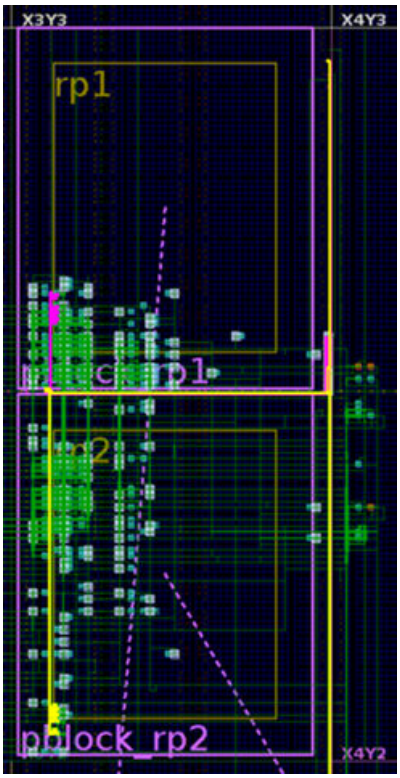


Figure 131: **Schematic View of Two Reconfigurable Modules With An Internal Clock**

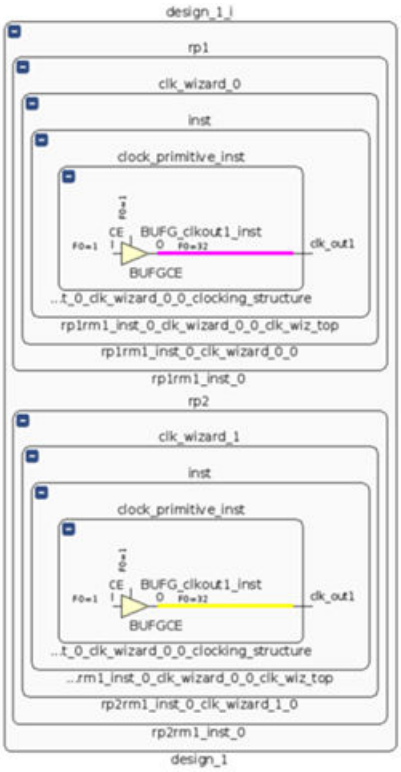
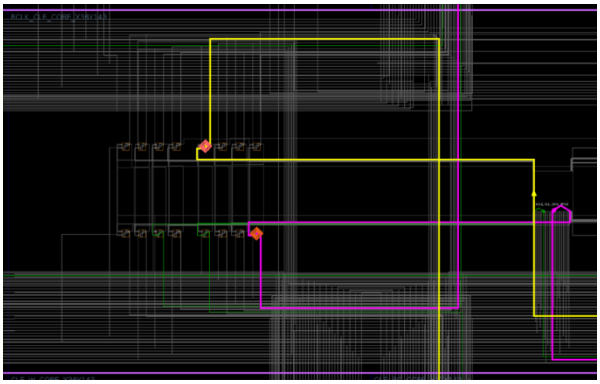


Figure 132: **Magnified View of Device. Top Half of RCLK is Used by One Partition's Clock Net And Bottom Half is Used By Other Partition's Clock Net**



Floorplanning Visualization

In the DFX flow for Versal devices, you can visualize the placement and routing footprints using the `get_dfx_footprint` command, which offers the following benefits:

- Faster visualization of DFX components (by up to tens of minutes when compared with the `hd_visual` commands)

- Floorplanning DRC analysis
- Guide generation for supported floorplanning of multi-RPs and RPs with disjoint Pblocks

The `get_dfx_footprint` command returns tile, site, and cell objects, which can be used in `create_pblock` and `resize_pblock` commands to create valid reconfigurable Pblocks. You must define the reconfigurable Pblock before executing the command. Enter `get_dfx_footprint -help` in the Vivado Design Suite Tcl Console to get details of the command.

The following table lists the supported command arguments and usage for the `get_dfx_footprint` command.

Table 13: **get_dfx_footprint Command Options**

Option	Usage
`-of_objects`	Returns objects associated with specified reconfigurable module or device tiles and accepts one object as input. ***Note***: This option is dependent on the argument passed. Most of the arguments take the reconfigurable module (RM) cell name as a valid value for `-of_objects`.
`-place`	Returns the tiles that are included in the placement footprint of a reconfigurable module specified by `-of_objects`.
`-route`	Returns the tiles that are included in the routing footprint of a reconfigurable module specified by `-of_objects`.
`-pu`	Returns the tiles included in programmable unit (PU) that are the minimum required resource set for partial reconfiguration for the tile specified by `-of_objects`.
`-frame`	Returns the tiles included in a frame of the given tile specified by `-of_objects`. A reconfigurable frame is the smallest sized physical region that can be reconfigured and aligns with clock region boundaries.
`-half_frame`	Returns the tiles included in a half frame of the given tile specified by `-of_objects`. A reconfigurable frame is the smallest sized physical region that can be reconfigured and aligns with clock region boundaries. A half frame aligns with half of the clock region boundary.

Table 13: **get_dfx_footprint Command Options** *(cont'd)*

Option	Usage
`-site_type`	Returns the sites types specified for the value and takes the reconfigurable module cell name in `-of_objects`. Following are the valid values: • `* hsr`: Returns the valid sites of the horizontal super region that needs to be part of the reconfigurable Pblock of the reconfigurable- module specified by `-of_objects`. • `* fsr`: Returns the valid sites of the fabric logic region that needs to be part of the reconfigurable Pblock of the reconfigurable module specified by `-of_objects`. • `* fsr_hsr`: Returns the valid sites of horizontal super region and adjoined fabric logic region that needs to be part of the reconfigurable Pblock of the reconfigurable module specified by `-of_objects`. The clock sources needs to be placed in the HSR, and the clock control logic needs to be placed in this adjoined fabric logic region in case of a disjoint Pblock. ***Note***: The `-site_type` option is only used for disjoint Pblocks. This option is disabled for non-disjoint Pblocks and does not return any objects.
`-cell_type`	Returns the cell types specified for the value. The cells are associated with the reconfigurable module specified by `-of_objects`. Following are the valid values: • `* non_clock`: Returns list of cells that need to be placed in the fabric logic region. • `* clock`: Returns list of clock type cells that need to be placed in the HSR. • `* clock_control`: Returns list of cells that are part of the clock control logic and need to be placed in the fabric logic region above the HSR where the source clock cells are placed. ***Note***: The `-cell_type` option is only used for disjoint Pblocks. This option is disabled for non-disjoint Pblocks and does not return any objects.
`-overlap`	Returns the overlapping tiles of the routing footprint for the reconfigurable module specified by `-of_objects`. This argument can be used to get the tiles that are part of multiple reconfigurable Pblocks in a DFX design that might lead to DRC errors.
`-shared`	Returns the tiles for the reconfigurable module specified by `-of_objects` that can be shared with another reconfigurable Pblock. This lists all shared clocking tiles of the specified RM cell.
`-bli_tiles`	Returns the boundary logic interface (BLI) tile connected to given HSR tile specified by `-of_objects`. BLI tiles are register stages available for signal going in and out of FSR to and from HSR resources.
`-illegal_nodes`	Returns the illegal nodes of a RP module specified by `-of_objects`, or illegal nodes prohibited for static if no cell is specified. Illegal nodes cannot be used in routing of RM or static net.

Table 13: **get_dfx_footprint Command Options** *(cont'd)*

Option	Usage
`-illegal_clock_nodes`	Returns the illegal clock nodes of a clock net specified by `-of_objects`. Illegal clock nodes cannot be used in clock routing. These nodes are subset of illegal nodes.
`-snapped_tiles`	Returns the list of tiles added to derived ranges by snapping and are not present in XDC constraints. With `SNAPPING_MODE` on, the tool includes PU of tiles in the Pblock derived ranges that are not included in XDC.

`-is_reconfigurable`	Returns 1 for the tile specified by `-of_objects` is reconfigurable, and 0 if the tile is non-reconfigurable. This switch is not design-dependent.
	The following examples report whether the object specified is reconfigurable or not:
	`get_dfx_footprint -is_reconfigurable -of_objects [get_tiles CLE_W_CORE_X0Y623]`
	`1`
	`get_dfx_footprint -is_reconfigurable -of_objects [get_tiles -of_objects [get_sites PS9_X0Y0]]`
	`0`

Note: For UltraScale and UltraScale+ devices `-place`, `-route`, and `-illegal_nodes` are supported.

Floorplanning Visualization Examples

The following example gets the programmable unit (PU) of a CLE_E_CORE tile:

```
get_dfx_footprint -pu -of_objects [get_tiles CLE_E_CORE_X27Y4]
```

The following example highlights the overlapping tiles of the RM cell `design_0_i/rp1_rm1`:

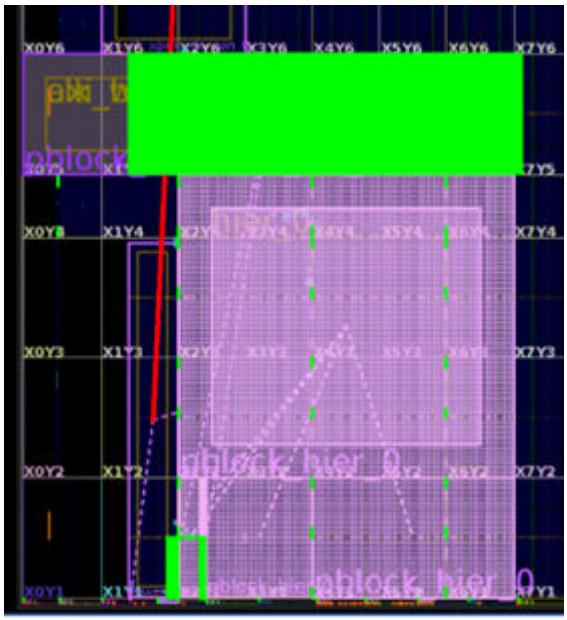
```
highlight_objects -color yellow [get_dfx_footprint -overlap -of_objects
[get_cells design_0_i/rp1_rm1]]
```

The following example highlights the routing footprint of a disjoint Pblock.

```
highlight_objects -color green [get_dfx_footprint -route -of_objects
[get_cells design_0_i/bramctrl_rm]]
```

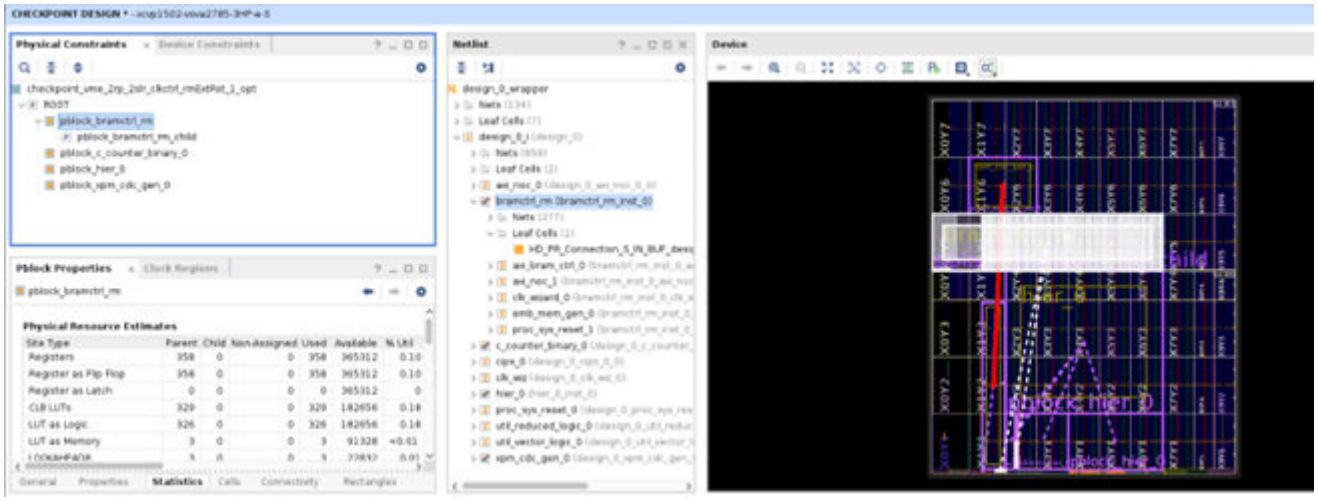
In the following figure, the green highlighted tiles show the routing footprint of a disjoint Pblock.

Figure 133: **Disjoint Pblock Routing Footprint**



You can use the `get_dfx_footprint` command to assign the valid cells to the child Pblock of a disjoint Pblock. In the following example `pblock_bramctrl_rm` is the parent Pblock that is disjoint, and `pblock_bramctrl_rm_child` is the child Pblock. The design has clock control logic that needs to be placed in the extended fabric region near the clock source. The rest of the logic needs to be placed in the remote upper region. The following figure shows the disjoint Pblock Device view and the parent Pblock Properties window, which indicate that RM cells are assigned to the parent Pblock. All non-clock cells must be assigned to child Pblocks.

Figure 134: **Parent Pblock of Disjoint Pblock in Device View and Physical Properties Window**

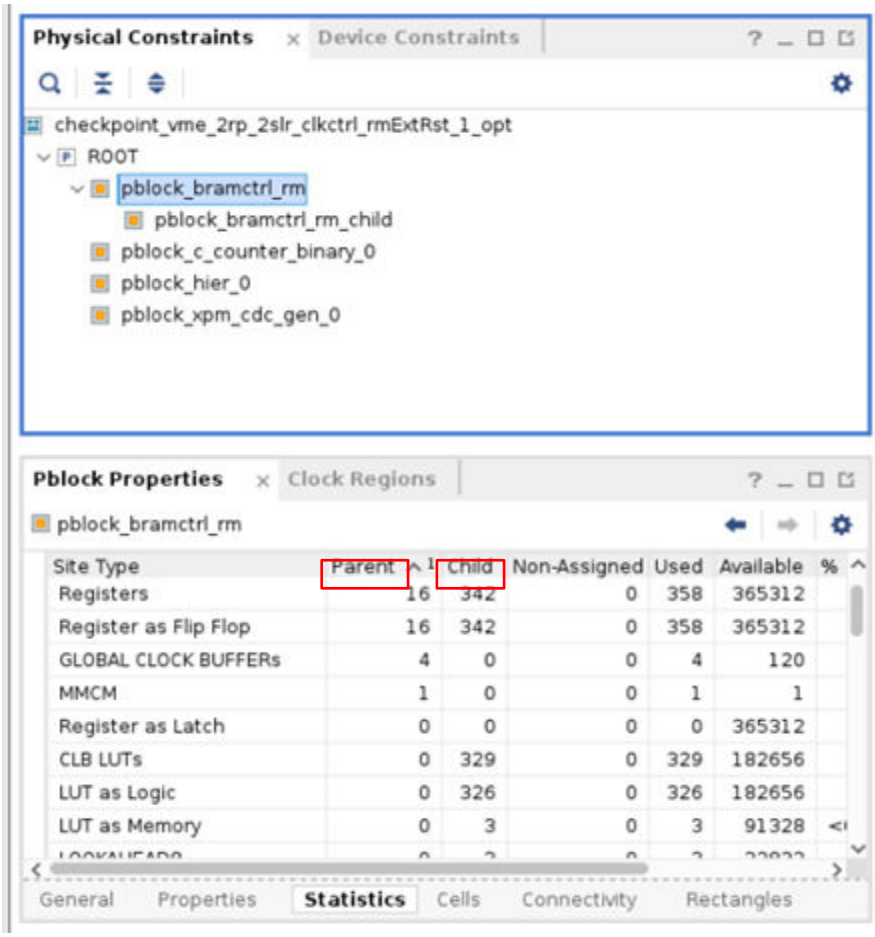


You can use the `get_dfx_footprint` command to fetch all the cells that must be placed in the `pblock_bramctrl_rm_child` child Pblock of the disjoint Pblock. See the following example:

```
set rm_cell [get_cells design_0_i/bramctrl_rm]
    set fsr_tiles [get_dfx_footprint -of $rm_cell -site_type fsr]
    set non_clock_cells [get_dfx_footprint -of $rm_cell -cell_type
non_clock]
    resize_pblock [get_pblock pblock_bramctrl_rm_child] -add $fsr_tiles
    # Add sites and cells to child pblock
    add_cells_to_pblock [get_pblocks rp_child_pblock] $non_clock_cells
-clear_locs
```

After executing these Tcl commands, the non-clock cells of the RM are assigned to the child Pblock, and the clock cells and the clock control logic remain assigned to the parent Pblock. The following figure shows the distribution of the assigned cells between the parent and child Pblocks.

Figure 135: **Distribution of Cells Between Parent and Child Pblocks in Parent Pblock Properties Window**



X28848-111023

BLI Floorplan Alignment

In Versal devices, boundary logic interfaces (BLI) tiles are additional register stages available for signals going in and out of programmable logic (PL) to and from XPIO logic resources. The BLI register stages help optimize the timing of interfaces.

Based on the location, a BLI tile can be used by multiple sites:

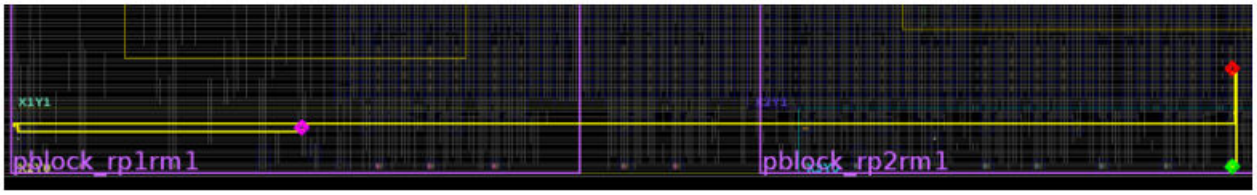
- XPHY and XPIOLOGIC sites in a `XPIO_NIBBLE` tile.
- DDRMC.
- XPLL sites in a `CMT_XPLL` tile.
- DPLL sites in a `CMT_DPLL` tile.
- BUFGCE sites in a `CLK_REBUF_BUFGS_HSR_CORE` tile.
- MMCM sites in a `CMT_MMCM` tile.

Because the BLIs are aligned geometrically to each of the site types that use the BLI, the DFX flow automatically pulls the BLIs based on the tiles in the range of Pblocks. Following are some of the rules associated with adding BLI ranges to the reconfigurable Pblock:

- BLI tiles can be independently added to the reconfigurable Pblock range, even though none of the tiles mentioned above are added in the Pblock range.
- If an XPIO tile is ranged into an RP, the connected BLI is pulled into the placement footprint.
- If clocking resources like BUFG or MMCM are ranged in a reconfigurable Pblock, the connected BLI is pulled into the placement footprint by the tool.
- If AIE_PL or AIE_NOC sites are ranged in a reconfigurable Pblock, the connected BLI is pulled into the RP Pblock range automatically.
- If a conflict is observed that breaks the automatic pulling of BLI ranges, a DRC is flagged. This can happen when there are two tiles trying to use the same BLI, but those two tiles happen to be in two separate reconfigurable partitions. The DRC message also provides the resolution to remove the corresponding tile from the Pblock to avoid the conflict.

In the following figure, the CMT_MMCM tile (marked in magenta) is present in pblock_rp1rm1 and the XPIO tile (marked in green) is present in pblock_rp2rm1. The BLI tile (marked in red) is added in the placement footprint of both the RPs due to the HSR routing (highlighted in yellow).

Figure 136: **BLI Sharing and Floorplan Alignment**



This triggers the following DRC error, because the BLI is being shared by tiles of two different RPs.

```
HDPR-66- Error
HD.RECONFIGURABLE Pblock 'pblock_rp1rm1' and Pblock 'pblock_rp2rm1'
overlap.  Please re-floorplan Pblocks to ensure that reconfigurable
pblocks don't have overlap with other rm pblocks.To get the overlapping
tiles, use 'get_dfx_footprint -source [get_pblocks pblock_rp1rm1] -conflict
[get_pblocks pblock_rp2rm1] -ranged'.pblock_rp2rm1

HDPR-156- Error
Reconfigurable pblock pblock_rp1rm1 has overlapping tile in its definition
or in placement expansion. Please check other DRCs for overlapping sites/
tiles or use 'get_dfx_footprint -overlap -of_objects [get_cells design_1_i/
rp1rm1_0]' to get the list of all overlapping tiles.
```

You can remove one of the shared tiles to resolve the issue reported by the DRC and avoid having the same BLI shared by two RPs. You must resize the Pblocks to avoid overlaps. One resolution is to remove the conflicting CMT_MMCM tile from pblock_rp1rm1 and add the adjacent CMT_MMCM tile:

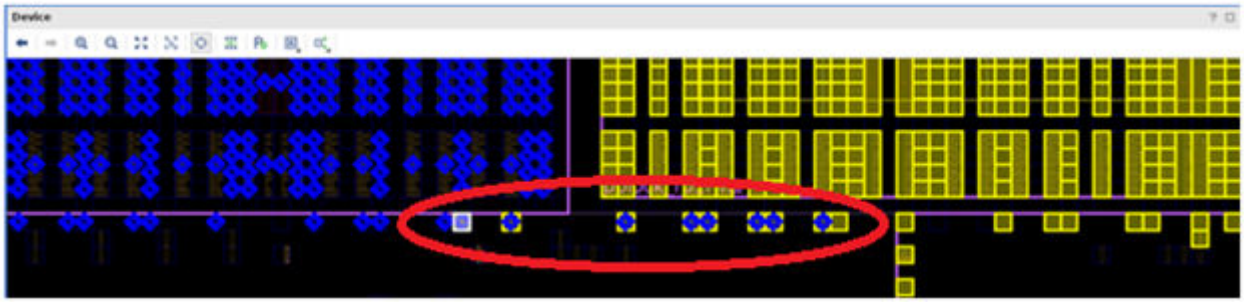
```
resize_pblock pblock_rp1rm1 -remove [get_sites -of_objects [get_tiles
CMT_MMCM_X14Y0]]

resize_pblock pblock_rp1rm1 -add [get_sites -of_objects [get_tiles
CMT_MMCM_X4Y0]]
```

To visualize the overlapping tiles, you can use the following commands to highlight and mark the placement footprint of the two RPs:

```
highlight_objects -color yellow [get_dfx_footprint -place -of_objects
[get_cells <RP1_cellname>]]

mark_objects -color blue [get_dfx_footprint -place -of_objects [get_cells
<RP2_cellname>]]
```

In the following example, the overlapping tiles are yellow and blue (shown in the red circle).

Figure 137: **Overlapping Tiles Example**



Expanded Routing

Similar to the UltraScale+ architecture, the expansion of the routing area also happens in Versal devices for logical signals. The routing footprint of a reconfigurable Pblock can be fetched by using the `get_dfx_footprint -route -of_objects <rm cell inst name>` command that returns the list of tiles in the routing footprint. For clock routing, the necessary clock routing tiles (for example, CLK_VNOC) of the RP Pblock are automatically pulled into the routing footprint.

WARNING! *The Expanded Routing feature must not be disabled for Versal devices to ensure the highest possibility of routing success.*

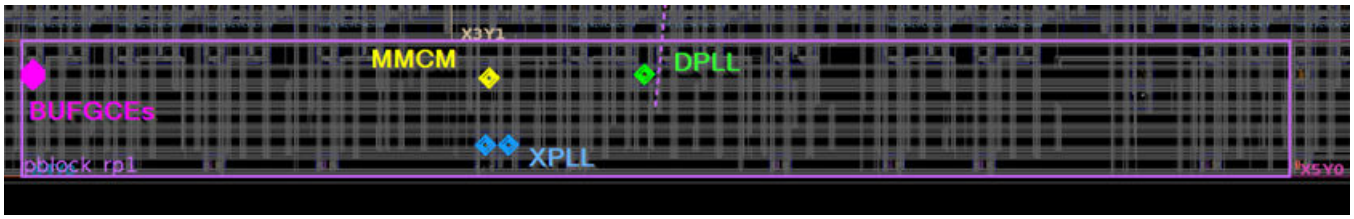
Related Information

Expansion of CONTAIN_ROUTING Area

Clocking Resources

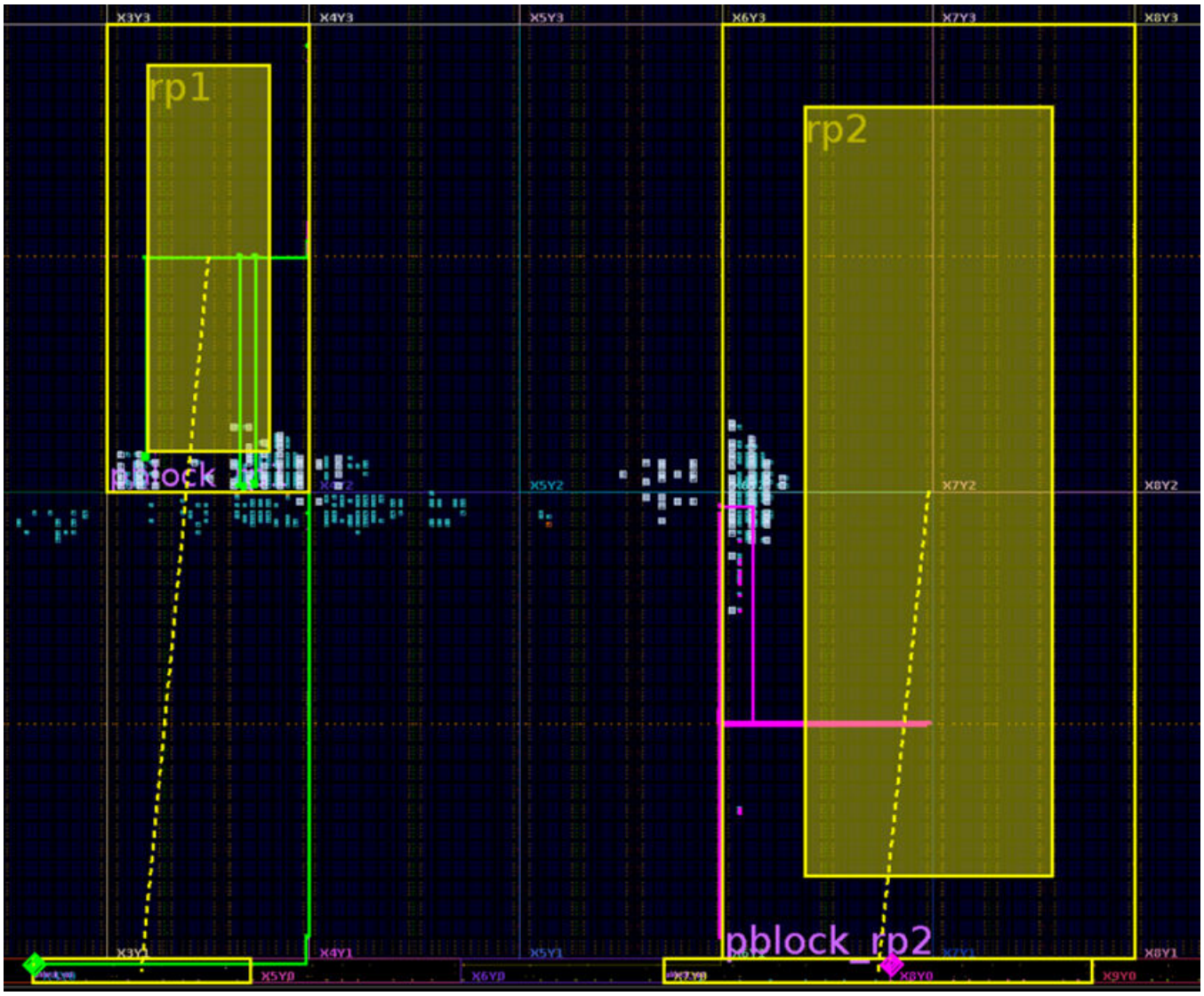
In Versal, the majority of clocking resources appear in the bottom horizontal super row (HSR). Since most of the clocking resources are located at bottom row of the device, users can have an island for reconfigurable Pblock in the bottom HSR if they need internal clocking resources inside the reconfigurable partition. For example, in the following figure `rp1` Pblock is split into two rectangles, one for the placement of logic and other for the placement of clocking resources.

Figure 138: **Clocking Resources**



Clock Routing expansion automatically picks up required clock tiles for routing even though clocking resources are in a separate island of the Pblock.

Figure 139: **Clock Routing Expansion**



IMPORTANT! *It is recommended to draw contiguous Pblocks like `rp2`. This avoids static region's island-like placement and eases routability and timing closure.*

Floorplanning Design Rule Checks

The floorplanning design rule checks (DRCs) ensure that floorplanning guidelines for DFX designs were followed. DRC violations might be due to unsupported Pblock shapes or due to overlapping Pblocks caused by footprint expansions.

The overlapping Pblock DRC messages provide the following information:

- Rule that was violated.

- Overlapping tile.
- Reason for the tile to be added to the RP placement, routing, or clocking footprint.
- Suggested resolution of resizing the reconfigurable Pblocks to avoid the overlap.

The DRC messages indicate the reason for the footprint expansion as follows:

- **Programmable Unit (PU):** Expanded to include all the programmable units of the tiles that belong to the RP Pblock.
- **HSR_ROUTING:** Expanded to add BLI tiles required for connecting the HSR tiles (CMT_MMCM, CMT_DPLL, and CMT_REBUF_BUFGS_HSR_CORE) to the fabric.
- **FRAME_ALIGNMENT:** Expanded to include all tiles programmed by the reconfigurable frame to the same Pblock.
- **CLK_REBUF_EXPANSION:** Collected the BUFG tiles on bi-directional clock nodes in HSR region and included the tiles in the expanded RP footprint.
- **PBLOCK_CONSTRAINT:** Collected the list of tiles from user-specified Pblock constraints and added the tiles to the expanded Pblock footprint.

Global Clocking Rules

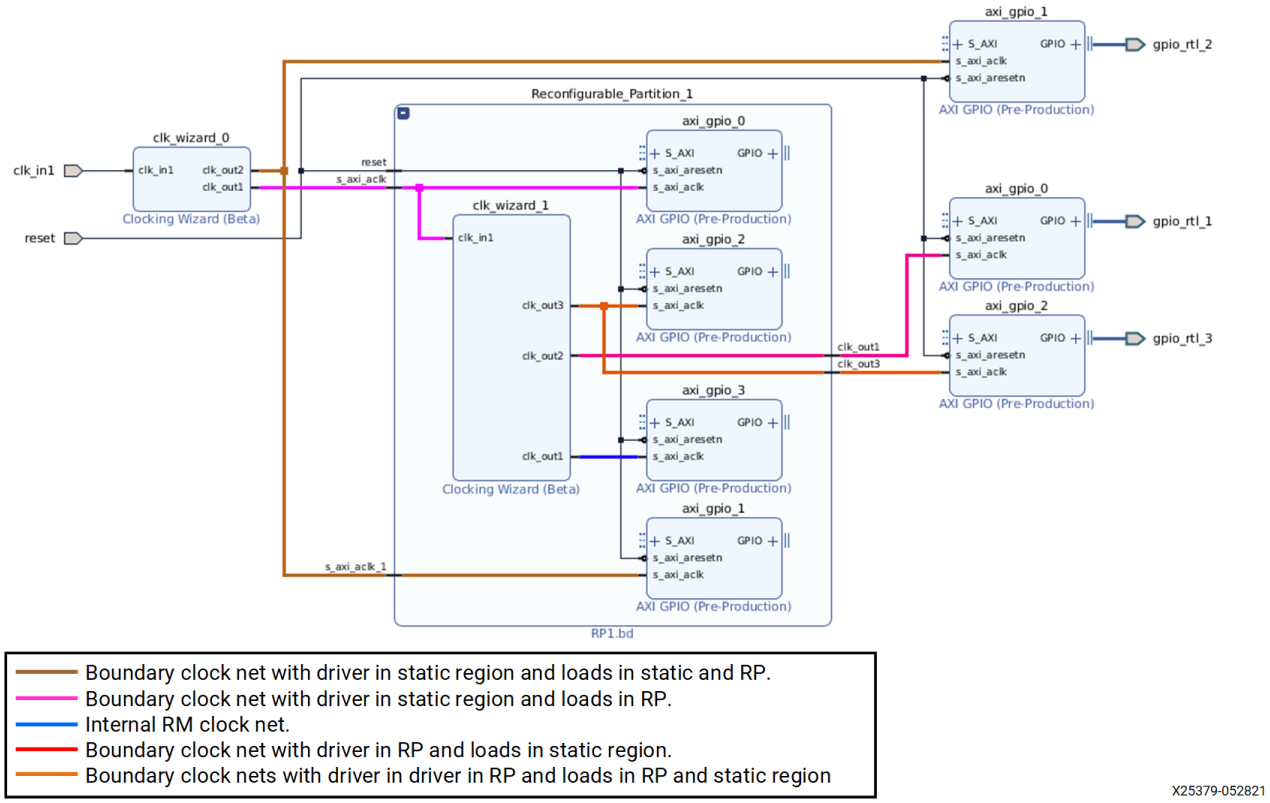
Most of the clocking guidelines of UltraScale+ stay the same for Versal devices. However, the Versal device clocking structure provides more clocking primitives and guidelines to better manage skew and meet timing closure more easily. For more information on the clocking structure in Versal devices, see the *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide* (UG1387).

This section lists possible clocking topologies supported in the DFX flow. In general, DFX designs clocks can be categorized into two types: boundary clocks and internal clocks.

Table 14: **DFX Designs Clocks Categorization**

Clock type	Description
Internal clocks	Clocks with driver and loads inside the reconfigurable partition
Boundary clocks	Clocks with nets crossing the reconfigurable module cell boundary • Driver in the static region and loads in the RM. • Driver in the RM and loads in the static region. • Driver in the static region and loads distributed between RM and static region. • Driver in the RM region and loads distributed between RM and static region.

Figure 140: **DFX Design Clocks**



Following is the DFX behavior for different categories of clock nets:

- Internal RM Clock Net
  - Clock root is placed at the center of loads inside RP Pblock.
  - More flexibility for placement and routability of the internal clock of RM in subsequent implementation.
  - Global clock partitioning is only required to allocate enough clock regions for a given net to drive RM loads. This can reduce the chance of clock partitioning errors.
  - This is recommended whenever possible to achieve better skew and optimal clock root placement.
- Boundary Clock Net
  - Boundary clock net track gets locked down after first implementation.
  - The PPLOCs of the boundary clock nets are distributed to all clock regions covered by the RP Pblock.
  - Global clock partitioning requires prerouting boundary clocks to every clock region within the RP even if the loads can fit into a single clock region.

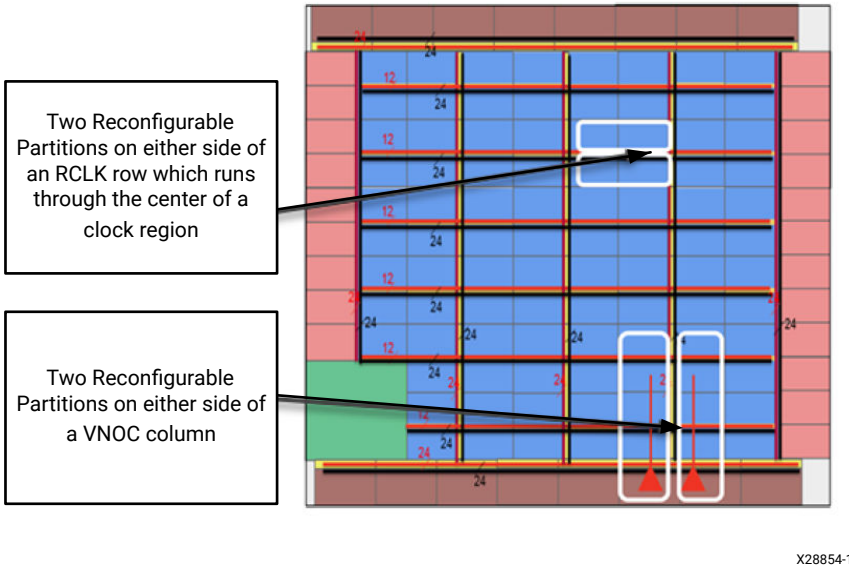
- The clock root of the boundary clock net can be placed anywhere in the device, because it can drive both static and RP loads. If the loads of boundary clock are located more in static region, it is possible that the clock root is placed in static region.
- If the first implementation is done using training logic in the RP Pblock, it is possible that boundary clock nets are locked down after first implementation with sub-optimal clock root location. AMD recommends using the `USER_CLOCK_ROOT` constraint on the boundary clock net to manually constrain the `CLOCK_ROOT` location.

Major Enhancements for Clocking in Versal Device DFX Designs Compared to UltraScale+ Devices

This section covers notable changes in Versal clocking. Versal allows clock tile splitting among multiple RPs. However, this is taken care by the tool. You only need to range the clock sources like MMCM and BUFGs to the Pblock, and DFX flow automatically pulls in the required clocking tiles for routing.

The following figure *Clock Tile Partition* shows `RCLK` row and `VNOC` column sharing among multiple reconfigurable partitions. The `RCLK` row sharing among two reconfigurable partitions allows sharing the clock region between two RPs (one above the `RCLK` and one below the `RCLK`). The `VNOC` tiles are also shared between multiple reconfigurable partitions. Versal devices support clock tile sharing for two RPs. However, if more than two RPs compete to find clock routing solution in a single VNOC column, it might cause unroutable situation. If there are more than two RPs, try to keep a clock region gap between them so that internal clocks of all the RPs do not compete for solution in same `VNOC` tile.

Figure 141: **Clock Tile Partition**



Controlling Clock Distribution within Dynamic Regions

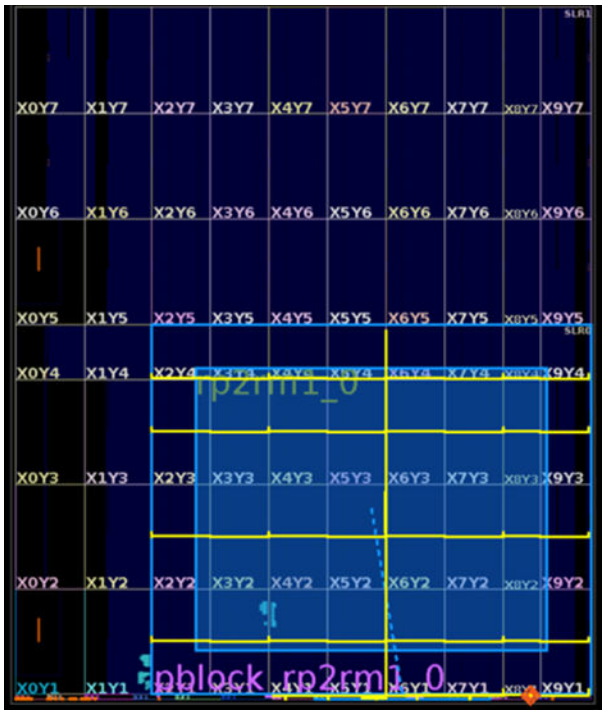
To create a generic solution for the full range of permissible RM loads, boundary clock net tracks are locked after the parent implementation, and the PPLOCs of boundary clock nets are distributed to all clock regions of an RP Pblock. This acts as a physical partition between RM and static domains. However, this approach can impact RM internal clocks, especially if there is a shift with fewer boundary clock loads and increased RM internal clock demands for RMs in child configurations.

To address this challenge, you can define a bounding box for boundary clock nets using the `USER_CLOCK_EXPANSION_WINDOW` property.

You can define a subset of clock regions within the RP Pblock with `USER_CLOCK_EXPANSION_WINDOW`, enabling boundary clock routing to cover only those specified in the RP. The clock tracks outside of the `USER_CLOCK_EXPANSION_WINDOW` region become available for other clocks. In the DFX flow, the `USER_CLOCK_EXPANSION_WINDOW` property must be applied on boundary clock nets in the parent implementation as routing is locked in the static design checkpoint.

The following figure shows a single RP design with the RP Pblock highlighted in blue and the BUFGCE source of a boundary clock net marked in red. The boundary clock net track highlighted in yellow is distributed to all clock regions within the RP, which is locked after the parent implementation.

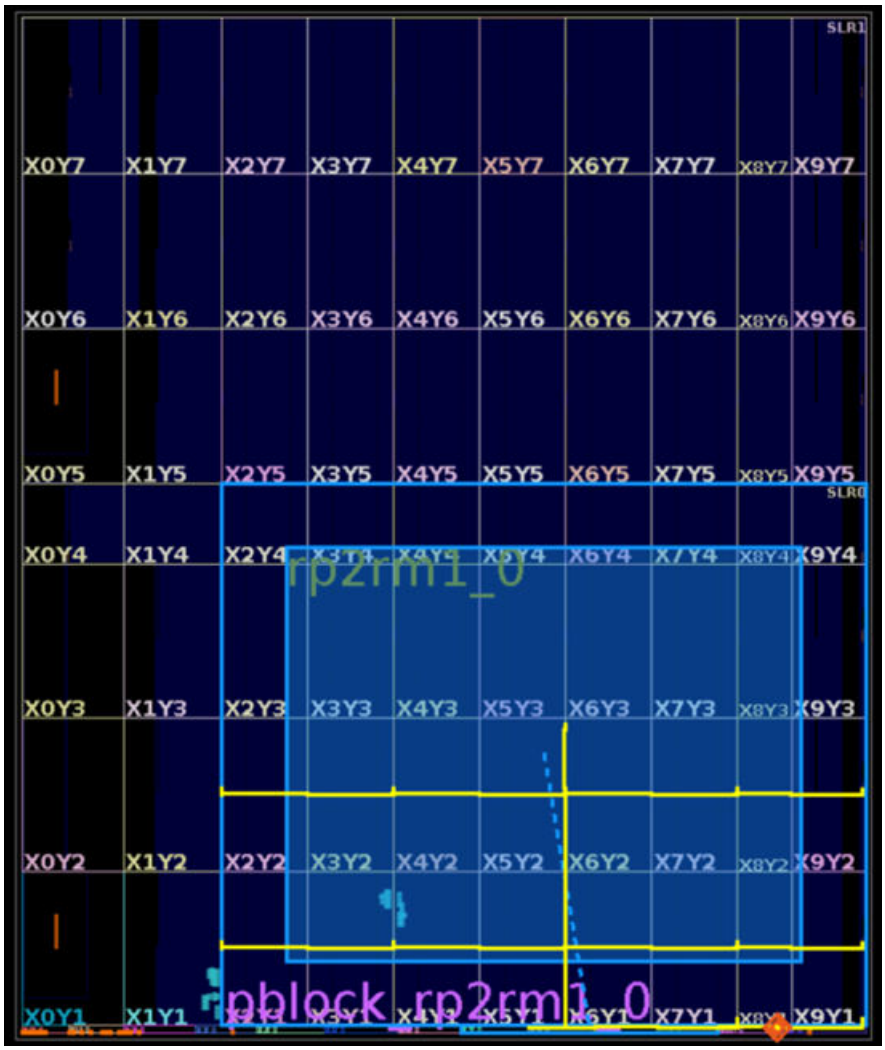
Figure 142: **Boundary Clock Net with Driver in a Static Region and Loads in Static and RP**



When you apply `USER_CLOCK_EXPANSION_WINDOW` to this boundary clock net to be contained within the range defined by the lower half of the RP Pblock, the clock track assignment is greatly reduced. Logic using this clock is limited to placement within this more restricted range.

```
set_property USER_CLOCK_EXPANSION_WINDOW
CLOCKREGION_X1Y1:CLOCKREGION_X9Y2 [get_nets clk1]
```

Figure 143: **USER_CLOCK_EXPANSION_WINDOW Applied on a Boundary Clock Net**



Known Restrictions for Clocking in Versal Device DFX Designs

Use of Divided Outputs in MBUFGCE Primitives for Boundary Clock Nets

MBUFG primitives in Versal devices allow leaf-level clock division to reduce clock track utilization and improve timing closure on synchronous clock domain crossings (CDCs). In earlier releases, Vivado restricted divided outputs of MBUFGCE primitives to static clock nets and internal Reconfigurable Module (RM) clock nets. You could not use these outputs on boundary clock nets in Dynamic Function eXchange (DFX) designs. The undivided O1 clock output from a static MBUFG can drive loads in one or more dynamic regions.

Vivado now supports divided outputs (O2, O3, O4) of MBUFGCE primitives as boundary clock nets, subject to the requirements in this section.

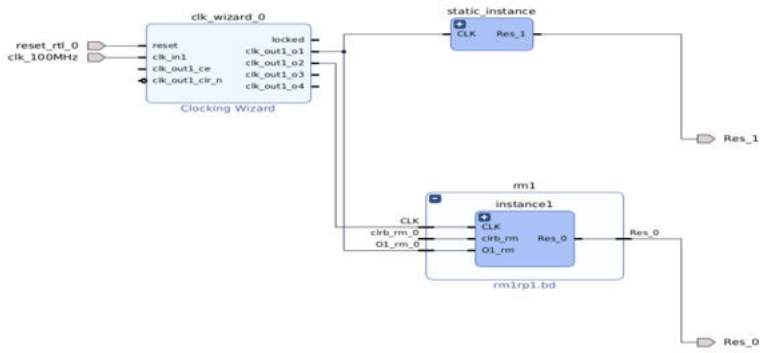
Supported Usage

The following configuration is supported:

- The MBUFGCE instance is in the Static region.
- One or more divided outputs (O2, O3, O4) drive loads within a single RM.

Required Design Methodology

- O1 Boundary Clock Connection
  - You must connect the O1 output of the MBUFGCE to the RM through a hierarchical boundary pin.
  - The O1 output defines the routed boundary clock net. Vivado implements divided clocks (O2/O3/O4) using BUFDIV_LEAF resources near the loads. Vivado derives PPLOC assignment and clock region expansion from the O1 boundary clock net.

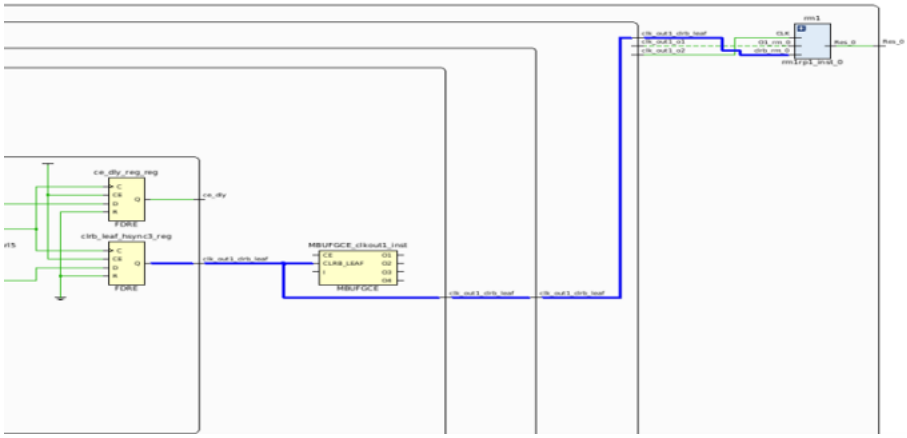


  - Using a divided output as a boundary clock without establishing the O1 boundary connection results in the following design rule check (DRC) error.

```
[DRC HDPR-154] Versal RM Interface Pin Check for MBUFG divided clocks:
Reconfigurable module <RM name> has a MBUFGCE <instance> divided clock
used as a boundary clock net, but the corresponding master clock (O1)
is not connected to the RM boundary.
```

- CLRB_LEAF Signal Connection
  - You must connect the CLRB_LEAF input to the RM through a hierarchical boundary pin.
  - The CLRB_LEAF input on MBUFG primitives asynchronously resets the BUFDIV_LEAF divider elements. In DFX designs, you must ensure that BUFDIV_LEAF elements are reset to a known state across Static and RM regions. If the upstream clock driving the MBUFG is reset during operation, the MBUFG output clocks must also be reset to maintain synchronization.

- Although it is logically a static signal, CLRB_LEAF drives divider elements located inside the RM. You must therefore model it as a boundary net with explicit connectivity.



- Using a divided output as a boundary clock without establishing the CLRB_LEAF boundary connection results in the following DRC error.

```
[DRC HDPR-154] Versal RM Interface Pin Check for MBUFG divided clocks:
Reconfigurable module <RM name> has a MBUFGCE <instance> divided clock
as a boundary clock net, but its CLRB_LEAF driver net is not connected
to the RM.
```

Unsupported Boundary Clock Topologies

The following configurations using MBUFGCE divided outputs as boundary clock nets are not supported:

- MBUFGCE located in RM driving boundary loads: An MBUFGCE instance within an RM must not drive boundary clock nets or static region loads. This configuration results in a DRC error.

```
[DRC HDPR-158] Versal Illegal MBUFGxx drivers in pblock
Reconfigurable module <RM name> has a MBUFGCE divided clock as a boundary
clock net with MBUFGCE in RM. This is not currently supported. MBUFG
divided clocks can be used as boundary clocks only when MBUFG is in
static.
```

- Use across multiple RMs: Divided outputs from a single MBUFGCE must not drive loads in multiple RMs. This configuration results in a DRC error.

```
[DRC HDPR-162] Unsupported MBUFGCE connectivity of divided clock: Static
MBUFGCE <instance> with active CLRB_LEAF pin drives a divided clock with
loads in multiple Reconfigurable Modules. This will cause unintended
clock resets. Please ensure the divided clock output from a static
MBUFGCE drives loads within a single Reconfigurable Module (RM) only.
```

- Use across static and RM regions: Divided outputs from a single MBUFGCE must not drive loads across Static and RM regions. This configuration results in a DRC error.

```
[DRC HDPR-162] Unsupported MBUFGCE connectivity of divided clock: Static
MBUFGCE <instance> with active CLRB_LEAF pin drives a divided clock with
loads in static and Reconfigurable Module. This will cause unintended
clock resets. Please ensure the divided clock output from a static
MBUFGCE drives loads within a single Reconfigurable Module (RM) only.
```

Restrictions in Clock Resource Usage Due to Clock Tile Splitting

When a row of clock tiles is shared between multiple RPs, it is possible that some of the sites along this row cannot be used for placement. To avoid potential unroutability because of tile splitting, the DFX flow automatically prohibits usage of certain clocking or logical resource tiles. To avoid this scenario, AMD recommends keeping a gap of at least one clock region between multiple RP Pblocks if utilization estimation meets the design need.

For example, in a multi-RP design, (with RPs RP1 and RP2), if a clock for RP2 is required to traverse through RP1 to reach loads in RP2, some block RAM sites in the traversed RP (RP1) are prohibited for use by the placer.

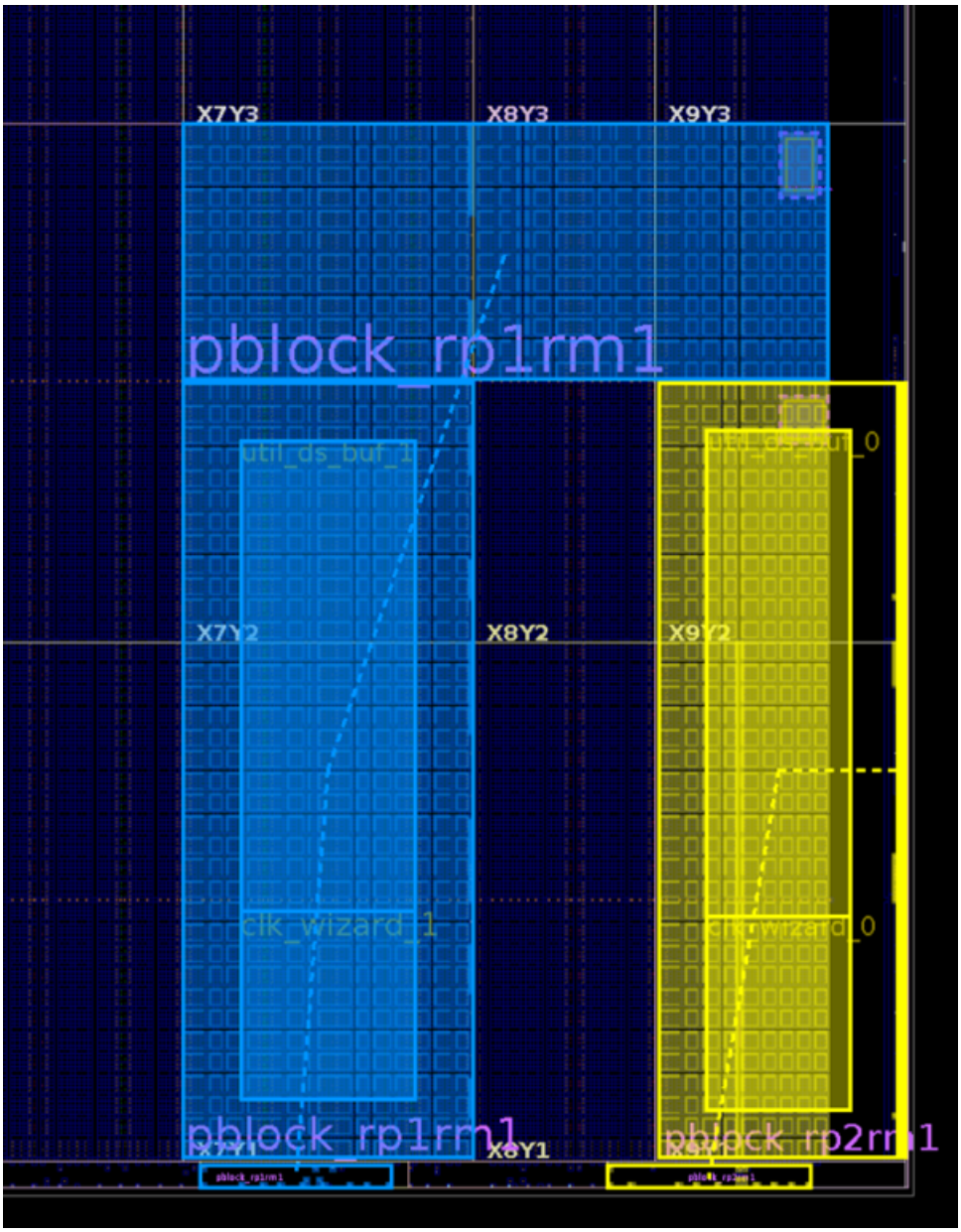
In this scenario some of the clock routing resources in RP1 are also claimed by RP2 so they are shared equally. The RCLK tiles RCLK_BRAM_CLKBUF_* are part of clock routing network and due to sharing by the 2 RPs, only the top or bottom half can be claimed by RP1. Due to the configuration frame programming during reconfiguration, RCLK_BRAM_CLKBUF tiles must be programmed together with all block RAM tiles in the same half column. A critical warning is issued during `opt_design` for such a scenario. The prohibited sites can be viewed in the Device View.

```
[Constraints 18-5689] RCLK tile RCLK_BRAM_CLKBUF_CORE_X*Y* is shared by
PBLOCK RP1 (owns LSB tracks) and PBLOCK RP2 (owns MSB tracks). For the
shared usage, BRAM tiles and their adjacent interface tiles at the NORTH
of the shared RCLK tile are prohibited because they could not be used for
placement within PBLOCK RP1.
```

Clocking Instances Can Be Prohibited Due to Expanded Routing Footprints

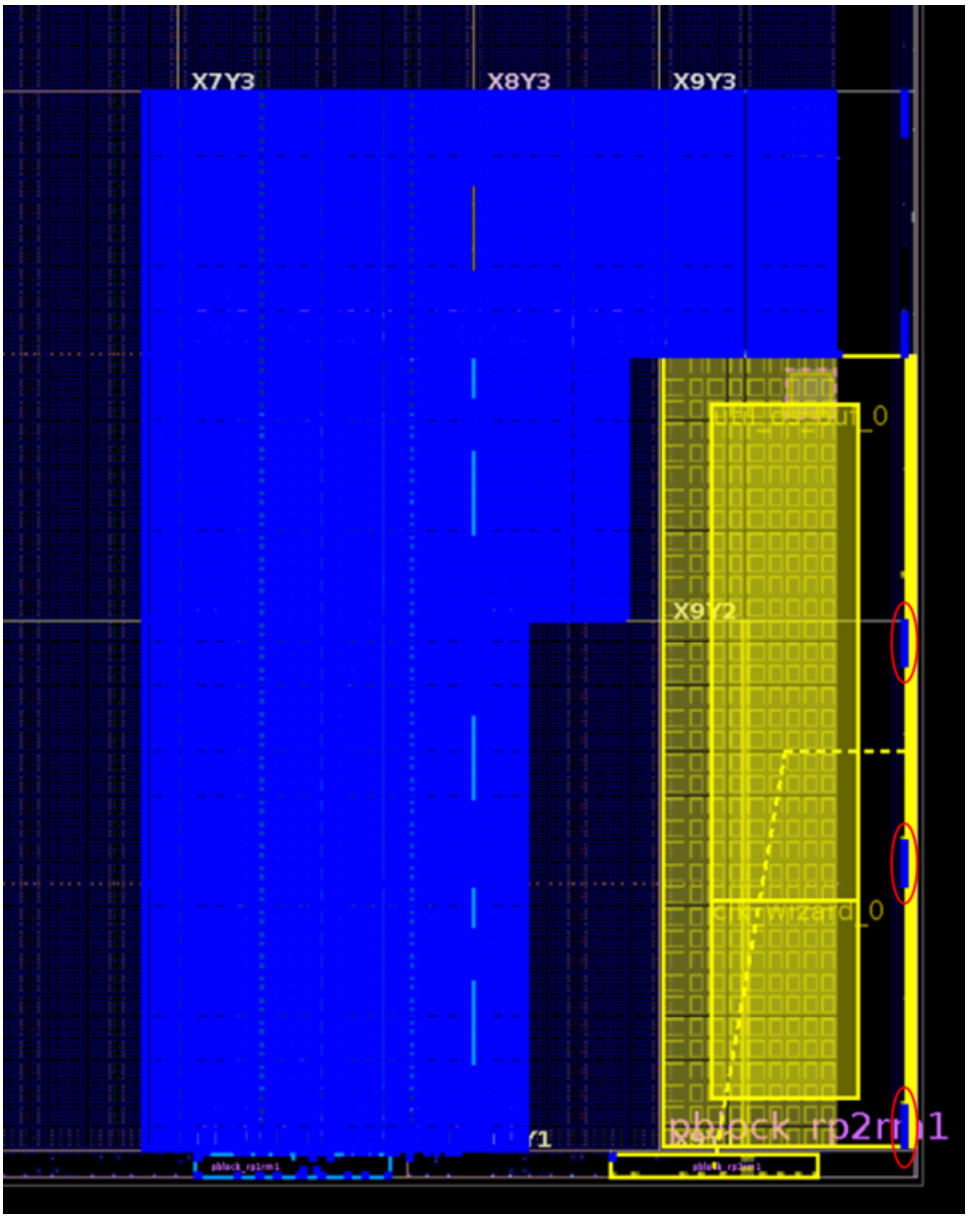
For DFX designs with two or more Reconfigurable Partitions, clock buffers can be blocked from use if two RP footprints expand to both cover the same resources. In the following floorplan, both RP (rp1rm1 shown in blue and rp2rm1 shown in yellow) occupy space in the X9 column (the farthest clock region on the right). Clock region X9Y2 is legally shared between the two partitions.

Figure 144: **Clocking Example with Two RPs**



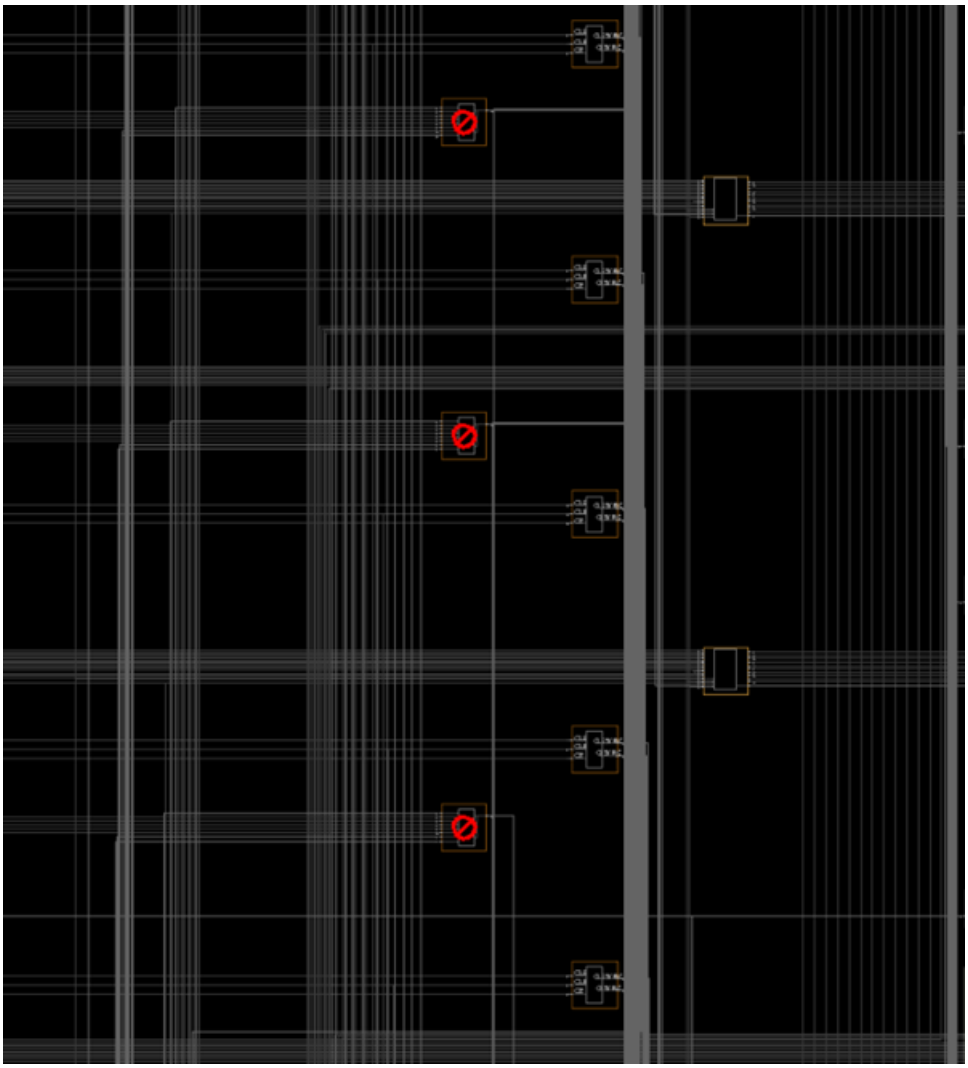
When routing expansion obtains BUFG_GT resources, both try to collect the sites along the right side of the chip. The following image shows the expanded routing footprint of pblock_rp1rm1, which includes sites within the pblock_rp2rm1 area.

Figure 145: **Clocking Example with Expanded Routing Footprint**



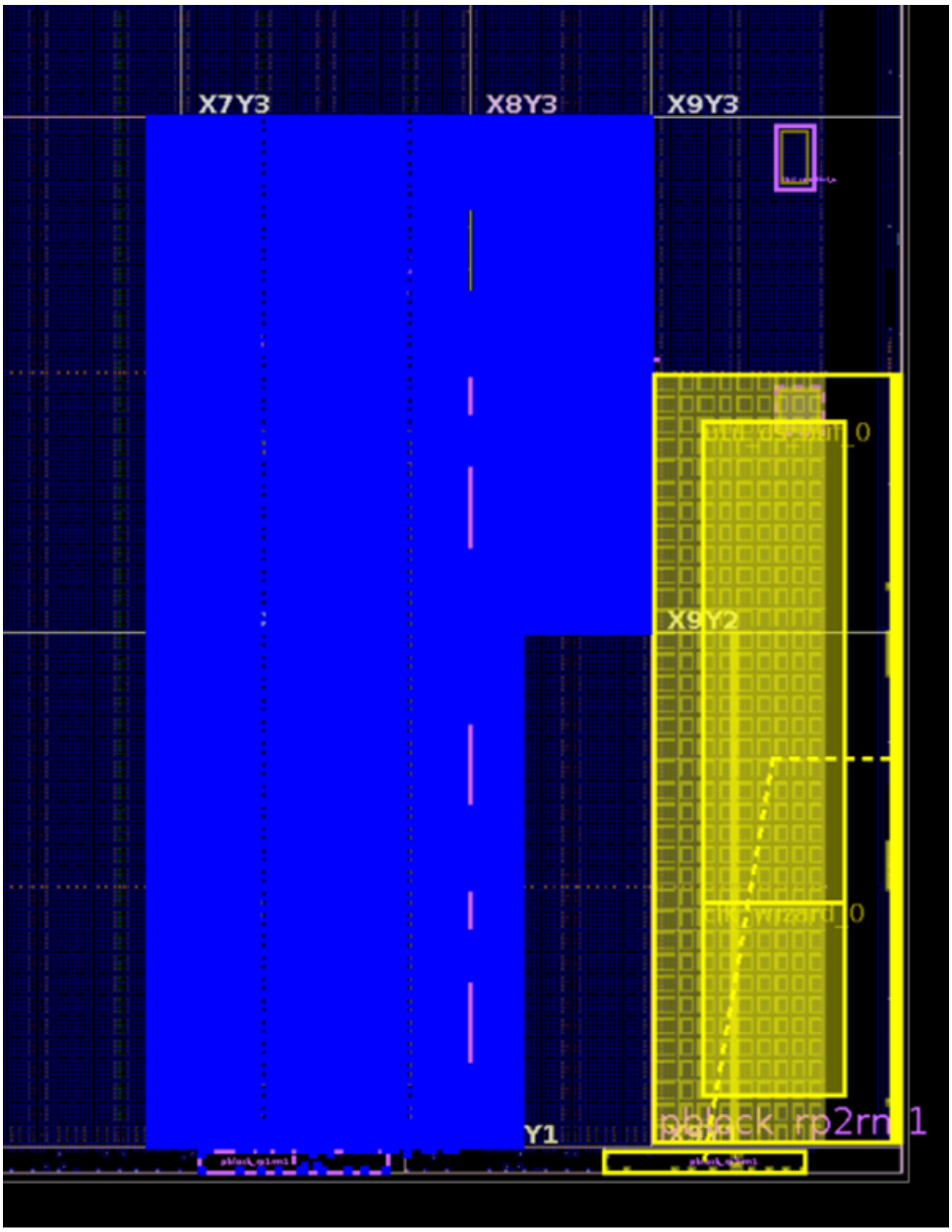
Two independent RPs cannot both own the same resources. Any sites that are in conflict are marked with a PROHIBIT property so neither RP is able to use them. DFX and Floorplanning DRCs do not currently flag this condition. The only indication that this conflict exists is the existence of prohibited locations after `opt_design`. In general this does not lead to an error, unless the reduced set of clocking resources are insufficient to implement the design.

Figure 146: **Prohibited Sites**



To resolve this resource overlap, adjust the blue Pblock to avoid the clock regions in the X9 column. The yellow Pblock then has sole ownership of the BUFG_GTs on the right side of the chip. The shared X9Y2 clock region is irrelevant. Even if the blue Pblock does not include this clock region but does occupy clock regions above (for example, X9Y3) the same resource usage would be requested.

Figure 147: **Adjusted Clocking Example with Two RPs**



Network on Chip

The Network on Chip (NoC) is an important new silicon feature within Versal devices, enabling fast communication throughout each device. NoC elements can be assigned to the static or dynamic parts of the design, just like other fundamental resources such as CLB or BRAM. This section contains a summary of different ways to use the NoC in a DFX design, and the rules and considerations that go along with them.

The NoC is composed of NMUs, NSUs, NPSs, and NIDBs. The NoC master unit (NMU) is the traffic ingress point. The NoC slave unit (NSU) is the traffic egress point. Both hard and soft IPs have some number of these master and slave connections. The NoC Inter-Die-Bridge (NIDB) connects two super logic regions (SLRs) together, providing high bandwidth between dies. The NoC Packet Switch (NPS) is the crossbar switch, used to fully form the network. The Inter-NoC Interface (INI) provides a means of connecting two NoC instances (either axi_noc or axis_noc). An INI link represents a logical connection within the physical NoC that is resolved at the time the NoC compiler is called.

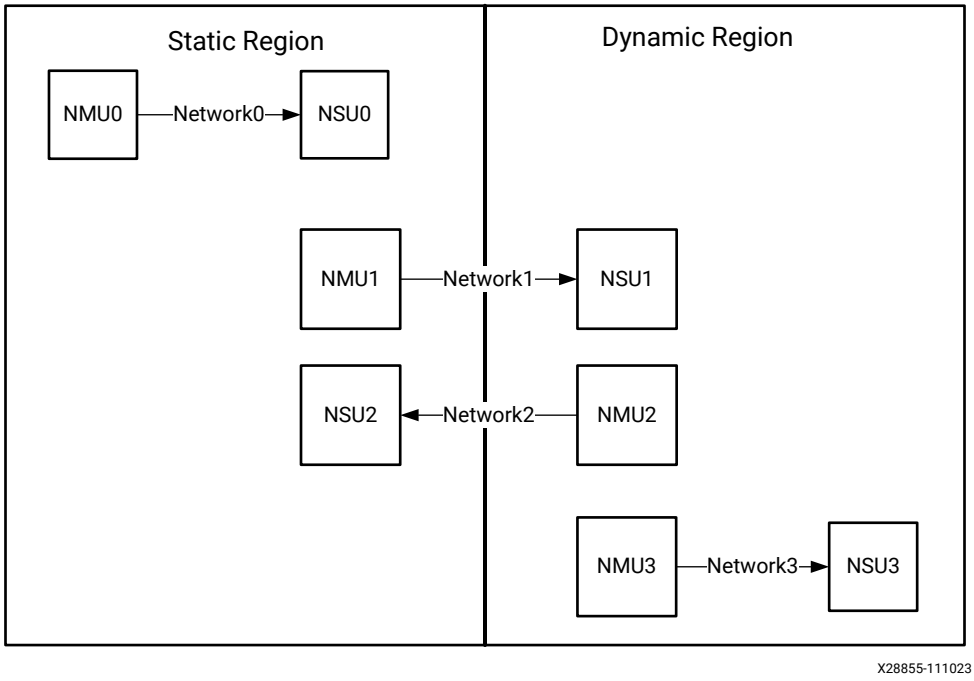
For detailed information on the NoC in Versal devices, refer *Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313).

RECOMMENDED: *Avoid using pass-through addressing connectivity instances within Reconfigurable Modules (RMs). If you use this approach, see Answer Record 34127.*

NoC Topologies Supported in the DFX Design Flow

In a DFX design, the network on chip (NoC) networks can be distributed across both static and dynamic regions of the design, or they can be contained within a single region. The following figure shows four simplified NoC networks to illustrate this concept.

Figure 148: **NoC Topologies Supported in DFX Design Flow**



- **Network-0:** This NoC topology includes both the source (NMU) and target (NSU) components located within the static region. During the parent implementation phase of the DFX flow, the Quality of Service (QoS) parameters for this network are established. The NoC compiler solution to meet the QoS requirements includes factors like read bandwidth, write bandwidth, latency, and location, and this solution is locked down within the static region. This ensures that subsequent child RM implementations adhere to these established QoS parameters.
- **Network-1:** In this NoC topology, the source component (NMU) is located in the static region, and the target component (NSU) is in the dynamic region. To facilitate communication across the DFX partition, peripherals within the static region can access the NoC network through the ingress Endpoint NMU and exit through the egress Endpoint NSU to communicate with peripherals in the dynamic region.
- **Network-2:** This NoC topology is similar to the Network-1 topology except that the source component (NMU) is located in the dynamic region, and the target component (NSU) is located in the static region. One common use case for this topology is the programmable logic (PL) peripherals in the dynamic region accessing the DDR memory resources located in the static region. In Versal devices, DDR memory controllers are NoC target Endpoints (NSUs).
- **Network-3:** With regard to DFX design flexibility, this NoC topology is the most versatile. Both Endpoints of the network, the source component (NMU) and the target component (NSU), are positioned within the dynamic region. As a result, this topology can undergo a complete reconfiguration to align with the specific requirements of each reconfigurable module.

Design Entry Methodology for DFX Designs that use NoC

The modular NoC technology offers RTL-centric customers greater flexibility to integrate NoC into their designs. This is particularly advantageous for DFX users who use RTL-centric scripted flows to develop their DFX designs. This section describes two methodologies:

- RTL Modular NoC Flow for DFX Designs
- IP Integrator Block Design Container Flow

RTL Modular NoC Flow

In modular NoC flow, you instantiate Xilinx Parameterized Macros (XPMs) of PL-facing NMU and NSU in your RTL. The XPM NMU and NSU can be instantiated in the static or dynamic regions of the design. Xilinx design constraints (XDCs) are used to make connections between NMU and NSU. To learn more about the modular NoC flow, see the *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide* (UG1387).

As stated in NoC Topologies Supported in the DFX Design Flow, DFX flow supports all four NoC topologies described above. It is straightforward when NoC endpoints (NMU or NSU) are entirely within either the static region or dynamic region. However, if a NoC network spans DFX partitions (for example, NMU in static and NSU in dynamic), additional constraints are needed.

Related Information

NoC Path Ownership in the DFX Design Flow

Virtual NoC Interface

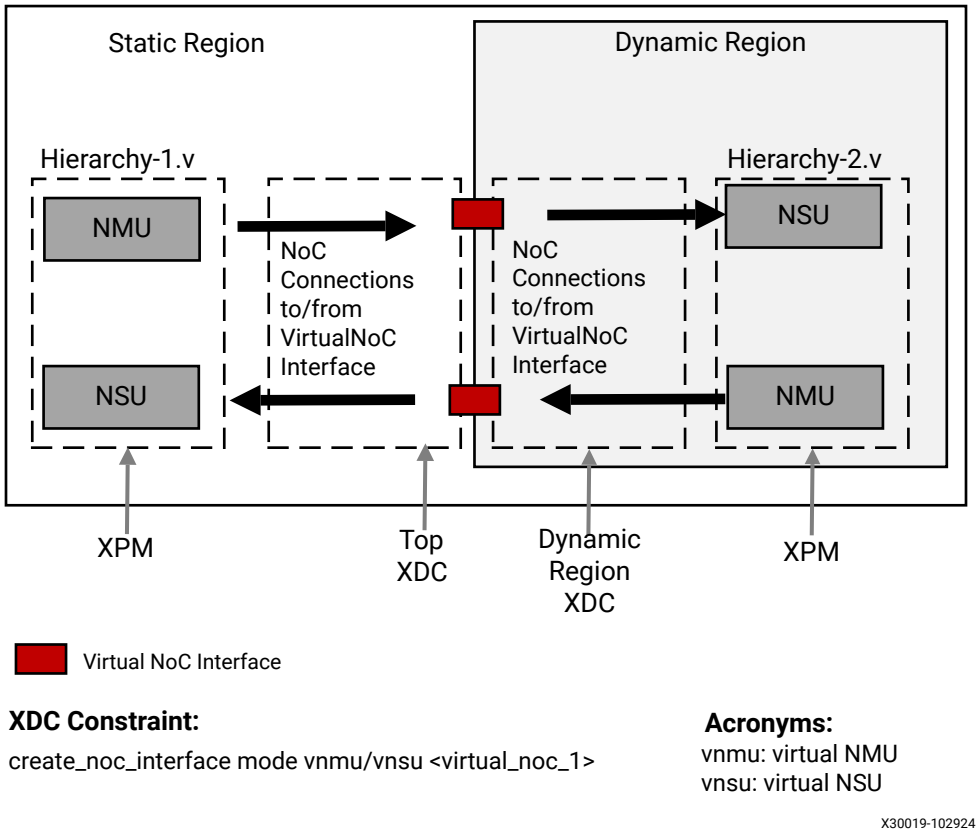
A virtual NoC interface is a software construct created for DFX designs to enable the following:

- Legalize the boundary NoC path in a self-contained static or dynamic region synthesis.
- Define the aperture of the boundary NoC path that the peripherals connected to it can access.

A virtual NoC interface is created as an XDC constraint in the modular NoC flow as follows:

```
create_noc_interface -mode [vnmu|vnsu] <virtual_noc_1>
```

Figure 149: **Boundary NoC Paths Across the DFX Partition Traversing Through Virtual NoC Interfaces**



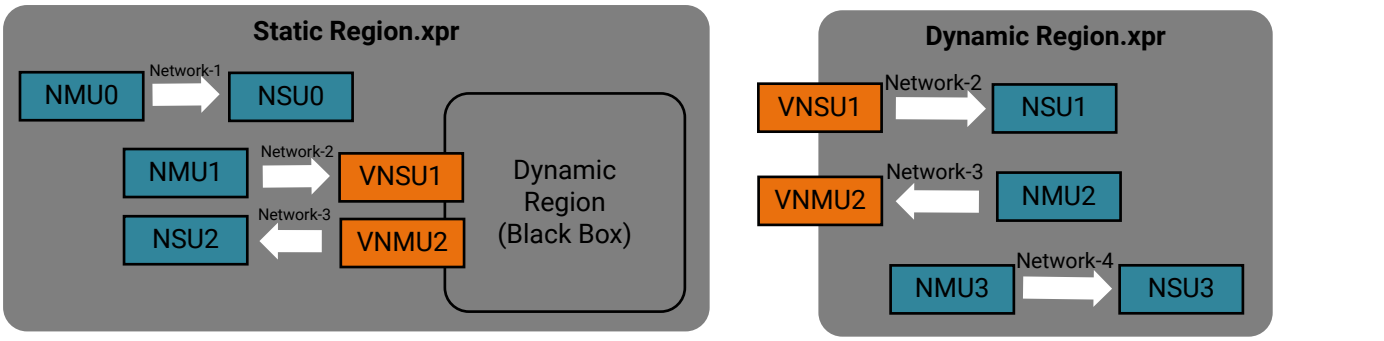
Scope the DFX Region NoC Constraints

Scoping of constraints to the dynamic region is strongly recommended in the DFX flow. This is primarily due to the design flow requirement that dynamic regions are synthesized in out-of-context mode, separate from the top-level static design. A virtual NoC interface concept enables scoping the NoC constraints to the dynamic region. A virtual NoC interface acts as an anchor point to model the ingress or egress point of boundary NoC networks. This also enables basic validation of the solution within the context of dynamic region synthesis.

***Note*:** The NoC is a system-wide resource, and a complete validation of the NoC solution is only possible after all individual modules are assembled (after the `link_design` phase).

The following figure shows the methodology of using the virtual NoC interfaces in two separate projects: one for a static region synthesis and the other for a dynamic region synthesis. The same name is used in both projects. The `link_design` command or `read_checkpoint -cell` command matches the name of the virtual NoC interfaces from the static and dynamic region DCPs to resolve this, completing the NoC path from NMU to NSU.

Figure 150: **Virtual NoC Interfaces Defined at Static and Dynamic Region Projects**



top.xdc
create_noc_interface vnsu1
create_noc_connection -source nmu1 target vnsu1

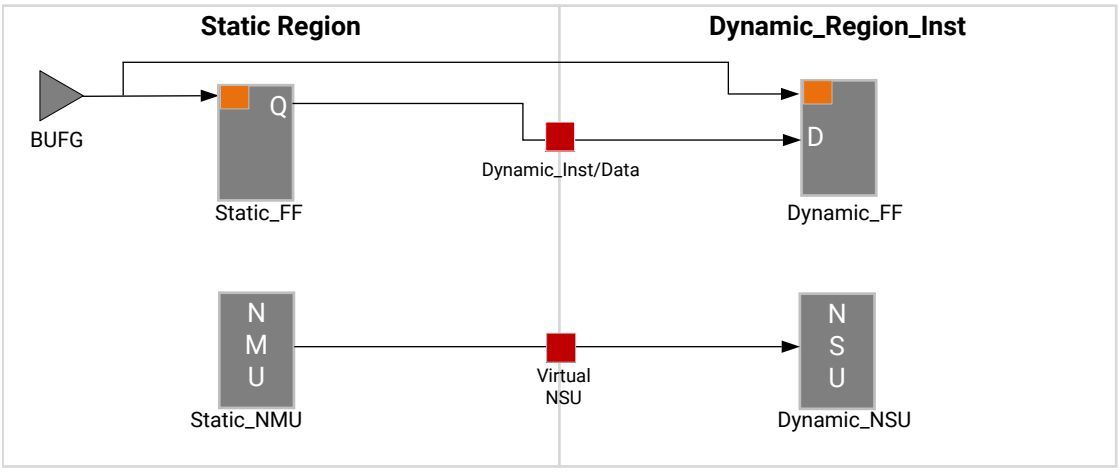
rm.xdc
create_noc_interface vnsu1
create_noc_connection -source vnsu1 target nsu1

X30020-102924

Drawing an Analogy to Timing Constraints

The virtual NoC interface functions similarly to the hierarchical pins at the boundary of the DFX module, but it is modeled within the XDC domain. The timing constraints methodology suggested for paths crossing DFX modules also applies to NoC constraints. For instance, consider the scenario depicted below.

Figure 151: **Treat Virtual NoC Interface as a Hierarchical Pin of the NoC Connection at the DFX Boundary**



X30023-102924

If you need timing constraints applied to paths across DFX module boundaries (such as the false path constraint on all timing paths from a specific static leaf cell to an RM leaf cell of all RM variants), defining the constraint referencing an object inside a specific RM variant might lead to invalid constraints if the next RM variant has a different netlist. For example, a false path constraint defined as:

```
set_false_path -from [get_pins static_FF/Q] -to [get_pins Dynamic_FF/D]
```

is referring to an object inside the dynamic region (D pin of a register inside the reconfigurable module). This constraint can become invalid as the next RM variant might not have this netlist object. Therefore, the recommendation is to define the timing constraint referencing the hierarchical pin at the DFX module boundary. The same false path constraint can be rewritten to point to refer all paths traversing through the hierarchical pin:

```
set_false_path -from [get_pins static_FF/Q] -through [get_pins Dynamic_Inst/
Data]
```

This approach allows the constraint to be maintained in the abstract or full shell, ensuring that any timing path through that hierarchical pin always receives the intended constraint.

Similarly, NoC constraints across DFX boundaries also follow a comparable methodology, where any NoC connection between static and DFX partitions must pass through the virtual NoC interface. For example, instead of creating a NoC connection direction from NMU in static to an NSU in dynamic region:

```
create_noc_connection -source static_NMU -target dynamic_NSU
```

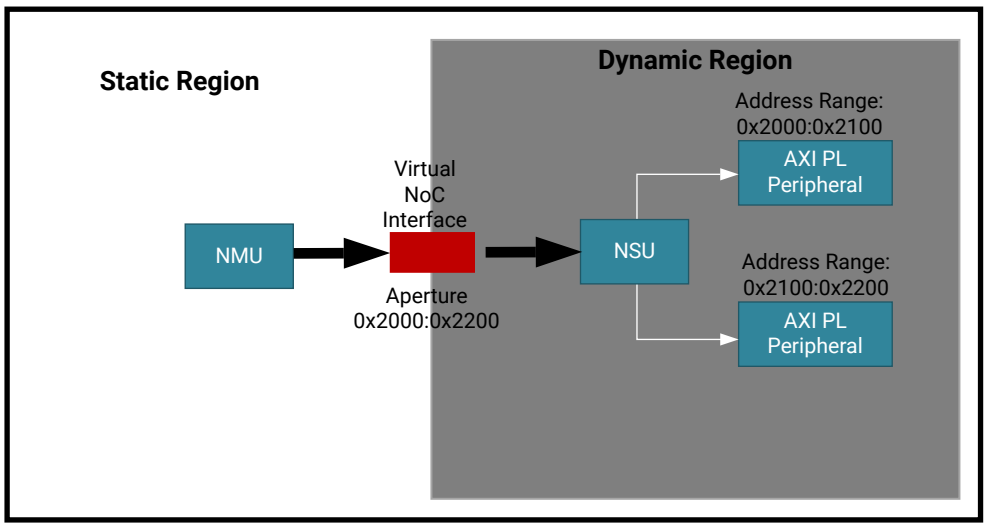
AMD recommends creating the NoC connection to/from the virtual NoC interface created at the DFX boundary:

```
create_noc_connection -source static_NMU -target virtual_NSU
```

Defining Aperture for DFX Region Peripherals

Virtual NoC interfaces are also used to define the address aperture to encapsulate the address segments of peripherals connected through that NoC path. DFX designs can have multiple reconfigurable modules with different address range requirements depending on the IPs used in each RM. A virtual NoC interface is used to define the aperture covering the comprehensive requirements of all RMs.

Figure 152: **Aperture of a Boundary NoC Path at the DFX Boundary Defined at the Virtual NoC Interface in XDC**



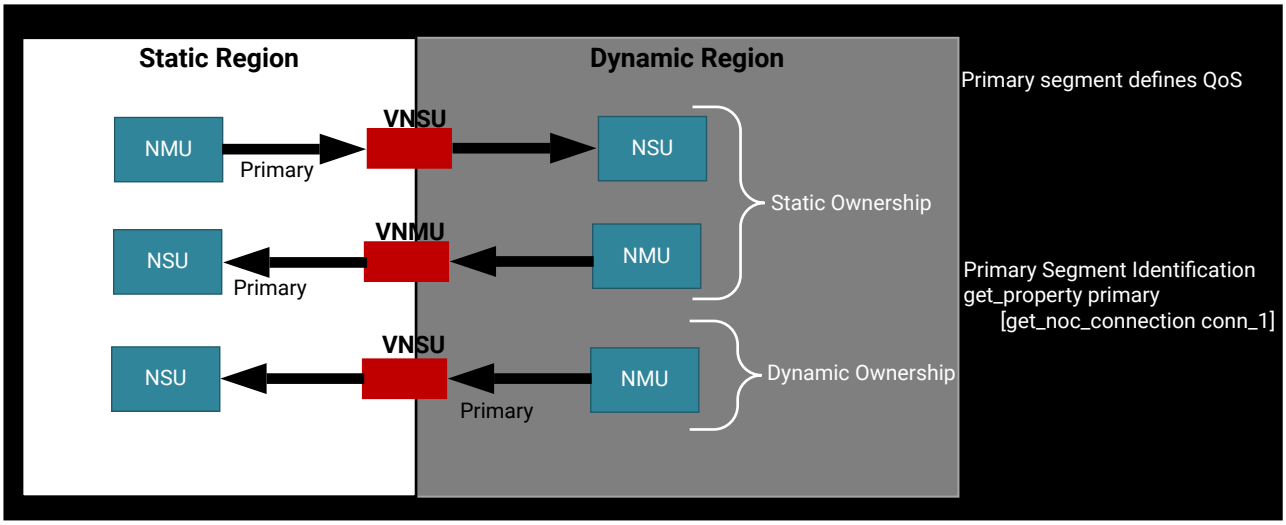
Ownership Concept

When a NoC path crosses a DFX boundary, the specific path can be owned by the static region or the dynamic region. Static ownership of a NoC path ensures that the NoC solution for that boundary NoC path is locked down after initial parent implementation and the same NoC solution is honored and reused for subsequent child implementations. The following figure demonstrates the different ownership possibilities for DFX boundary NoC paths.

- When an NMU is in the static region drives an NSU in a dynamic region, the only supported ownership is static ownership.
- When an NMU is in a dynamic region drives an NSU in the static region, you can choose the path to be either static-owned or dynamic-owned. AMD recommends static ownership if there are more than one reconfigurable partitions in the design.
- Due to the presence of two connections depicting the specific boundary NoC path – one between the virtual NoC interface and the NoC endpoint in the static region, and the other between the virtual NoC interface and the NoC endpoint in a dynamic region – the primary property has been introduced for the NoC connection to indicate ownership.

```
get_property primary [get_noc_connections <connection_name>]
```

Figure 153: **Boundary NoC Path Ownership Options**



Static Region Ownership and Virtual NoC Interfaces

In the case of static ownership:

- For each NMU or NSU in the dynamic region, define a corresponding virtual NMU or NSU in the XDC. This creates a one-to-one mapping between the virtual NoC interface and the real NoC interface within the dynamic region.
- In the static region, multiple NMUs can connect to a single Virtual NSU, and multiple NSUs can connect to a single virtual NMU. This creates an N-to-1 mapping between the real NoC interface in the static region and the virtual NoC interface.

Dynamic Region Ownership and Virtual NoC Interfaces

In the case of dynamic ownership:

- For each NSU in the static region, define a corresponding virtual NSU in the XDC. This creates a one-to-one mapping between the NSUs in the static region and the virtual NSUs defined in the XDC.
- Multiple NMUs in dynamic region can connect to single virtual NoC interface defined in XDC creating a N-to-1 mapping between the real NMUs in the dynamic region and the virtual NoC interface.

Modular NoC DFX Tutorials

Refer to the modular NoC-based DFX design tutorial posted on GitHub.

IP Integrator Block Design Container Flow

Inter-NoC Interconnect (INI)

For users familiar with IP integrator, the block design container (BDC) flow can be used for DFX designs. The reconfigurable module is a block design, and custom RTL submodules in the dynamic region must be integrated using RTL module reference or IP packaging. AXI NoC IP is the main design entry for IP integrator-based NoC designs. Use the inter-NoC interconnect (INI) for NoC connectivity between static and dynamic regions. INI, similar to the virtual NoC interface, connects two logical NoC IPs in different block designs or hierarchies but it is important to note that this approach cannot be used to traverse across RTL files. Instead, it connects the top block design (static) and the reconfigurable module block design directly. As a result, AMD recommends using INI only for the IP integrator BDC-based DFX flow and not for the RTL-based DFX flow. For RTL-based DFX designs that use NoC, you must use the RTL modular NoC flow. Also, AMD recommends the use of IP integrator BDC-based DFX flow only if you have relatively little custom RTL in the dynamic region. Otherwise, repeated changes in the custom RTL can make the design iteration slower with the IP integrator BDC flow.

INI Ownership Concept

The INI strategy plays a crucial role in determining ownership for boundary NoC paths, which are paths that traverse the DFX boundary. Following are the configurable options for the INI strategy:

- **Single Load:** Driver owns the path
- **Single Driver:** Load owns the path

For boundary NoC paths that include an NMU located in the static region, the INI strategy must be set to single load, driver owns the path. This strategy ensures that the static region retains ownership of the path. To enable 1:N connectivity, you might need to incorporate multiple INI interfaces to connect to the NMU, as described in the next section.

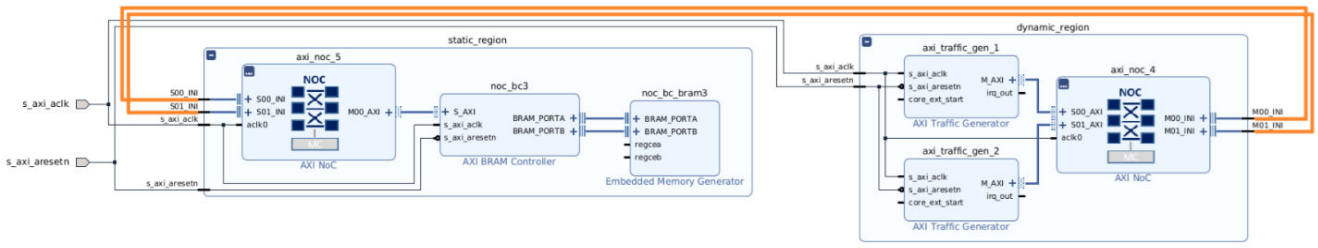
Similarly, for boundary NoC paths where the NMU resides in the dynamic region, you have the flexibility to determine ownership as either the static or dynamic region. Selecting static region ownership requires configuring the INI strategy as single driver, load owns the path. Conversely, selecting dynamic region ownership requires configuring the INI strategy as single load, driver owns the path, as described in the next section.

Static Region Ownership and INI

In the case of static region ownership:

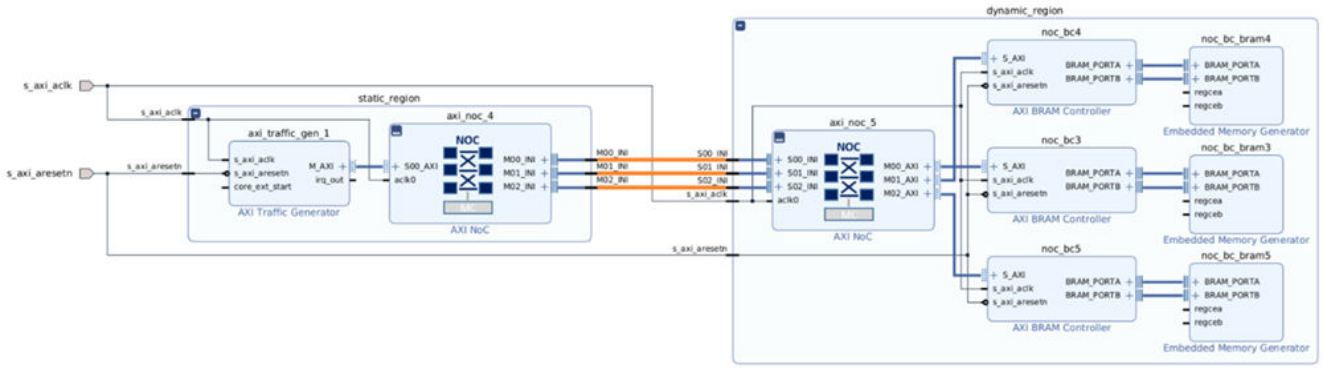
- For each NMU in the dynamic region, define a corresponding INI (strategy: Single Driver, Load Owns the path) in the dynamic region BD. This creates a one-to-one mapping between the INI and the real NMU interface within the dynamic region BD.
    - In the static region, multiple NSUs can connect to a single INI interface (strategy: same as dynamic region BD: Single Driver, Load owns the path). This creates an N-to-1 mapping between the real NoC interface in the static region BD and the INI.

Figure 154: **Two NMUs in the Dynamic Region (Two INIs Configured as a Single Driver)**



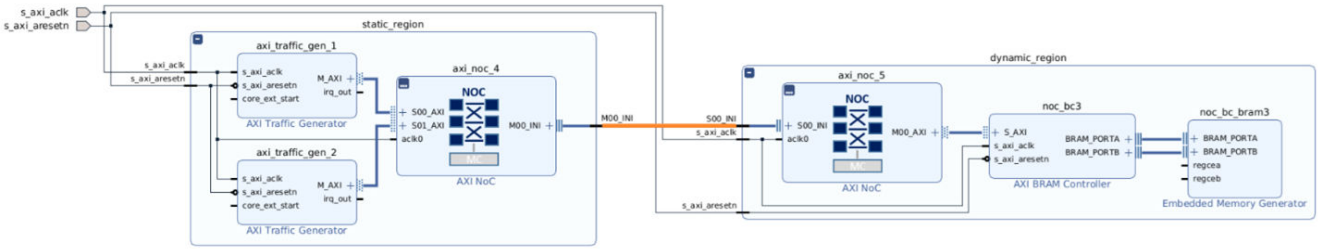
For each NSU in the dynamic region, define a corresponding INI (strategy: Single Load, Driver owns the path) in the dynamic region BD. This creates a one-to-one mapping between the INI and the real NSU interface within the dynamic region BD.

Figure 155: **Three NSUs in the Dynamic Region (Three INIs Configured as a Single Load)**



In the static region, multiple NMUs can connect to a single INI interface (strategy: Single Load, Driver owns the path). This creates an N-to-1 mapping between the real NoC interface in the static region BD and the INI.

Figure 156: **Two NMUs in the Static Region Connected to a Single INI Interface**

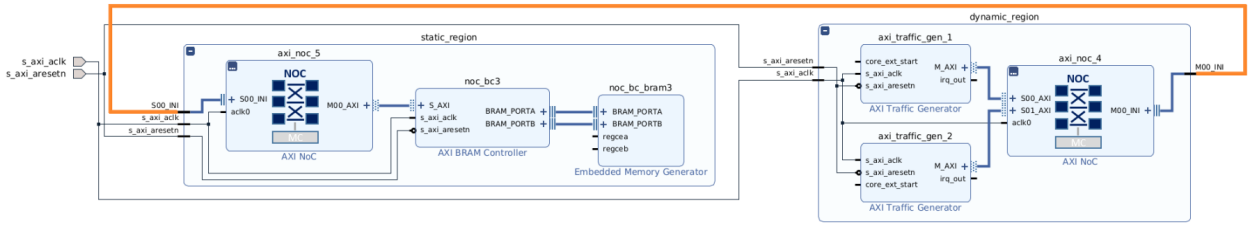


Dynamic Region Ownership and INI

In the case of dynamic region ownership:

- For each NSU in the static region, define a corresponding INI (strategy: Single Load, Driver owns the path) in the static region BD. This creates a one-to-one mapping between the NSUs in the static region and the INI.
- Multiple NMUs in dynamic region can connect to single INI defined in BD creating a N-to-1 mapping between the real NMUs in the dynamic region and the virtual NoC interface.

Figure 157: **Two NMU in a Dynamic Region Connecting to a Single INI as the INI is Configured to Single Load**



Considerations with IP Integrator BDC DFX Flow

- The IP integrator BDC flow is considered as a BD-top design flow where your design is primarily IP integrator-centric. This design flow necessitates that each reconfigurable module is a block design by itself and any custom RTL inside each reconfigurable module must be brought inside the block design by either RTL module reference or IP packaging.
- The IP integrator BDC flow is primarily developed for a single project environment. If a design team has a relatively higher number of reconfigurable modules per DFX platform or requires a distributed project environment, AMD recommends using a non-project scripted flow using RTL Modular NoC.
- INI can be used to connect between different AXI NoC logical IP instances, either in the same BD or in different BDs, (using the BDC flow). INI must not be used to traverse custom RTL logic. AMD recommends that advanced RTL flow users leverage the modular NoC-based DFX design flow.

IP Integrator Block Design Container DFX Tutorials

Refer to the DFX tutorials based on the IP integrator block design container technology for more information.

NoC Path Ownership in the DFX Design Flow

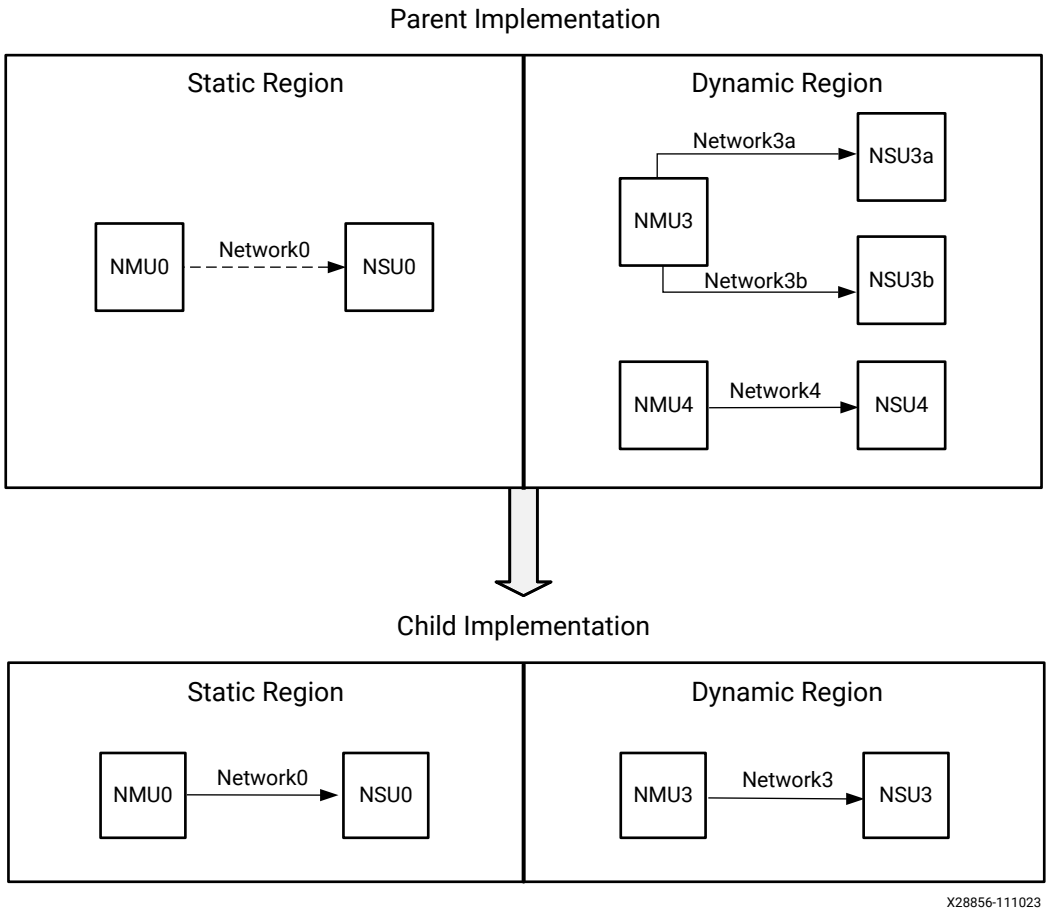
The DFX design flow supports various NoC topologies. It is important to have an understanding of which NoC Endpoint determines the Quality of Service (QoS) and therefore, assumes ownership of the path within the DFX flow.

NoC Paths Located Entirely Within a Specific Region

For static or dynamic networks entirely located within a specific region (Network-0 and Network-3), the concept of path ownership is straightforward. In a network entirely positioned within the static region, the path is owned by the static region. Similarly, a network located within the dynamic region is owned by the reconfigurable module in that region. During child implementations of reconfigurable modules, the QoS for the NoC path within the static region remains consistent and is adhered to as defined in the parent implementation. However, for the path owned by the reconfigurable module within the dynamic region, the reconfigurable module can define and configure the QoS parameters for that NoC path.

The following figure shows that for the NoC path entirely located within the static region (Network-0), the DFX flow maintains the entire path with the QoS requirements as initially defined in the parent implementation. In contrast, for the NoC paths that are entirely located within the dynamic region (Network-3), you have the flexibility to choose to reconfigure the entire path. This includes the option to add more NMUs or NSUs, resulting in configurations like Network-3a and Network-3b. In child implementations, you can introduce entirely new NoC paths within the dynamic region, as represented by Network-4.

Figure 158: **NoC Topologies Contained Within One Region**



NoC Paths with NMU and NSU in Separate Regions

For the NoC topologies where endpoints NMU and NSU are in separate regions (Network-1 and Network-2 in Figure 148), you must define the NoC path ownership and adhere to specific DFX rules.

Network-1: NMU in Static Region Accessing NSU in Dynamic Region

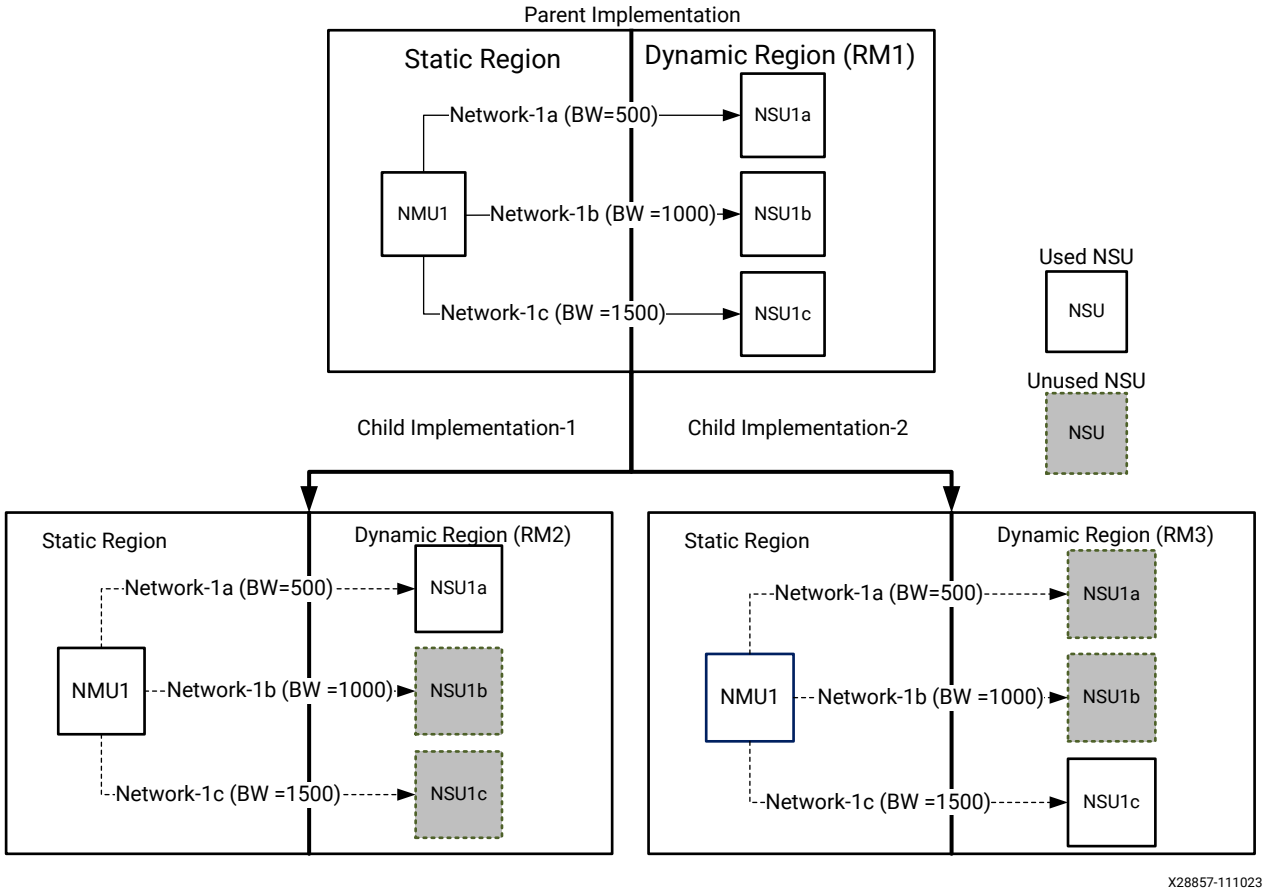
When the static region accesses an NSU in the dynamic region, it is essential for ownership to remain within the static region. In this NoC topology, NMUs are programmed with the destination IDs of the NSUs that the NMUs need to communicate with. Therefore, if an NMU is positioned within the static region, reprogramming the NMU during dynamic reconfiguration is *not* allowed. This means that you must determine the maximum number of NSUs within the dynamic region that the static NMU needs to communicate with, and you must make connections to the static NMU during the initial implementation.

The following figure shows an example of an NMU in the static region that needs to access three NSUs in the dynamic region, each with different bandwidth requirements. However, not all reconfigurable modules (RMs) require the use of all three NSUs. Because the path is owned by the static region, all three networks (Network-1a, Network-1b, and Network-1c) are established and configured with their corresponding Quality of Service (QoS) requirements during the initial implementation. These configurations are preserved and followed in all child implementations.

In each child implementation, you can choose which of the three NSUs (NSU1a, NSU1b, or NSU1c) to use based on the specific QoS requirements of the corresponding RM. You can opt to use all or a subset of NSUs based on the RM requirements. In this example, RM2 variant (child implementation-1) uses NSU1a and leaves NSU1b and NSU1c unused. Conversely, RM3 variant (child implementation-2) uses NSU1c and keeps NSU1a and NSU1b unused.

IMPORTANT! *Because the path is owned by the static region, all three networks are considered to be within their corresponding QoS requirements in all implementations, regardless of whether the corresponding NSU is used or not.*

Figure 159: **NoC Topology for NMUs in Static Region Driving NSUs in Dynamic Region**



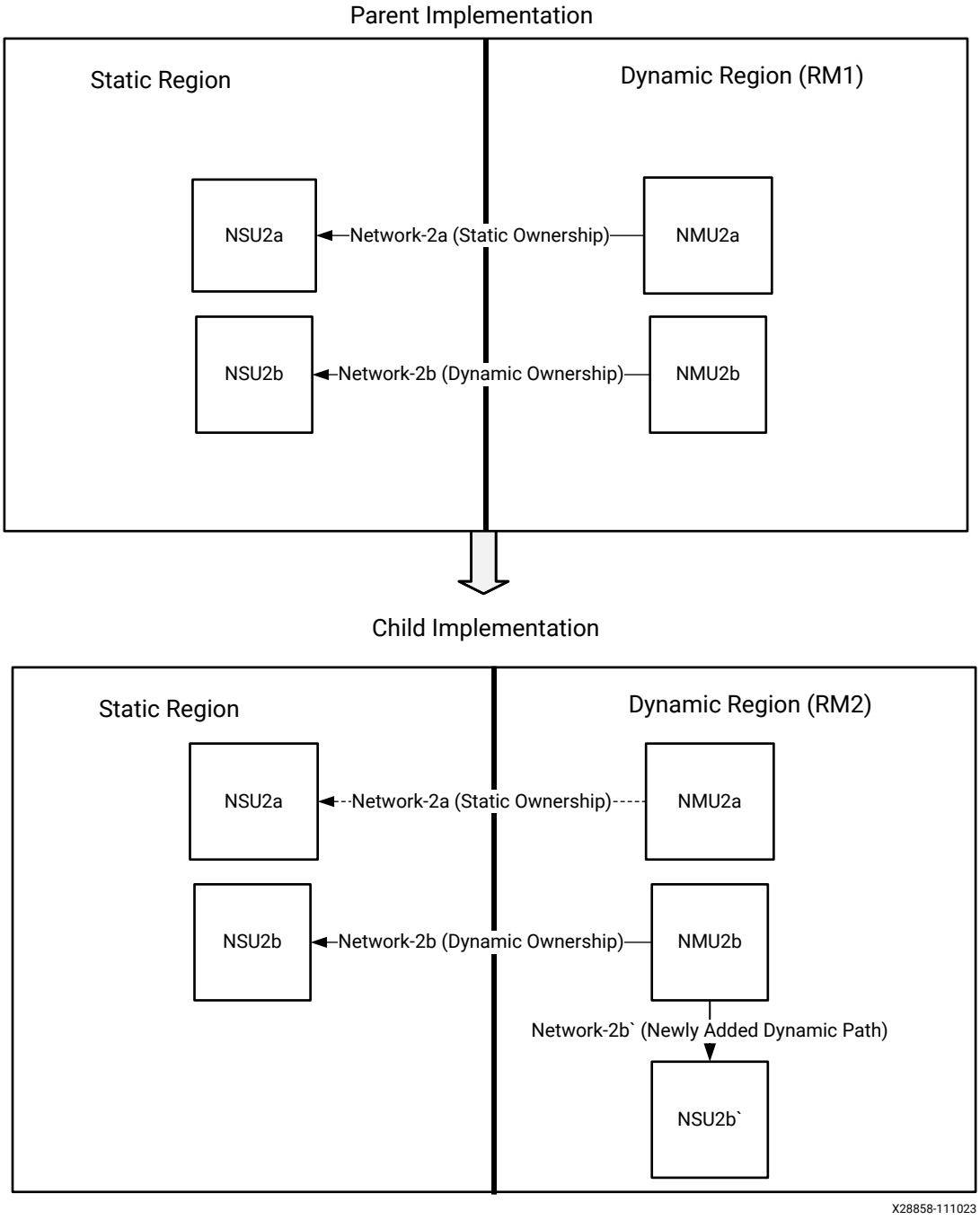
Network-2: NMU in Dynamic Region Accessing NSU in Static Region

When the NMU in the dynamic region accesses the NSU in the static region, you have more flexibility when determining the path ownership. You can either keep the ownership within the static region or within the dynamic region. If the ownership is within the static region, you cannot add any more NSUs in the dynamic region that the NMU can communicate with in subsequent child implementations. However, if the ownership is within the dynamic region, you can add more NSUs in the dynamic region that the dynamic region NMU can communicate with in subsequent child implementations.

In the following figure, Network-2a is a path from the NMU in the dynamic region to the NSU in the static region. This path is configured to be owned by the static region. Therefore, in subsequent implementations, NMU2a retains its existing configuration and does not undergo reconfiguration. The communication capabilities for Network-2a remain limited to NSU2a in any future RM variations.

The following figure also shows Network-2b, which represents a similar communication path extending from the NMU (NMU2b) in the dynamic region to the NSU (NSU2b) in the static region. Unlike Network-2a, this path is configured to be owned by the dynamic region. Therefore, in subsequent child implementations, you have the flexibility to reconfigure this path with different QoS requirements. Additionally, you have the option to expand communication capabilities of this NMU by introducing additional NSUs (NSU2b`) within the dynamic region that NMU2b can communicate with.

Figure 160: **NoC Topology for NMUs in Dynamic Region Driving NSUs in Static Region**



DRCs for NoC Paths Across DFX Boundaries

When working with NoC paths that cross DFX boundaries, you must adhere to the following path ownership rules. If you do not adhere to these rules, Design Rule Check (DRC) violations might occur.

- When an NMU in a static region drives an NSU in a dynamic region, path ownership must stay with the driver in the static region. Failure to adhere to this requirement can trigger the following DRC violation in the Vivado IP integrator:

```
[BD 41-2479] NoC DFX Rule: NoC paths must be owned by the static if
they have a static NoC master (must have INI strategy=load).
To fix: set the INI strategy on INI interfaces for this path to 'load'
```

- For DFX designs that have multiple reconfigurable partitions (RPs), when a boundary path is owned by the dynamic region, the static NSU can be driven by the NMU of only one reconfigurable partition. When there are multiple RPs in a design, both the RPs can be swapped independently in the hardware, and this can lead to conflicting QoS requirements for the static NSU. If you do not adhere to this rule, the IP integrator issues the following DRC violation:

```
[BD 41-2480] NoC DFX Rule: Static NoC slave can connect to at most one RP
where the noc master owns the path (INI strategy=load).To fix: Only have
one RP drive the NoC slave.
```

  ***Note*:** A boundary path can be owned by the dynamic region only if the path has an NMU in a dynamic region driving an NSU in a static region.

- NoC Endpoints of the boundary NoC path can be owned by either the static region or by the dynamic region, *not* both. If you try to connect a NoC Endpoint to multiple INIs of a different strategy, the IP integrator issues the following DRC violation:

```
[BD 41-2477] NoC DFX Rule: NoC Endpoints can only connect to one type of
INI strategy (either driver or load, but not both) at the RP boundary.
```

- NoC paths cannot have a pass-through across dynamic regions if there are no NMUs or NSUs for the path inside the dynamic region. If you fail to adhere to this rule and use an INI interface as an input to the dynamic region directly connected to another INI interface in the output, the IP integrator issues the following DRC violation:

```
[BD 41-2476] NoC DFX Rule: An RP cannot have a NoC passthrough.
To fix: Ensure an RP has a noc endpoint.
```

- A NoC path between multiple reconfigurable partitions must have a static NoC Endpoint to establish the ownership of the path to the static region. Although NoC resources (NMU or NSU) are not used in the static region, you must instantiate the AXI NoC IP in the static region. To establish the ownership of the static region, you must configure the strategy at its ingress (RP0<->Static) and egress (Static<->RP1) INI ports. If you do not adhere to this rule, the IP integrator issues the following DRC violation:

```
[BD 41-2478] NoC DFX Rule: NoC path between two RPs must be static.
To fix: add a noc instance in the static to connect this path,
and have it own the path by setting the input INI strategy to driver, and
the output INI strategy to load.
```

- Incomplete NoC paths are not allowed. You must allocate an NMU or NSU in the dynamic region for the boundary paths even if the NMU or NSU is not used by the specific RM. Except for greybox implementations, you must add the NMU or NSU manually inside the dynamic region. If you do not adhere to this rule, the IP integrator issues the following DRC violation:

```
[BD-41-2616]: Static NSU/MC should be driven by an NMU in the RM.
```

RECOMMENDED: *For any NoC-based designs, AMD recommends keeping some NoC Endpoints inside the RM in the initial implementation so that the tools can establish a more accurate Quality of Service for the NoC traffic specification. Therefore, do not use greybox implementation for the initial implementation of a DFX design, especially when there is a NoC interface between the static region and an RP.*

Greybox Support for NoC

You can use greybox implementation for designs with NOC INI interfaces. For dedicated connections, instead of tie-offs, the pins are left dangling for legal connectivity. This enables you to take advantage of the benefits of greybox PDI, including smaller partial PDI size, lower dynamic power consumption in hardware, and faster PDI download. However, this does not mean that the NoC Endpoints are accessible in the hardware when the greybox partial PDI is loaded. Do not try to access NSUs without attached peripherals, which can lead to unpredictable behavior within the NoC network and might even require a system reset. To ensure a successful operation, make sure your software or hardware applications do not access peripherals in the dynamic region when the greybox partial PDI is being downloaded.

NoC Traffic Impact During Reconfiguration

During reconfiguration, the NoC supports built-in quiescing for the NoC paths where the NMU is in the dynamic region and the NSU is in the static region. This means that the startup and shutdown sequence of the NoC traffic is embedded within the partial PDI, and you do not need to instantiate IP, such as the DFX AXI Shutdown Manager IP or DFX Decoupler IP in the PL. However, you must oversee AXI transactions to and from the RP during reconfiguration to avoid NoC timeout errors. It is the designer's responsibility to pause traffic to and from a Reconfigurable Partition while it is being reconfigured.

The NoC facilitates dynamic ownership of its paths across DFX partitions. This allows subsequent RMs to incorporate additional NMUs in the dynamic region to service specific NSUs in the static region with higher QoS requirements. Consequently, the NoC compiler might offer solutions where physical channels of the NoC are shared between locked static paths and dynamically owned paths. This competition for NoC resources within the same physical channel between AXI transactions in the static region and NoC resources from the dynamic region can introduce unintended delays in static region functionality. You can prevent this scenario by applying the EXCLUSIVE_ROUTING_GROUP constraint to ensure that two NoC networks do not use the same physical channels.

In multiple-RP designs, AMD recommends that you maintain ownership of boundary NoC paths within the static region. This guarantees that the QoS parameters and physical paths of these boundary NoC topologies remain fixed and unaffected by changes from the dynamic region. Allowing dynamic ownership of boundary NoC paths in multiple-RP designs can lead to overlapping NoC solutions due to the absence of enforced routing exclusivity.

Related Information

Logical Decoupling
Network-2: NMU in Dynamic Region Accessing NSU in Static Region

DFX for SSI Technology Versal Devices

With the introduction of support of Versal Premium Series for the DFX flow, stacked silicon interconnect (SSI) technology is supported. Solution requirements are the same for monolithic devices in general, but care must be taken when floorplanning the design around SLR boundaries.

Multi-SLR Dynamic Regions

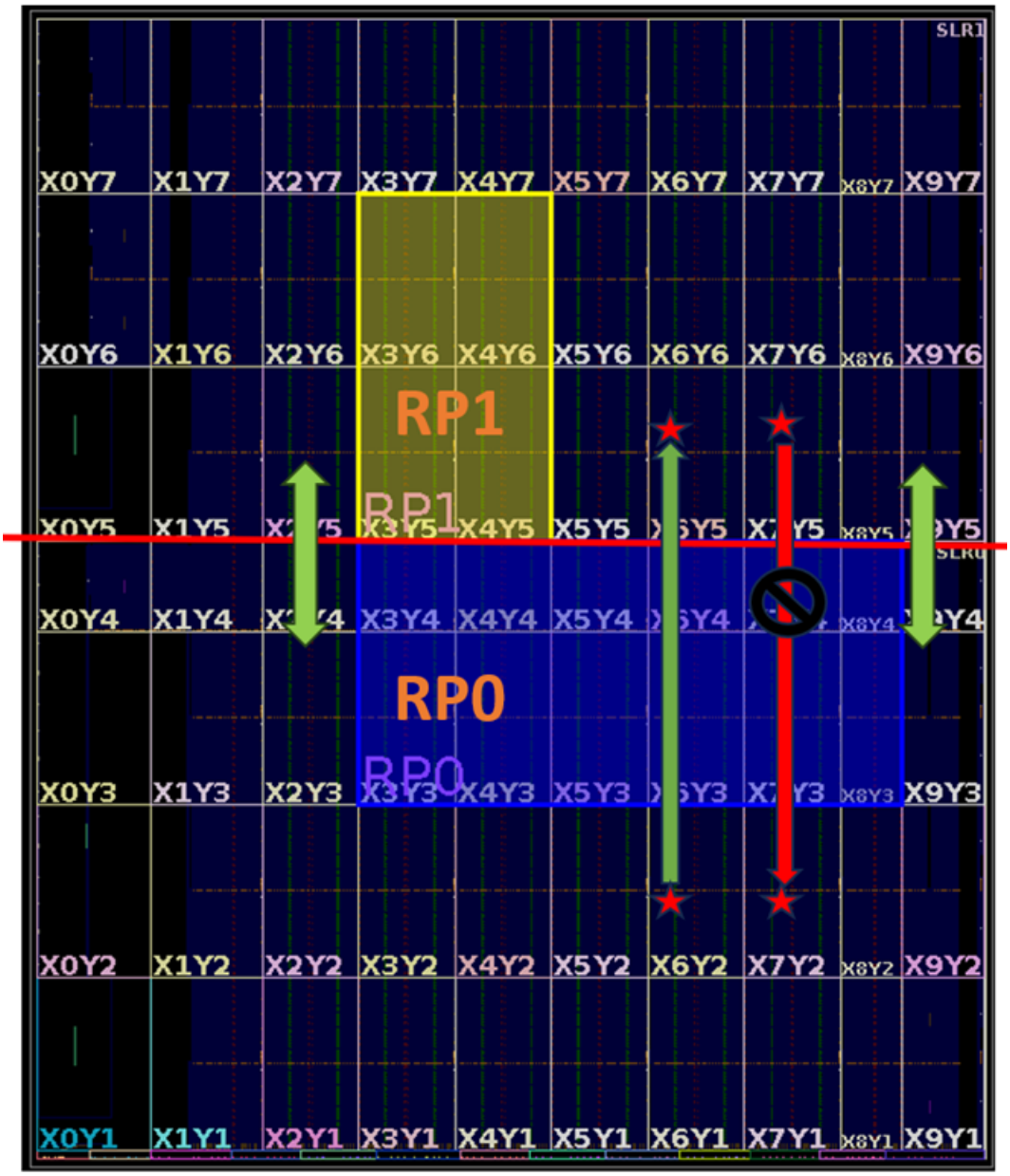
When creating Reconfigurable Partitions that span multiple SLR, it is recommended to include a full clock region above and below the SLR boundary to capture as many super long lines (SLL) within the width of the RP as possible. These SLLs are the routing resources crossing the SLR boundary, and by allowing a full clock region above and below, you improve the routability for that RP. If high-performance modules have difficulty closing timing, consider placing these modules within their own Pblock within the reconfigurable Pblock to ensure the logic is not spread across multiple SLRs.

Static Routing Across SLR Boundaries

In SSI technology Versal devices, the SLL nodes crossing SLRs exit through CLE tiles. The routing of static nets or the static portion of boundary nets (that is, the part of the net between the PPLOC and static region) cannot use the slice route-through after crossing the SLR boundary within a reconfigurable partition (RP). If RP Pblocks span or are next to an SLR boundary, only routes within the reconfigurable modules (RMs) within that RP can cross that area. If non-NoC routes must traverse the SLR boundary, leave space on one or both sides of the RP Pblock to give space for `route_design` to find a solution.

The following figure shows a two-SLR VP1502 device with two RPs. The RP0 and RP1 Pblocks align with the SLR boundary (shown in red), and adequate space is provided on either side (shown in green) to ensure that static routes and static portion of boundary nets can cross the SLR boundary without any issues. Even though the DFX expanded routing feature expands left and right of the RP to obtain more resources, expanded routing does not use the SLLs in the expanded region, leaving those resources for the static design. The routing for the SLR0<--SLR1 static net is highlighted in red, because it requires the CLE inside RP0 to exit the SLL node, which is not allowed. The SLR0-->SLR1 static net is highlighted in green, because CLE resources from the static region are available in SLR1 to exit the SLL node.

Figure 161: **Two-SLR VP1502 Device with Two Reconfigurable Partitions**



When there are insufficient SLL nodes available to cross the SLR boundary, `route_design` fails with an SLL assignment error. The SLL assignment phase in `route_design` provides an SLL Capacity Report, which can be used debug to the error. The report is divided into the following tables:

- **SLL Capacity Report (DFX):** The information captured in this table is based on the DFX floorplan and shows the number of available SLLs per SLR pair for static routing and for each RP routing.
- **Net Demand Report (DFX):** The information captured in this table is calculated based on the DFX design placement and it shows static nets and RP nets.
- **Utilization Report (DFX):** This table displays the utilization of SLL, calculated as Utilization = Demand / Capacity. A value greater than 1 signifies that the demand (requirement) exceeds the number of available SLL nodes for the given DFX floorplan.

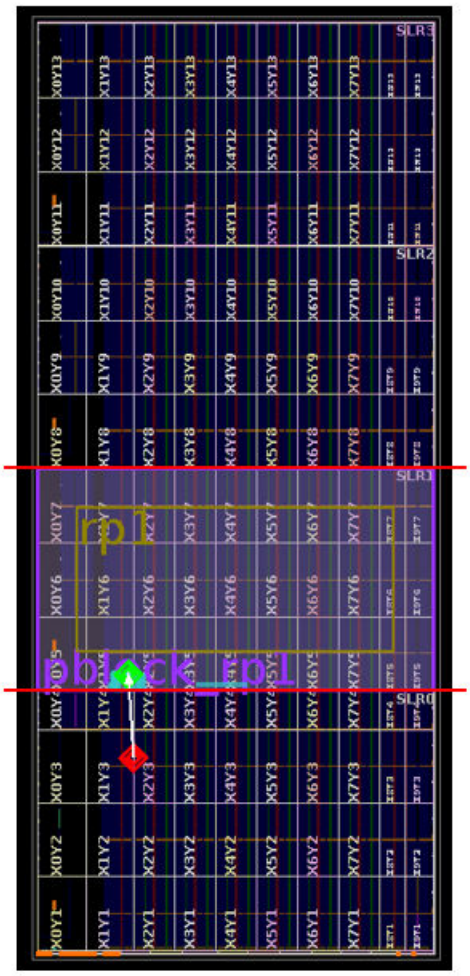
For each SLR boundary, the crossing capacity and the number of nets crossing are detailed per direction (North and South) and combined for both directions.



TIP: *You can use the following Tcl command on a placed design to get all the SLR crossing nets in the design:* `get_nets -hierarchical -top_net_of_hierarchical_group -filter { CROSSING_SLRS =~ "*SLR*" }`

The following figure shows an example floorplan for one RP design that covers the complete SLR (SLR1) and highlighted boundary net from static region to RP, which requires an SLR crossing from SLR0-->SLR1.

Figure 162: **Example Floorplan of One Reconfigurable Partition**



In this example, the design fails during `route_design` and reports the following in the log file.

Figure 163: **Log File for Example Floorplan of one Reconfigurable Partition**

SLL Capacity Report (DFX)

SLR Cut	static	rp1
[0->1]	0	18870
[0<-1]	15096	0
[0--1]	15096	18870
[1->2]	15096	0
[1<-2]	0	18870
[1--2]	15096	18870
[2->3]	15096	0
[2<-3]	15096	0
[2--3]	15096	0

Net Demand Report (DFX)

SLR Cut	static	rpl
[0->1]	267	0
[0<-1]	48	0
[0--1]	315	0
[1->2]	0	0
[1<-2]	0	0
[1--2]	0	0
[2->3]	0	0
[2<-3]	0	0
[2--3]	0	0

Utilization Report (DFX)

SLR Cut	static	rpl
[0->1]	268.00	0.00
[0<-1]	0.00	1.00
[0--1]	0.02	0.00
[1->2]	0.00	1.00
[1<-2]	1.00	0.00
[1--2]	0.00	0.00
[2->3]	0.00	1.00
[2<-3]	0.00	1.00
[2--3]	0.00	1.00

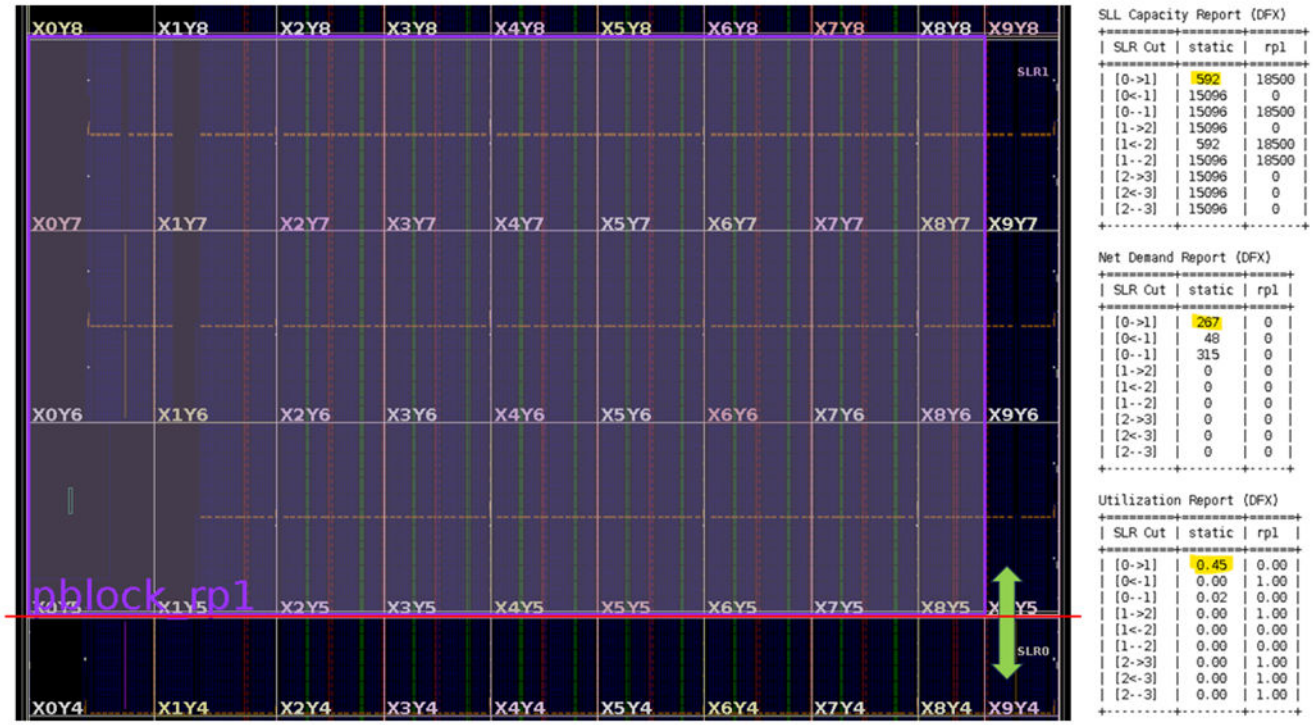
In the SLL capacity Report (DFX), the highlighted section indicates that the number of SLL nodes available for static nets is zero. Based on the DFX floorplan, routing through the CLE, which is part of RP, is not possible.

In the Net Demand Report (DFX), static requires 267 nets to route from 0->1 (SLR0 to SLR1). This is prohibited because the net crossing the SLR boundary needs to route through the CLE in RP.

The only way to cross the SLR is to leave space at the device edges to allow the boundary nets to route through the CLE. This is a design-dependent decision.

Following is an example of a routed design with space left for static nets and boundary nets to cross the SLR.

Figure 164: **Routed Design with Static Nets and Boundary Nets Crossing the SLR**



USER_SLL_REG

USER_SLL_REG instructs the implementation tool to place the register in a location that is optimal for connections to the SLL. This is a dedicated register that drives data to, or receives data from the SLL. In DFX, the USER_SLL_REG property can be used for SLL guidance and PPLOC assignments. The cells where the USER_SLL_REG property is set to be true must be register cells and should have FD/Q → FD/D connectivity. The property should be set on both source and destination registers. For more information on USER_SLL_REG, see the *Vivado Design Suite Properties Reference Guide* (UG912).

The following are the use cases to show how the USER_SLL_REG is used in a DFX design:

- RM – RM path across SLR
- Static to RM path across SLR
- Abstract shell RM implementation

RM – RM Path Across SLR

In this design, source FlipFlop in SLR0 and destination FlipFlop in SLR1 are in different RMs. In parent implementation run, you must apply the USER_SLL_REG property on both cells in different SLRs to guide the optimal SLL usage. The tool applies PPLOC on the UBUMP and the LAG_OUT node.

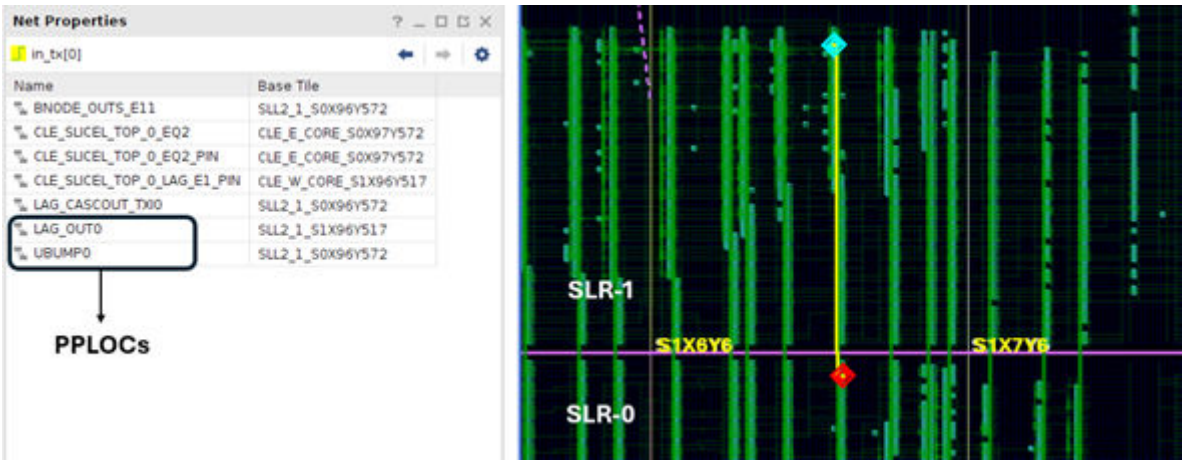
Figure 165: **Schematic View of Net Crossing SLRs (FF/Q → FF/D)**




```
report_route_status -of_objects [get_nets rp_slr0_dut_inst/
crossing_0_1_inst/in_tx[0]]
------------------------------------------------------------
Route information for rp_slr0_dut_inst/crossing_0_1_inst/in_tx[0]
  Route status: ROUTED
  This net is fully routed
------------------------------------------------------------
  The route tree for this net is:
    Route Tree:
    ----------------------------------------
    Subtree: 0
    [{       CLE_E_CORE_S0X97Y572/CLE_SLICEL_TOP_0_EQ2_PIN (65535)
             CLE_E_CORE_S0X97Y572/CLE_SLICEL_TOP_0_EQ2
( 0) CLE_E_CORE_S0X97Y572/CLE_E_CORE.CLE_SLICEL_TOP_0_EQ2_PIN-
>>CLE_SLICEL_TOP_0_EQ2
             SLL2_1_S0X96Y572/BNODE_OUTS_E11 ( 6) SLL2_1_S0X96Y572/
SLL2.LOGIC_OUTS_E14->>BNODE_OUTS_E11
             SLL2_1_S0X96Y572/LAG_CASCOUT_TXI0 ( 1) SLL2_1_S0X96Y572/
SLL2.BNODE_OUTS_E11->>LAG_CASCOUT_TXI0
          p         SLL2_1_S0X96Y572/UBUMP0 ( 0) SLL2_1_S0X96Y572/
SLL2.LAG_CASCOUT_TXI0->>UBUMP0
          p       SLL2_1_S1X96Y517/LAG_OUT0 ( 0) SLL2_1_S1X96Y517/
SLL2.UBUMP0->>LAG_OUT0
        }]  CLE_W_CORE_S1X96Y517/CLE_SLICEL_TOP_0_LAG_E1_PIN
( 0) CLE_W_CORE_S1X96Y517/CLE_W_CORE.CLE_SLICEL_TOP_0_LAG_E1-
>>CLE_SLICEL_TOP_0_LAG_E1_PIN
    ----------------------------------------
============================================================
```

```
set_property USER_SLL_REG 1 [get_cells rp_slr0_dut_inst/crossing_0_1_inst/
in_rx_reg[0]]
set_property USER_SLL_REG 1 [get_cells rp_slr1_dut_inst/crossing_1_0_inst/
in_tx_reg[0]]
```

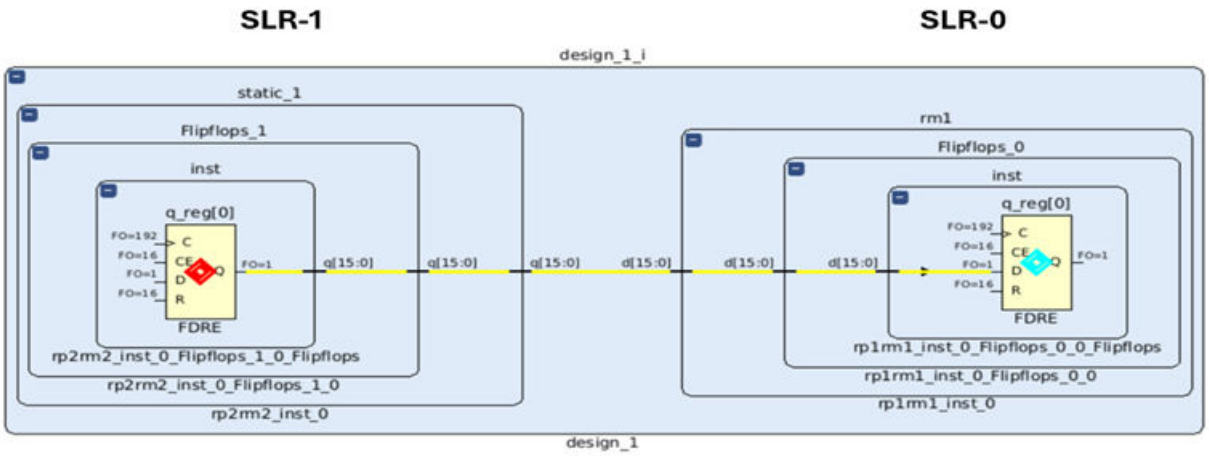
Figure 166: **Device View of the SLR Crossing Net After Applying the USER_SLL_REG Property**



Static to RM Path Across the SLR

In this design, the source FlipFlop is static in SLR1, and the destination FlipFlop is RM in SLR0. In the parent implementation run, you must apply the USER_SLL_REG property on both cells in different SLRs to guide the optimal SLL usage. For the RM boundary pin the tool applies the PPLOC on LAG_OUT node.

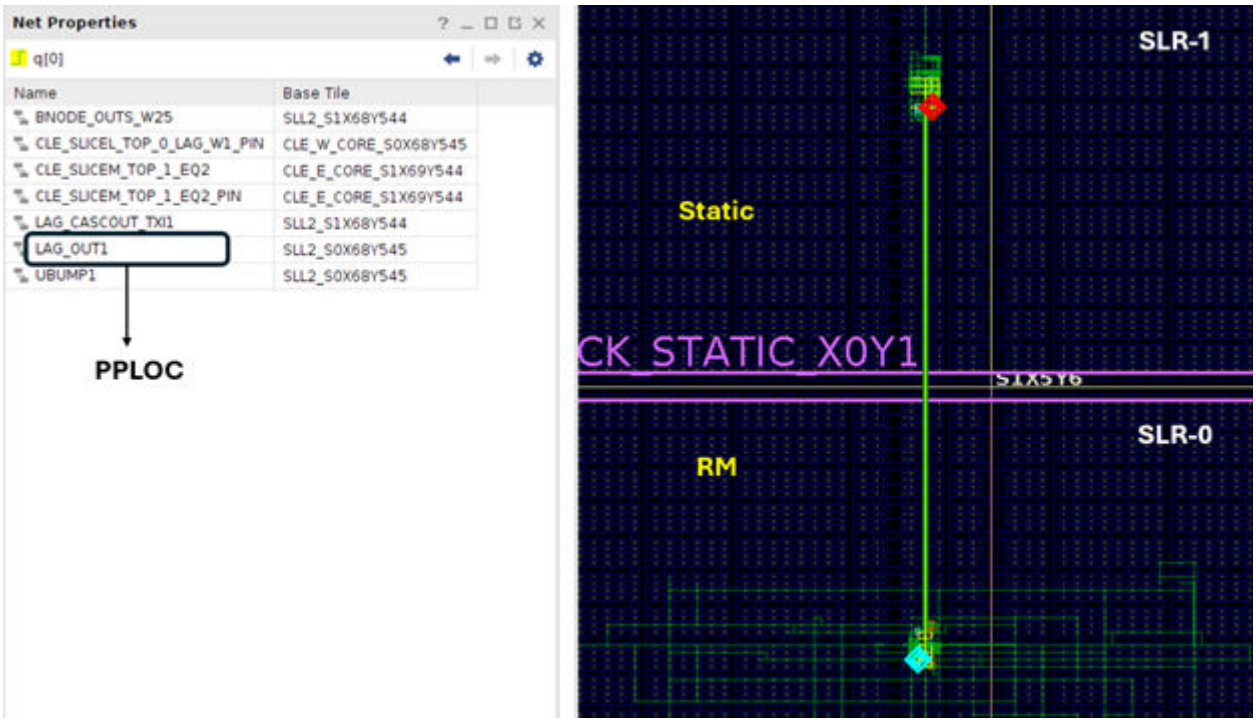
Figure 167: **Schematic of Static to RM SLR Crossing Net**




```
report_route_status -of_objects [get_nets design_1_i/static_1/Flipflops_1/
inst/q[0]]
------------------------------------------------------------
Route information for design_1_i/static_1/Flipflops_1/inst/q[0]
  Route status: ROUTED
  This net is fully routed
------------------------------------------------------------
  The route tree for this net is:
    Route Tree:
    ----------------------------------------
```

```
    Subtree: 0
    [{       CLE_E_CORE_S1X69Y544/CLE_SLICEM_TOP_1_EQ2_PIN (65535)
             CLE_E_CORE_S1X69Y544/CLE_SLICEM_TOP_1_EQ2
( 0) CLE_E_CORE_S1X69Y544/CLE_E_CORE.CLE_SLICEM_TOP_1_EQ2_PIN-
>>CLE_SLICEM_TOP_1_EQ2
              SLL2_S1X68Y544/BNODE_OUTS_W25 ( 6) SLL2_S1X68Y544/
SLL2.LOGIC_OUTS_E38->>BNODE_OUTS_W25
             SLL2_S1X68Y544/LAG_CASCOUT_TXI1 ( 2) SLL2_S1X68Y544/
SLL2.BNODE_OUTS_W25->>LAG_CASCOUT_TXI1
                      SLL2_S0X68Y545/UBUMP1 ( 1) SLL2_S1X68Y544/
SLL2.LAG_CASCOUT_TXI1->>UBUMP1
           p        SLL2_S0X68Y545/LAG_OUT1 ( 0) SLL2_S0X68Y545/SLL2.UBUMP1-
>>LAG_OUT1
         }]  CLE_W_CORE_S0X68Y545/CLE_SLICEL_TOP_0_LAG_W1_PIN
( 0) CLE_W_CORE_S0X68Y545/CLE_W_CORE.CLE_SLICEL_TOP_0_LAG_W1-
>>CLE_SLICEL_TOP_0_LAG_W1_PIN
    ----------------------------------------
============================================================
```

Figure 168: **Device View of the SLR Crossing Net After Applying the USER_SLL_REG Property**



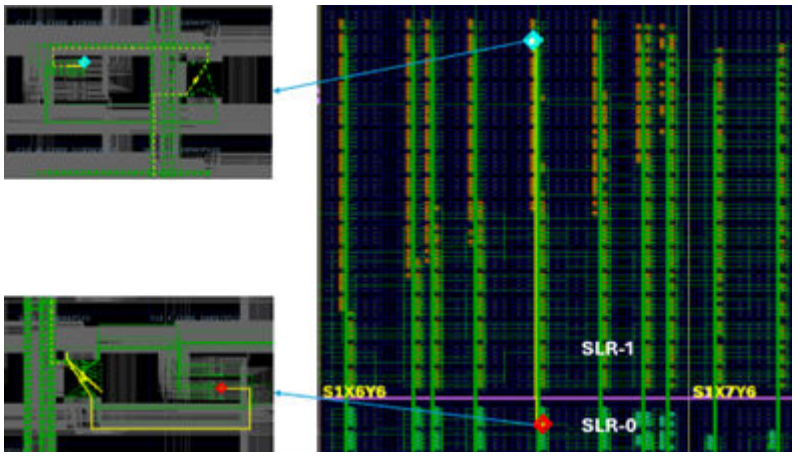
Abstract Shell RM Implementation

As explained previously, if the USER_SLL_REG property is used in the parent implementation, it must be reapplied in the abstract shell implementation only to the RM cell. If the property is not applied, the tool might not place the register in the dedicated sites that drive or are driven by the SLL node.

```
report_route_status -of_objects [get_nets rp_slr0_dut_inst/
crossing_0_1_inst/in_tx[0]]
------------------------------------------------------------
Route information for rp_slr0_dut_inst/crossing_0_1_inst/in_tx[0]
  Route status: ROUTED
  This net is fully routed, and has locked routing nodes
------------------------------------------------------------
  The route tree for this net is:
    Route Tree:
    ----------------------------------------
    Subtree: 0
    [{       CLE_E_CORE_S0X97Y572/CLE_SLICEL_TOP_0_EQ2_PIN (65535)
             CLE_E_CORE_S0X97Y572/CLE_SLICEL_TOP_0_EQ2
( 0) CLE_E_CORE_S0X97Y572/CLE_E_CORE.CLE_SLICEL_TOP_0_EQ2_PIN-
>>CLE_SLICEL_TOP_0_EQ2
             SLL2_1_S0X96Y572/BNODE_OUTS_E11 ( 6) SLL2_1_S0X96Y572/
SLL2.LOGIC_OUTS_E14->>BNODE_OUTS_E11
             SLL2_1_S0X96Y572/LAG_CASCOUT_TXI0 ( 1) SLL2_1_S0X96Y572/
SLL2.BNODE_OUTS_E11->>LAG_CASCOUT_TXI0
      *    p        SLL2_1_S0X96Y572/UBUMP0 ( 0) SLL2_1_S0X96Y572/
SLL2.LAG_CASCOUT_TXI0->>UBUMP0
      *           SLL2_1_S1X96Y517/LAG_OUT0 ( 0) SLL2_1_S1X96Y517/
SLL2.UBUMP0->>LAG_OUT0
      *  }]  CLE_W_CORE_S1X96Y517/CLE_SLICEL_TOP_0_LAG_E1_PIN
( 0) CLE_W_CORE_S1X96Y517/CLE_W_CORE.CLE_SLICEL_TOP_0_LAG_E1-
>>CLE_SLICEL_TOP_0_LAG_E1_PIN
    ----------------------------------------
============================================================
```

The following figure illustrates the route status of an SLR crossing net after the abstract shell implementation. In this case, the USER_SLL_REG property is applied to the RM cell that drives the net, and the tool places the RM cell in the cell that directly drives the used SLL node.

Figure 169: **Device View of the SLR Crossing Net After Applying the USER_SLL_REG Property**



Static Routing Across SLR Boundaries Using USER_SLL_GUIDANCE

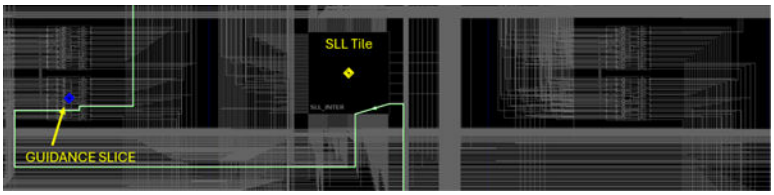
The `USER_SLL_GUIDANCE` is an advanced DFX property that lets users control which configurable logic element (CLE) exit sites are used by static net for super logic region (SLR) crossings. This property is intended for designs where the standard methodology of leaving one clock region at SLR boundaries for minimum static net crossing is not feasible.

To use the `USER_SLL_GUIDANCE` property, you must understand the static netlist placement across various SLRs and the number of static nets requiring SLR crossing. The number of static nets that can cross an SLR depends on the number of UBUMP nodes available in that super long lines (SLL) tile. You must remove the programmable unit (PU) of the CLE from the reconfigurable partition (RP) Pblock so that it becomes part of the static region and can be used for SLR exit.

***Note*:** `USER_SLL_GUIDANCE` is a hard constraint. If the router cannot honor the provided guidance, it errors out and stops routing.

Use the following methodology to apply `USER_SLL_GUIDANCE`:

1. Analyze static net placement. Identify the static nets that need to cross SLR boundaries and determine the SLRs they traverse.
2. Identify valid SLL tiles. Based on the placement of static logic, get the valid SLL tiles that can cross the SLR.
3. Identify exit slice sites. Determine the slice sites adjacent to the identified SLL tile that serve as exit points for the SLL nodes.
4. Resize the RP Pblock. Remove the PU of the identified slice site(s) from the RP Pblock so they become part of the static region.
5. Apply the `USER_SLL_GUIDANCE` property to the static net.



Syntax

- ```
  resize_pblock <rp_pblock> -remove "[get_sites -of_objects [get_dfx_footprint -pu -of_objects [get_tiles -of_objects [get_sites <SLICE_site>]]]"
  ```
- ```
  set_property USER_SLL_GUIDANCE
  "<SLICE_site>"
  [get_nets <static_slrcross_net>]
  ```

Examples

The following examples show how to apply `USER_SLL_GUIDANCE`:

1. Single SLR crossing (static-to-static net):

   ```
   set_property USER_SLL_GUIDANCE
   SLICE_X145Y333
   [get_nets net_1]
   ```

2. Multi-SLR crossing net: For nets that cross multiple SLR boundaries, the guidance must include exit sites in each SLR along the path:

   ```
   set_property USER_SLL_GUIDANCE
   "<SLICE_site_SLRm> <SLICE_site_SLRn>"
   [get_nets <static_slrcross_net>]
   ```

The following example shows a static net that requires SLR crossing in SLR1 through `SLICE_X145Y333` and SLR2 through `SLICE_X144Y621`:

```
set_property USER_SLL_GUIDANCE
"SLICE_X145Y333 SLICE_X144Y621"
[get_nets net_2]
```

Hardened DDR Memory Controllers

Hardened DDR memory controllers (DDRMCs) in Versal is implemented using NoC IP Wizard. User can choose to keep DDRMCs in either static region or reconfigurable partition. There are certain recommendation while using DDRMCs in a DFX design. They are:

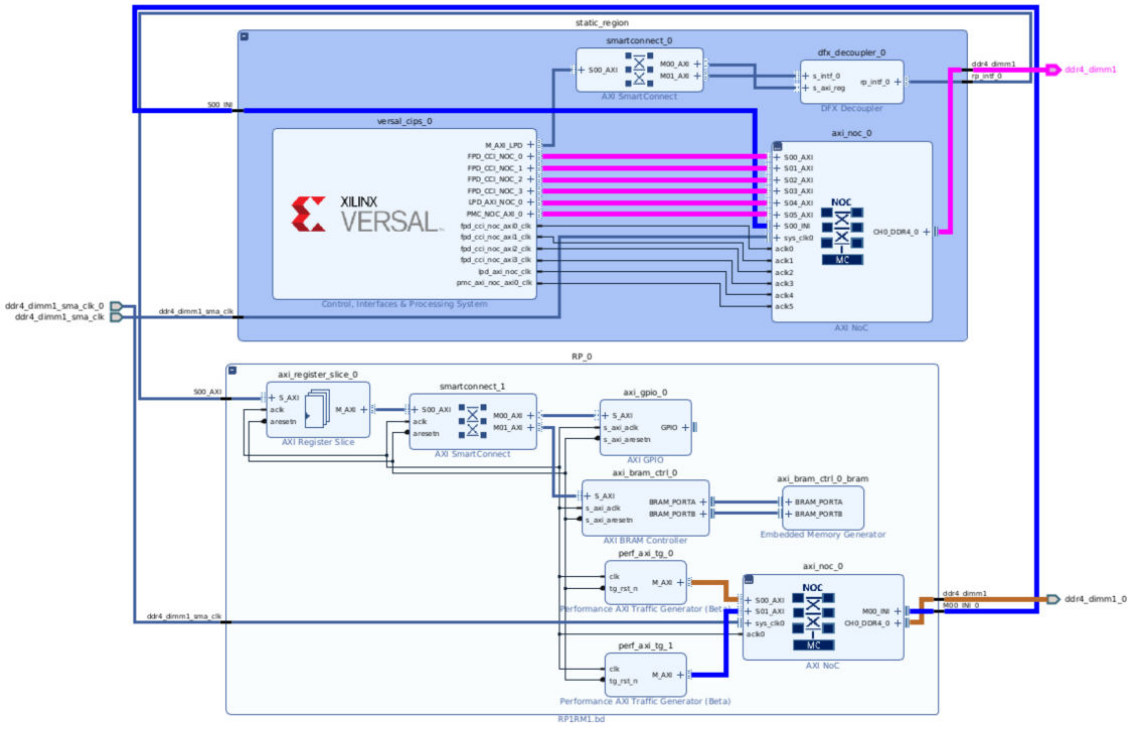
- It is not recommended to share a DDRMC between static region and reconfigurable partition or between multiple reconfigurable partitions.

- If a DDRMC port is shared between static and reconfigurable module (RM), traffic class and bandwidth requirements on static path can impact the traffic class and bandwidth of RM. User should consider overall static and RM usage requirement of NoC and allocate just enough bandwidth and QoS for static so that reconfigurable module gets more flexibility to meet its requirement. A NoC channel supports only two traffic classes. Hence if two traffic classes are already consumed by static region, no new traffic class can be used by the reconfigurable module. Please refer to *Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313) to learn more about Traffic Class, Virtual Channel and QoS of NoC.
- If certain use cases demand sharing of DDRMC between static and reconfigurable partition, NoC INI based connection can be used at static-RM interface for logic to communicate to DDRMC in other partition.
- When DDRMC is instantiated in the reconfigurable partition, include the necessary resources in the corresponding reconfigurable Pblock.

  Given below is a block diagram representation of Hard DDRMC usage in a DFX design. Three different usage of DDRMCs are shown below. They are:

  - Static Logic accessing Static DDRMC. This is shown in the magenta color where CIPS in static region is accessing DDRMC in static region using AXI NOC Interface IP.
  - AXI Traffic Generator in reconfigurable partition writing data to DDRMC inside the reconfigurable partition itself. This path is highlighted in brown color.
  - AXI Traffic Generator in Reconfigurable partition writing data to DDRMC in static region using NoC Inter-NoC-Interconnect (INI). This path is shown in dark blue color.

Figure 170: **Hardened DDR Memory Controllers**



Logical Decoupling

Because the reconfigurable logic is modified while the device is operating, the static logic connected to outputs of RM must ignore the data from RM during partial reconfiguration. The RMs do not provide valid output data until partial reconfiguration is complete and the reconfigured logic is initialized. It is not possible to predict or simulate the functionality of the RM. Logical decoupling isolates the dynamic part of the design from the static, ensuring no unintended activity disrupts the static design.

There are number of boundary types where logical decoupling should be inserted, based on the connectivity of the RP, and there are different strategies for each scenario. Boundaries can be within the PL, within the NoC, or at the PS-PL boundary.

At the end of partial reconfiguration a GSR event is triggered to initialize all the logic in the reconfigured region. For very large reconfigurable partitions, especially those spanning multiple super logic regions (SLR) on multi-die devices, this global set-reset can take multiple clock cycles to propagate through all the logical elements in that region. Because of this duration, it is possible that some circuitry might be released before others, potentially leading to unintended behavior. AMD recommends to employ techniques such as enabled clocks, local synchronous resets and/or safe state machines to ensure that all dynamic logic starts together and functions as intended.

Logical Decoupling at the RP Boundary

A common design practice to mitigate this issue involves registering all output signals on the static side of the interface from the RP. An enable signal isolates the logic until it is completely reconfigured. Other approaches include a simple 2-to-1 MUX on each output port or higher-level bus controller functions. In addition, using clock buffers with muxes (for example, BUFGMUX) or enables (for example, BUFGCE) on clock signals entering the dynamic region allows pausing any activity driven by those static clocks while reconfiguration occurs. This prevents logic from moving on from its initial state until the reconfigurable module is completely loaded and released for operation.

Two pieces of IP are available from AMD to provide decoupling capabilities in the PL. The DFX Decoupler IP allows you to insert multiplexers to easily and efficiently decouple AXI4-Lite, AXI4-Stream, and custom interfaces. This IP disables key signals to prevent unwanted activity on the RP boundary. The DFX AXI Shutdown Manager IP provides a more intelligent way to decouple AXI interfaces, offering different responses to requests rather than holding boundaries constant. Click here for more information about the DFX Decoupler.

Logical Decoupling in the NoC

If the boundary falls within the NoC, NoC quiescing is automatic only if the NMU/NSU is inside the RP. If the NoC/PL AXI interface is at the RP boundary, a logical decoupler is still required. In such a case, it is recommended to put the interface NoC in the RP to avoid the use of the AXI decoupler. Then the connection across partitions is no longer AXI-based but based on the NoC internal path through INI.

Although NoC quiescing occurs automatically and allows you to bypass IP that uses PL, such as the DFX AXI Shutdown Manager or DFX Decoupler, you must oversee AXI transactions to and from the RP during reconfiguration. Outstanding AXI transactions across DFX partitions during reconfiguration can lead to NoC timeout errors at NMU or NSU in the static domain. Additionally, prolonged delays on `ready` signals are identified by NMU/NSU, potentially causing AXI handshake timeouts. You can examine the NMU/NSU interrupt status register (ISR) to assess if the interrupt register based on timeouts is high.

Logical Decoupling at the PS-PL Interface

If the boundary between static (PS) and dynamic (PL) is specifically at the PS-PL interface, the following use cases apply:

- If the entire PL is included in the dynamic region, partial reconfiguration includes a power domain shutdown. This temporarily powers down the entire PL. The Segmented Configuration PL reload use case specifically covers this scenario.
- If a portion of the PL is included in the dynamic region and abuts the PS, you must manage decoupling from the PS domain. This means flushing transactions and pausing activity on the reconfigured PL-side interfaces to ensure no system lockup occurs.
- The static design should include the functional behavior required for the data and interface management. It can implement mechanisms such as handshaking or disabling interfaces, which might be required for bus structures to avoid invalid transactions. Also consider the down-time performance effect of a dynamic module. This is the unavailability of any shared resources in the dynamic module during or after reconfiguration. This is the most common scenario.

ECO Flow Support for DFX Designs

You can use engineering change orders (ECOs) to modify a post-implementation netlist with minimal impact to the original design. The AMD Vivado™ Design Suite includes an ECO flow that lets you open a design checkpoint, make changes, generate reports on the updated netlist, and produce programming files. For more information on basic ECO operations, see the *Vivado Design Suite User Guide: Implementation* (UG904).

Follow these steps to run the ECO flow:

1. Open a previously implemented design.
2. Modify the netlist using the ECO edit commands.
3. If the design is not fully placed, run the placer in ECO mode with the `place_design -eco` command.
4. If the design is fully placed and the ECO placer was not previously called, use the `finalize_eco` command to fix any illegal site configuration.
5. Run the router in ECO mode by using the `route_design -eco` command.
6. Generate the device image.

In ECO mode, the placer places the unplaced or newly added cells. Keep the scope of your changes to less than 1% of the total cells in the design. New cells are limited to the addition of flip-flops and LUTs.

***Note*:** For Dynamic Function eXchange (DFX) designs, you can edit ECOs in child implementations within the reconfigurable module (RM). Static modifications are not supported.

Debugging Versal Device DFX Designs

Versal devices provide more capability for you to debug your designs in hardware. This includes JTAG based debug as well as high speed debug protocol (HSDP) using GT transceivers or PCI® Express. For debugging DFX designs in Versal, you must take additional steps to ensure proper connectivity to debug cores like ILA, VIO that are contained within both the static region and the reconfigurable partition. For all DFX designs, you should instantiate an instance of the AXI Debug Hub IP with connectivity to the Versal CIPS IP inside each design partition, both static and reconfigurable, that might contain debug cores. The AXI Debug Hub IP instantiated in each design partition is used by the debug flow for the connectivity infrastructure to all debug cores (ILA, VIO, etc) contained within that design partition.

AMD recommends using NoC INI (Inter-NoC-Interconnect) interface across static-RM boundary to communicate to AXI Debug Hub in reconfigurable partition. This is preferred because isolation is built into the NoC architecture.

***Note*:** Accessing the AXI Debug Hub in an RP across a PL based DFX decoupler requires manual intervention. Contact AMD for more information.

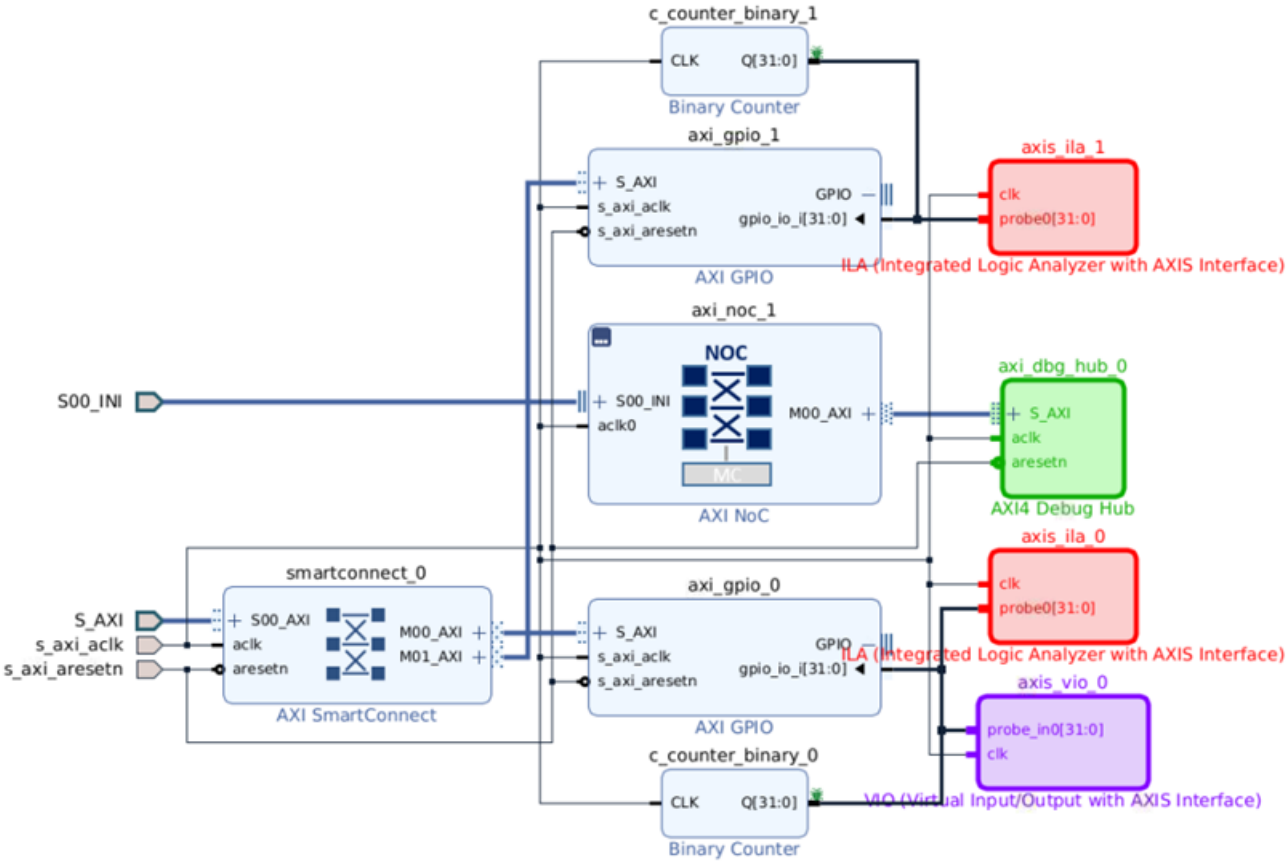
Adding Debug Cores for Versal Devices

For Versal device DFX designs, you can instantiate or insert debug cores. In either case, an AXI Debug Hub IP must be added to the design in the target partition where debug is desired. The debug hub must exist in each reconfigurable module to accommodate any debug cores in that RM. Each RM in the parent configuration must include a debug hub to establish the debug infrastructure that is used in any child configurations.

Instantiation

When using instantiation, manually add debug cores, and connect them to signals you want to probe. In the following block design, Debug Hub IP (green) was added and connects to the NoC. Multiple ILA (red) and VIO (purple) cores were added to monitor two counter IP. Green bug icons were added to the signals to be probed by the ILA cores. By explicitly adding the ILA cores and adding probe points, you can define all the debug details early in the design flow.

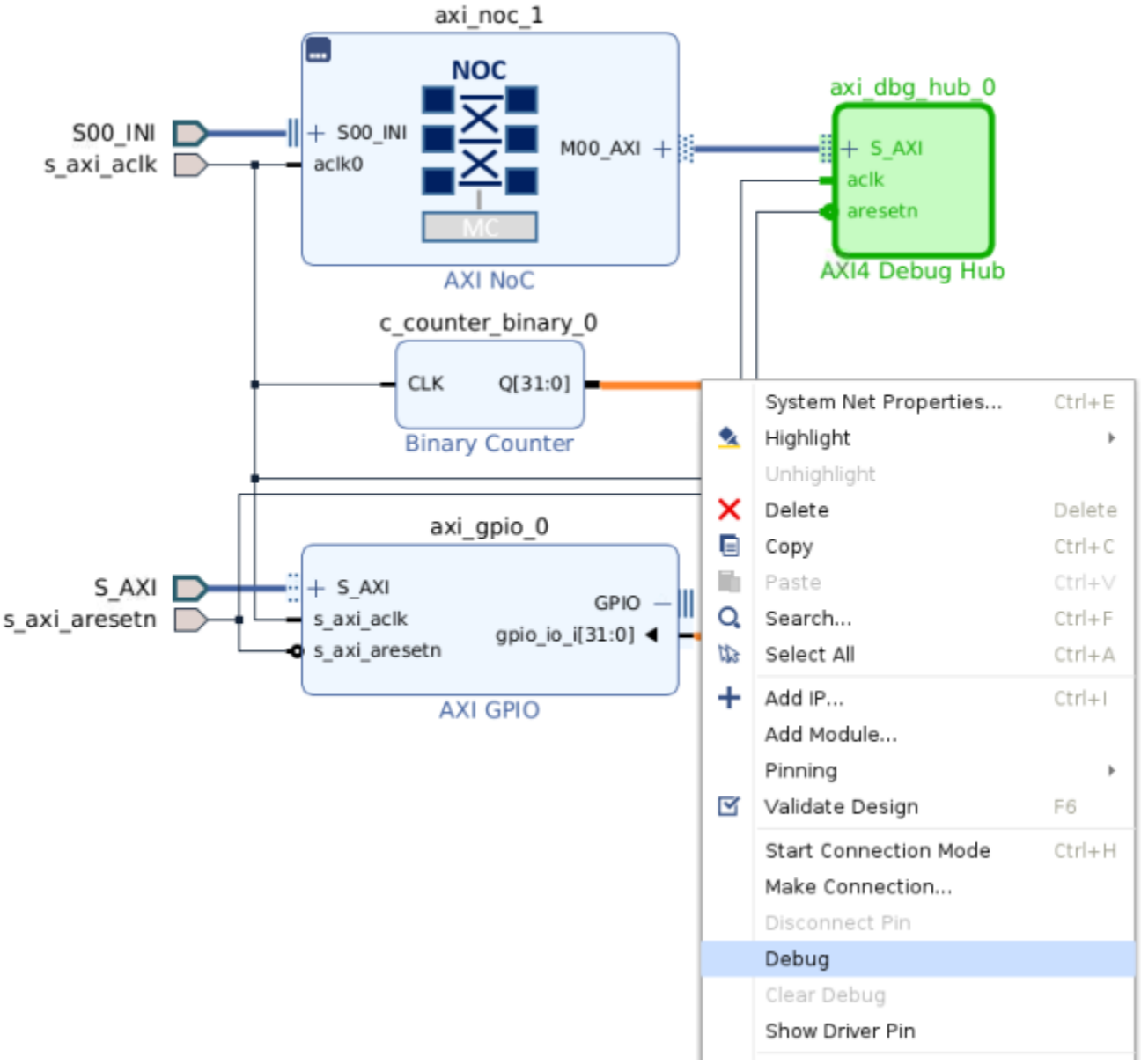
Figure 171: **Instantiation Example**



Insertion

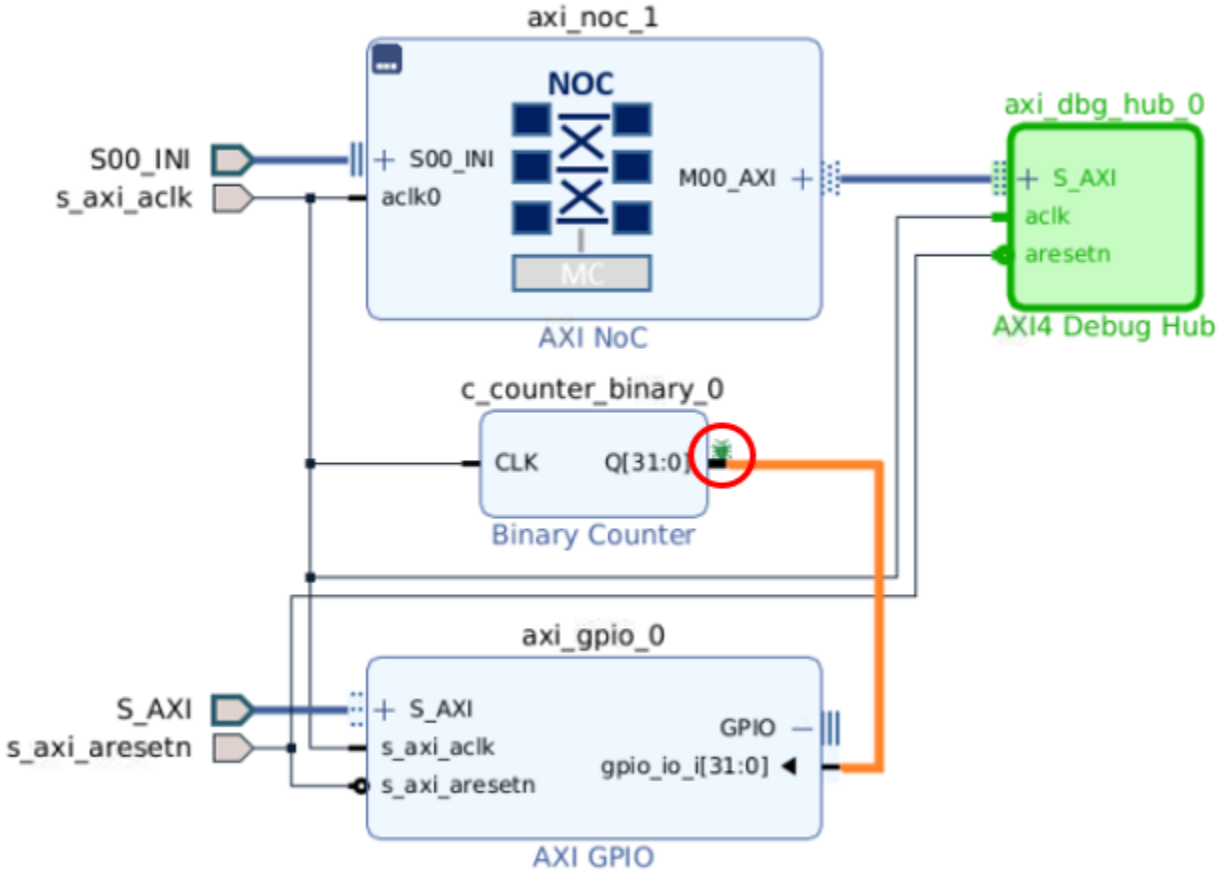
When using insertion, signals are identified in the block design or RTL source, and insertion of the ILA debug core is done later in the flow. To add a debug tag to a signal on a block design canvas, right-click and select **Debug**, as shown in the following figure.

Figure 172: **Insertion Example Before**



After you select Debug, the green bug icon is added to the signal, as shown in the following figure.

Figure 173: **Insertion Example After**



In the Tcl Console, this action appears as adding the DEBUG property to that net:

```
set_property HDL_ATTRIBUTE.DEBUG true [get_bd_nets {c_counter_binary_0_Q}]
```

The equivalent process within RTL is done using the MARK_DEBUG attribute. When RTL code that uses this attribute is added to a block design as a module reference, and the Debug Hub IP has been added, the same insertion technique is used. Here is the attribute applied in Verilog:

```
(* mark_debug = "true" *) reg [31:0] count_out;
```

Note: In each of these cases, the Debug Hub IP (green) was explicitly added to the BD canvas and connected to the NoC.

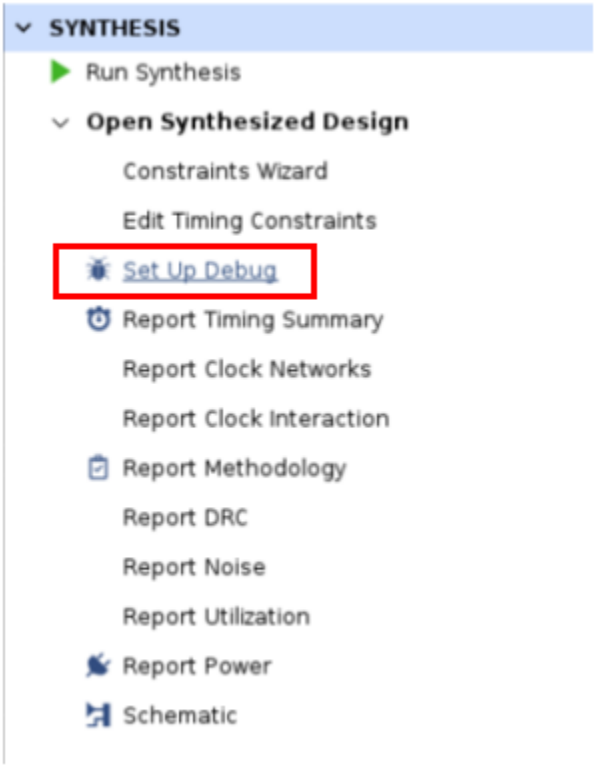
Setting Up Inserted Debug Cores

After synthesis, you can identify details about the debug features. This step is necessary for the insertion flow but not the instantiation flow. The insertion flow allows you to defer selection of which signals to probe and the customization of the ILA core itself.

After synthesis completes, open the synthesized design. This post-synthesis view of the parent run can be used for managing the DFX floorplan, because it shows the full design hierarchy for the primary configuration. With this view open, do the following:

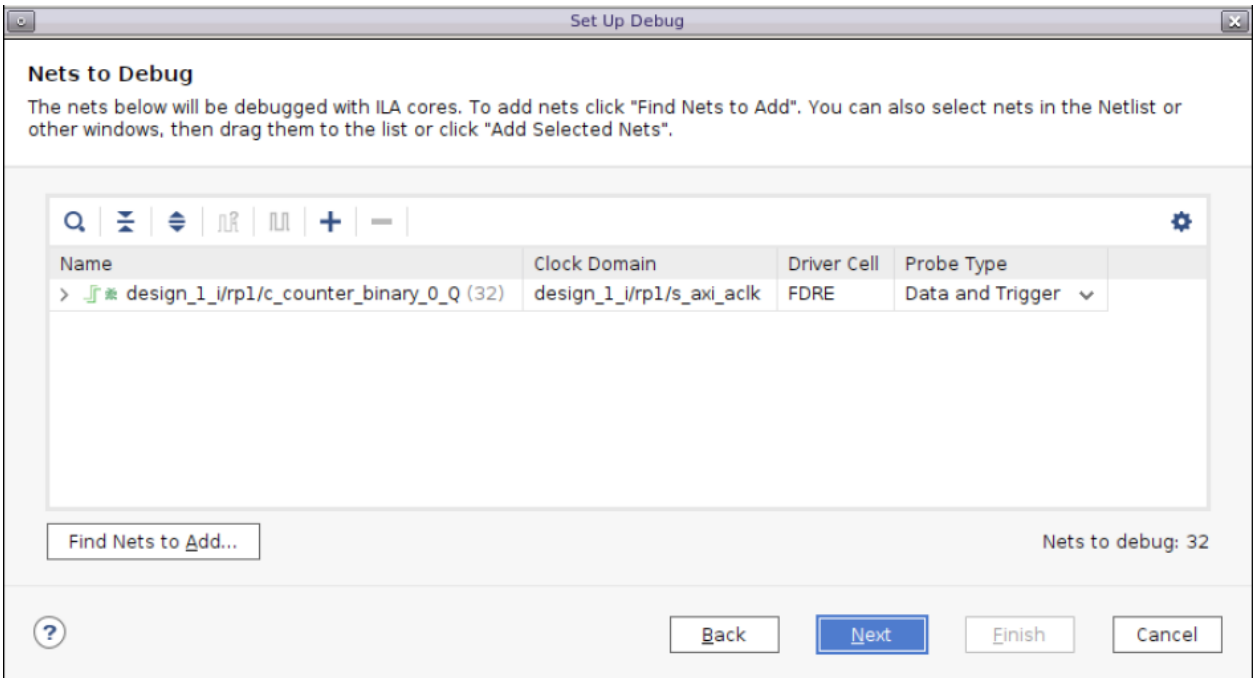
1. In the Flow Navigator, select **Set Up Debug**.

Figure 174: **Set Up Debug Command**



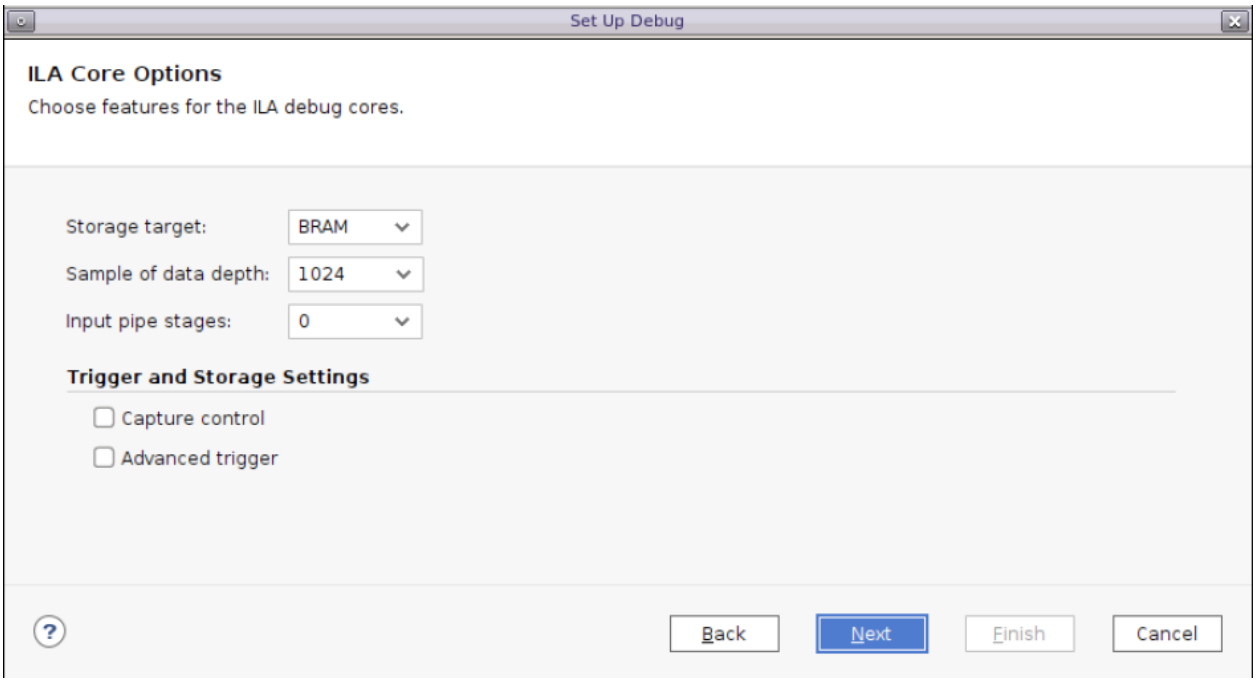
2. In the Set Up Debug dialog box, click **Next**.
3. In the Nets to Debug screen, adjust settings as needed, add more nets if desired, and click **Next**. This screen shows a list of signals that are tagged with the Debug (BD) or MARK_DEBUG (RTL) property.

Figure 175: **Set Up Debug Dialog Box Nets to Debug**



4. In the ILA Core Options screen, adjust the ILA debug core settings as needed, and click **Next** and then **Finish** to complete the debug setup.

Figure 176: **Set Up Debug Dialog Box ILA Core Options**

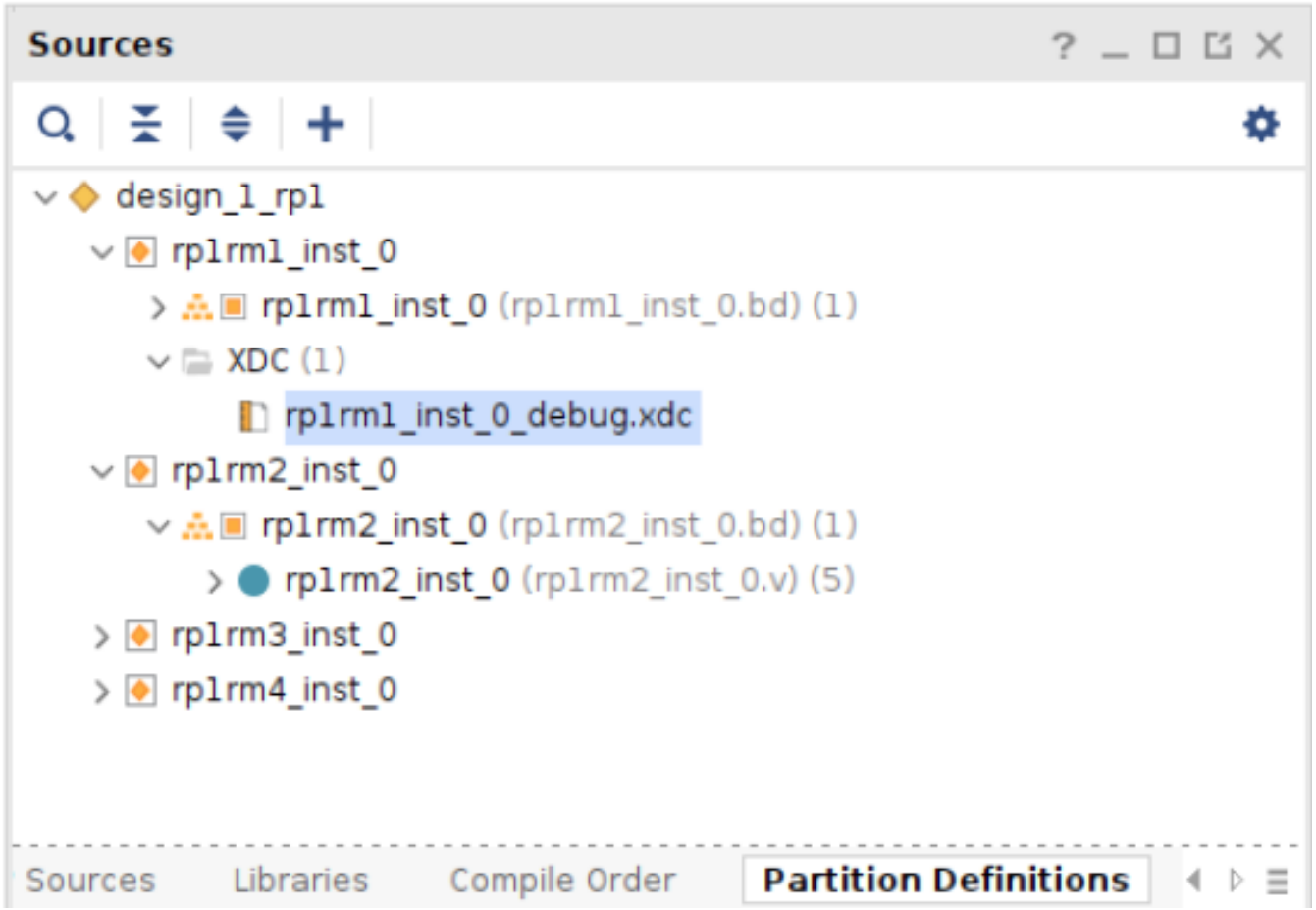


After you click Finish, the tools insert the ILA core. The core is declared, properties of the core are defined, and the core is connected to the existing design logic. This information is stored in the post-synthesis checkpoint for this configuration and in an RM-specific constraint file that is created and placed within the source set of the target RM in the Partition Definition. The generation of the ILA core itself (and the debug hub core) is actually done during the `opt_design` phase of implementation. In the Netlist view, these cores are still black boxes, as shown in the following figure.

Figure 177: **Netlist View**

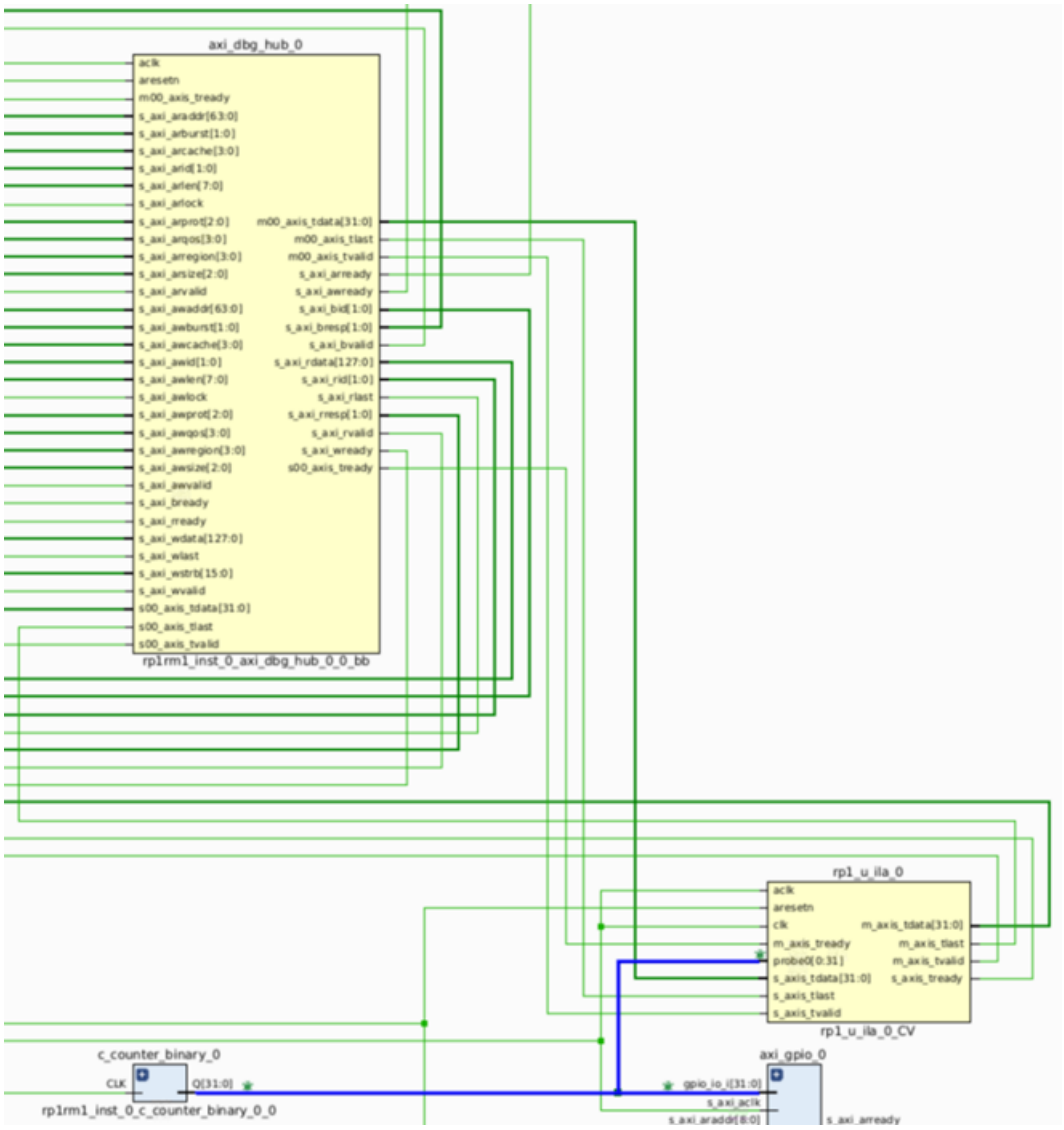


Figure 178: **Partition Definition View**



In the Schematic view, these cores are represented by yellow levels of hierarchy, as shown in the following figure.

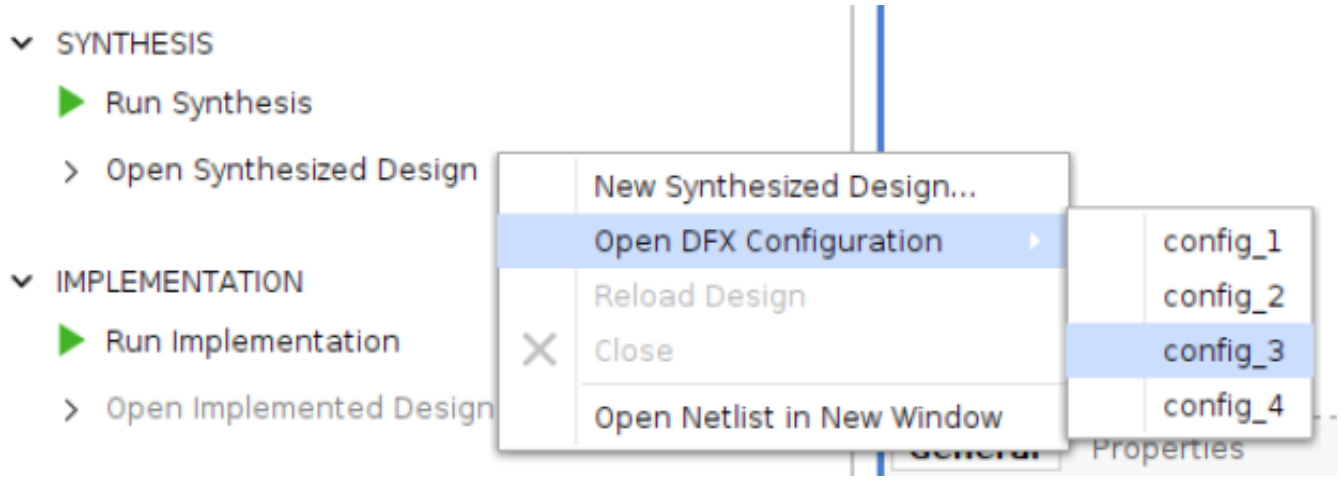
Figure 179: **Schematic View**



Save the design checkpoint before continuing.

This sequence of steps for the Set Up Debug command must be repeated for other configurations if reconfigurable modules not included in the parent configuration are to be debugged. For every other reconfigurable module that is expected to have ILA cores inserted, a configuration containing that RM must be opened to declare the ILA core details, if you are not using default settings. The Open Synthesized Design selection in the Flow Navigator can open any run configuration. Right-click on **Open Synthesized Design** and select **Open DFX Configuration**, followed by the desired configuration.

Figure 180: **Open DFX Configuration**



You can also perform this action of opening a design configuration from the Tcl Console. Open a parent or child run configuration by calling `open_run` directly as follows:

```
open_run synth_1 -name synth_1 -pr_config <configuration>
```

After opening the child run configuration, the process is the same. Call Set Up Debug to define the details of the ILA core and its connections, and save the modified configuration checkpoint.

Compile the parent and child runs as you would for any DFX design. Debug cores are generated during the `opt_design` step. You can confirm successful insertion by opening the routed checkpoints and examining the design hierarchy. Reconfigurable modules do not need to have ILA core insertion performed even if other RMs for the same RP have inserted ILA. The parent configuration is required to have debug core insertion to establish the NoC connectivity to the static design, and it is recommended that debug hubs are still added to all RMs to maintain a consistent NoC topology. Greybox configurations are supported as child configurations with the insertion flow.

Interacting with the debug cores for a DFX design, regardless of design flow, matches a standard debug solution. For more information on debug capabilities in Versal devices, see the *Vivado Design Suite User Guide: Programming and Debugging* (UG908). For an example design showing both flow methodologies, see the Versal Device DFX Debug Tutorials available from the AMD GitHub repository.

Recommended Floorplanning Constraints

The recommendations in this section apply to Versal devices. For AMD UltraScale™ and AMD UltraScale+™ devices, see Design Considerations and Guidelines for UltraScale and UltraScale+ Devices. For Versal devices, see Reduce the Number of Partition Pins and Considerations for Static Pblocks with CONTAIN_ROUTING Enabled.

Related Information

Design Considerations and Guidelines for UltraScale and UltraScale+ Devices
Reduce the Number of Partition Pins
Considerations for Static Pblocks with CONTAIN_ROUTING Enabled

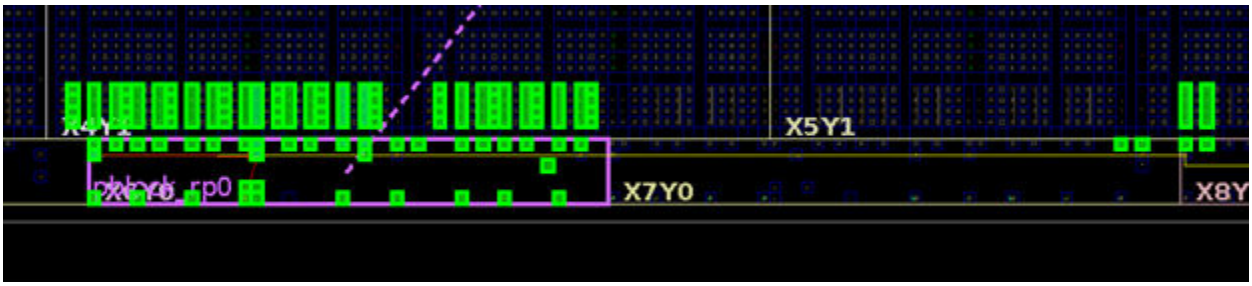
Range Only the Required Physical Sites within HSR Pblock Rectangles for Designs with Two RPs

DFX designs for Versal devices have unique challenges related to ranging Pblocks in designs with two reconfigurable partitions (RPs). Due to the alignment of BLI tiles that are automatically ranged based on other ranged sites within an HSR clock region, the placement and routing footprints can extend farther than originally intended for a given Pblock rectangle. The extended footprints can lead to overlapping Pblock DRCs.

Therefore, it is recommended that you only range the sites required for HSR Pblock rectangles. The following figures illustrate the differences. Originally, HSR resources are added to a Pblock using the entire clock region designation. The green highlighted sites outside of the `pblock_rp0` Pblock are automatically pulled into the placement footprint. You can view this using the `get_dfx_footprint -place -of_objects <rm cell inst name>` command. For example:

```
resize_pblock [get_pblocks pblock_rp0] -add
{CLOCKREGION_X6Y0:CLOCKREGION_X6Y0}
```

Figure 181: **Ranging all HSR Sites via Clock Region Constraints**

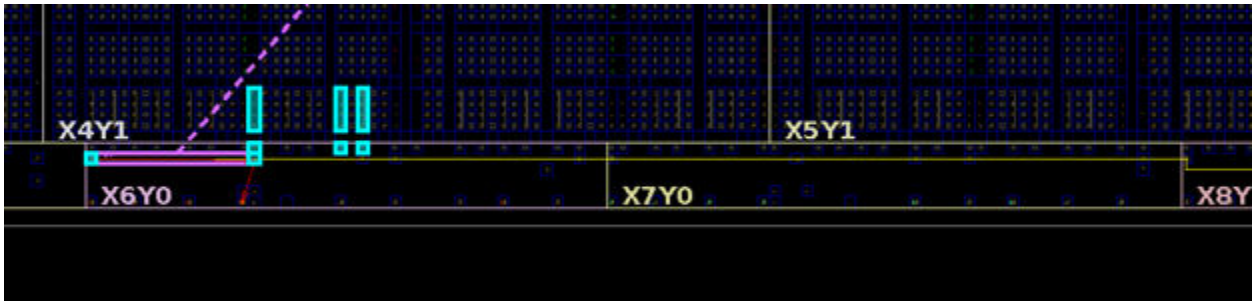


In this example, an MMCM is present that only requires the MMCM and connected buffer sites. The I/Os within the design are constrained to static banks, so sites within the HSR clock region, such as IOB, XPHY, and XPIOLOGIC, are not needed. This Pblock can be adjusted to remove the full clock region range and only range necessary sites. For example:

```
resize_pblock [get_pblocks pblock_rp0] -remove
{CLOCKREGION_X6Y0:CLOCKREGION_X6Y0}
resize_pblock [get_pblocks pblock_rp0] -add {BUFGCE_X6Y0:BUFGCE_X6Y23}
resize_pblock [get_pblocks pblock_rp0] -add {BUFGCTRL_X6Y0:BUFGCTRL_X6Y7}
resize_pblock [get_pblocks pblock_rp0] -add {MMCM_X6Y0:MMCM_X6Y0}
```

The adjusted Pblock range has a much smaller placement footprint, which is highlighted in blue in the following figure. This reduced footprint can help to reduce the chances of overlapping with a second RP Pblock.

Figure 182: **Ranging Only Required HSR Sites via Targeted Constraints**



Avoid Disjoint Pblocks Whenever Possible for Versal Devices

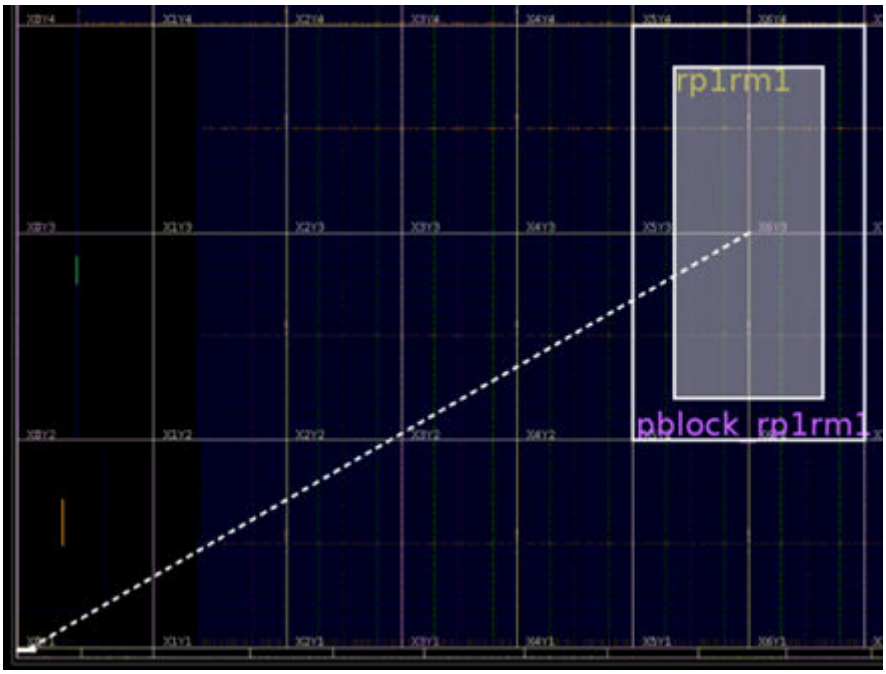
AMD highly recommends contiguous floorplanning for static regions and reconfigurable Pblocks for the following reasons:

- Disjoint Pblocks for either a reconfigurable partition or the static region can result in related logic being placed in separate areas, which has a negative impact on routability and timing closure. Without additional guidance, it is possible for the placer to create a situation that is unroutable if related logic is not kept together.
- Although static nets can cross through reconfigurable Pblocks to allow communication between two static islands, it is best to limit this approach because locked static nets can block routing during subsequent reconfigurable module implementations. Building contiguous regions and minimizing overlap is the best layout strategy for DFX designs.

The Vivado tools determine that two portions of a Pblock are disjoint if the corresponding expanded routing footprint for a collection of resources has a gap between it and another routing footprint section for the same Pblock. The reconfigurable Pblock does not own general routing resources within this gap. This means that finding a solution that connects these disjoint sections is either difficult or impossible, depending on how the Pblock is created. Use the `get_dfx_footprint -route -of_objects <rm cell inst name>` command to see the routing footprint of a reconfigurable partition Pblock.

There are some cases in which contiguous Pblocks cannot be used, and disjoint Pblocks must be used instead. The most common case is to accommodate clocking resources in Versal devices due to the layout of the silicon. Clocking resources such as BUFGCE, MMCM, DPLL and XPLL are located along the bottom (or sides) of the device and therefore are physically separated from a dynamic region that must be placed on a different area of the die. If a reconfigurable partition must contain clocking resources and be situated in the device in a location away from the bottom (or sides) of the device, the Pblock for the reconfigurable partition must have two or more disjoint sections. The only routing permitted between the disjoint sections is clock routing, because these resources are not part of the general routing domain. The following figure shows an example.

Figure 183: **Disjoint Pblock Example**



Following is the constraint syntax for the Pblock in this example:

```
create_pblock pblock_rp1rm1
add_cells_to_pblock [get_pblocks pblock_rp1rm1] [get_cells -quiet [list
design_1_i/rp1rm1]]
resize_pblock [get_pblocks pblock_rp1rm1] -add
{CLOCKREGION_X5Y2:CLOCKREGION_X6Y3}
resize_pblock [get_pblocks pblock_rp1rm1] -add {BUFGCE_X0Y0:BUFGCE_X0Y23}
resize_pblock [get_pblocks pblock_rp1rm1] -add {MMCM_X0Y0:MMCM_X0Y0}
```

The BUFGCE and MMCM instances selected are located in the lower-left corner of the device. Only the reconfigurable partitions with areas that occupy the Y1 row of the device can have a contiguous region that includes these clocking resources.

***Note*:** Other resources, such as IOB, XPIO, and XPHY, are also located in the bottom row of the device but should not be considered in a remote section of a Pblock. Without a contiguous region, the reconfigurable module logic cannot be routed between the disjoint sections.

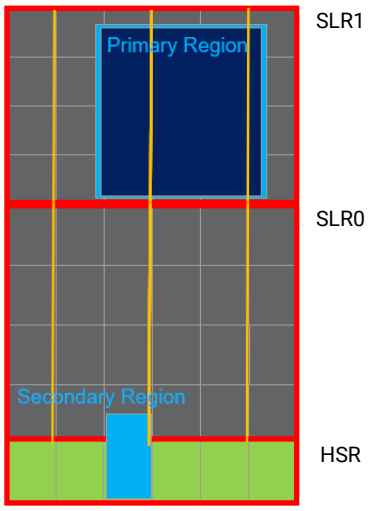
If design rule checks indicate that resources such as slices near the clocking site are also required, add these to your primary Pblock. If the required sites are currently occupied by a different reconfigurable partition Pblock, the listed sites must also be removed from that Pblock before adding them to the target Pblock.

Create Disjoint Pblocks for a Reconfigurable Partition

In a disjoint Pblock, the part of the reconfigurable Pblock in the fabric region that is away from the clocking resources at the bottom of the device is referred to as the *primary region*. The part of the reconfigurable Pblock in the fabric region that is adjacent to the clock sources, which includes the sites of the clock sources, is referred to as the *secondary region*. The following figure shows an example.

***Note*:** Currently, the Vivado tools support one primary region and one secondary region in a disjoint Pblock. If this rule is violated, DRC [HDPR-139] is flagged.

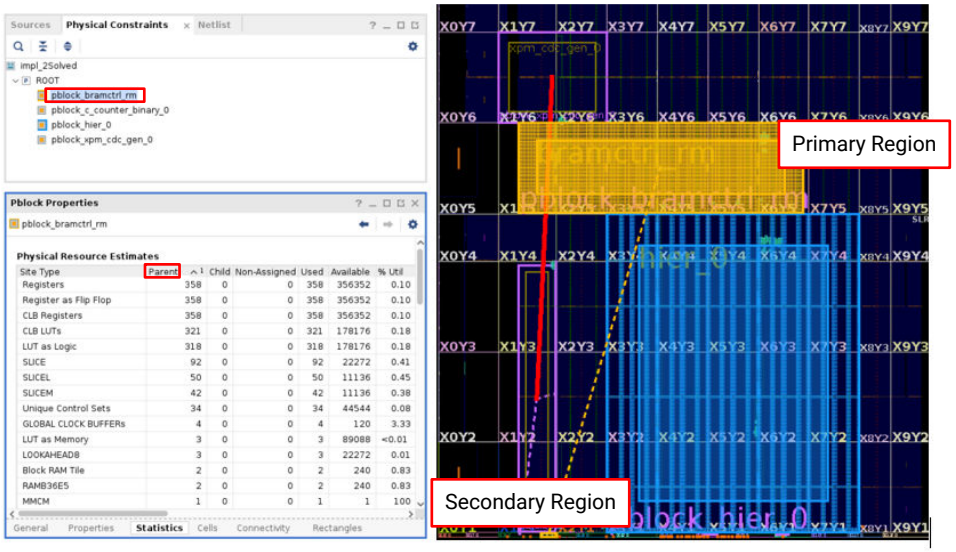
Figure 184: **Primary and Secondary Regions in a Disjoint Pblock**



To create the disjoint Pblock, use the following steps:

1. Create the top Pblock that defines the reconfigurable partition. Add the EXCLUDE_PLACEMENT and CONTAIN_ROUTING properties.

   The following figure shows the statistics for the top disjoint pblock_bramctrl_rm in the Device view with the Pblock highlighted in orange. The primary region of the Pblock is in SLR1. The secondary region of the Pblock is in the horizontal super row (HSR).



2. Create a child Pblock that defines the primary region. The child Pblock must match the ranges of the main contiguous reconfigurable Pblock. Omit the ranges for the clocking resources. The clocking resources must be part of the secondary region. You can use the `get_dfx_footprint` command to get the child Pblock site range. For example:

   ```
   get_dfx_footprint -site_type fsr -of_objects [get_cells design_0_i/
   bramctrl_rm]
   ```

   Follow these guidelines when creating the child Pblock:

   - Use the PARENT property to define the relationship between this new child Pblock and the top RP Pblock. For example:

     ```
     set_property PARENT <parent_Pblock> [get_Pblocks <child_Pblock>]
     ```

   - Do *not* use the `EXCLUDE_PLACEMENT` and `CONTAIN_ROUTING` properties for the child Pblock.
   - Only one child Pblock is supported.
   - Disable `IS_SOFT` property on child Pblock.

   Note: The Vivado tools automatically apply `SNAPPING_MODE = NESTED` to the child Pblock if the parent Pblock has `SNAPPING_MODE = ON`.

3. Optionally, add multiple parallel or nested Pblocks under the child Pblock to do more detailed floorplanning. The default `IS_SOFT` property for these nested Pblock under Child Pblocks is enabled.

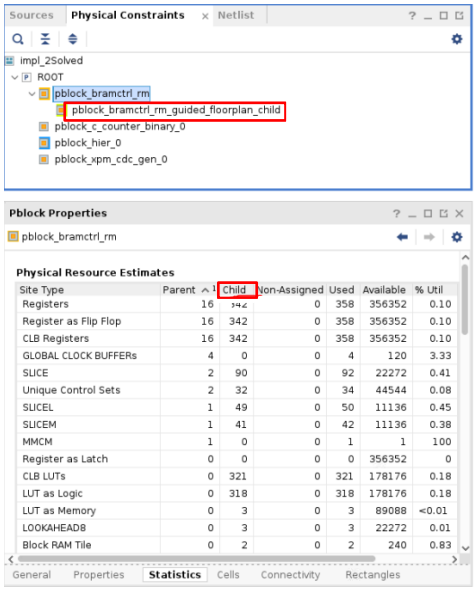
You must make specific cell assignments in the primary and secondary regions. This ensures that the Vivado placer does not place non-clock logic in the secondary region of a disjoint Pblock. If this non-clock logic connects to logic in the primary region of the disjoint Pblock, unroutes might occur because there are no standard, non-clock routing resources to bridge the gaps between the sections. Follow these guidelines when making cell assignments:

- Assign all the non-clock cells to the child Pblock or the primary region. Exclude assignment of the clock control logic from the child Pblock.
- Extend the secondary region into the fabric region to place the clock control logic (for example, Safe Clock StartUp) near the clock sources placed in the HSR.
- Use the `get_dfx_footprint` command to get the non-clock cells that need to be assigned to the child Pblock. For example:

  ```
  get_dfx_footprint -cell_type non_clock -of_objects [get_cells design_0_i/
  bramctrl_rm]
  ```

The following figure shows the Pblock statistics for the pblock_bramctrl_rm after creating the pblock_bramctrl_rm_guided_floorplan_child Pblock. All the resources are now assigned to the child Pblock except the clock resources like GLOBAL CLOCK BUFFERS, MMCM, and the 16 registers of the Safe Clock StartUp logic, which remain assigned to the parent Pblock.

Figure 185: **Pblock Statistics**

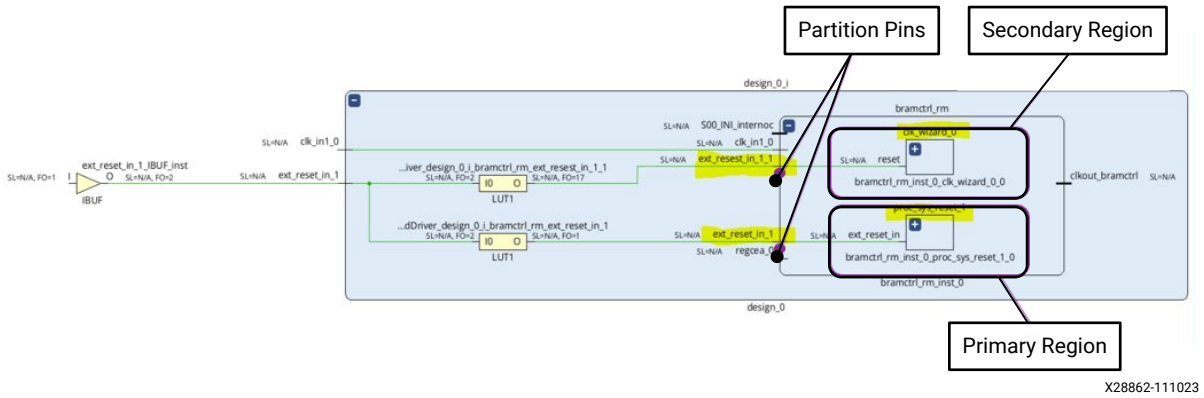


X28861-111023

In some cases, a non-clock boundary net must drive logic in both the primary region and secondary region. When this occurs, the Vivado tools need two separate PPLOCs for proper routing of the boundary net to each region. If the design has a single boundary pin for this boundary net, you must split the boundary pin into two. One pin must be used by the boundary net to drive the loads of the logic that is assigned to the primary region, and the other pin must be used to drive the loads in the secondary region. AMD recommends planning the hierarchy of the reconfigurable module so that the clock and clock control logic are in one module, which makes splitting this boundary pin straightforward. For example designs that show disjoint Pblock creation methodologies, see the Versal Adaptive SoC DFX Tutorials available from the GitHub repository.

The following figure shows an example of the `ext_reset_in_1` boundary net, which needs to drive the logic in the primary region and secondary region. The `ext_reset_in_1` boundary pin of the RM `bramctrl_rm` is split into `ext_reset_in_1_1` and `ext_reset_in_1` so that two PPLOCs are created, one for each disjoint Pblock region.

Figure 186: **Boundary Net Example**



If you do not follow the preceding recommended methodologies, the violations are reported by the following disjoint Pblock DRCs.

Table 15: **Disjoint Pblock DRC Violations**

DRC	Message	Violation
HDPR-139	The disjoint Pblock '%ELG' for Reconfigurable Module '%ELG' has more than one %STR logic regions which have no overlapping routing footprints. A disjoint Pblock can have only one HSR and one fabric logic region. Use get_dfx_footprint command to debug further and modify the Pblock.	The disjoint Pblock has more than one primary or secondary region that are disjoint.
HDPR-138	The net '%ELG' is connected to both HSR and fabric regions through one interface pin '%ELG' of Recongigurable Module '%ELG' of disjoint Pblock '%ELG'. This is an unrouteable situation. Please edit the design to have separate interface pins for HSR and fabric region of the disjoint Pblock.	The boundary pin of a boundary net that drives both regions of a disjoint Pblock is not split.
HDPR-137	The connection of interface pin '%ELG' of the reconfigurable module '%ELG' has changed from %STR region of the disjoint Pblock '%ELG' in parent implementation to %STR region. This is an unrouteable situation. Please change the connectivity of interface pin and ensure that all interface pins are connected to the same disjoint region as in the parent implementation.	There is a change in pin connections between the primary and secondary region in the child implementation when compared with parent implementation.

Table 15: **Disjoint Pblock DRC Violations** *(cont'd)*

DRC	Message	Violation
HDPR - 135	Reconfigurable Partition Pblock %ELG has has a child Pblock %STR with EXCLUDE_PLACEMENT set to 1. When disjoint Pblocks are used for DFX, only a single child Pblock is permitted, to define the primary region without EXCLUDE_PLACEMENT. Please reset the the property using : 'set_property EXCLUDE_PLACEMENT false [get_Pblocks %STR]'	The child Pblock includes the EXCLUDE_PLACEMENT property.
HDPR -130	Placement of logic within Reconfigurable Partition Pblock %ELG will require non-clock route connections between disjoint regions. This is an unroutable situation. Please adjust Pblock construction or location constraints to ensure all non-clock routing for the logic in this Reconfigurable Partition can be contained within the contiguous child Pblock %ELG in this disjoint Pblock case %STR	A non-clock connection exists between the primary region and secondary region.
HDPR -129	Logical elements that must be contained in Reconfigurable Partition Pblock %ELG are not included in child Pblock %ELG. When disjoint Pblocks are used for DFX, the only elements permitted outside the child Pblocks are clocking resources and elements that have explicit LOC constraints. Please move these elements to child Pblock %ELG or provide explicit LOC constraints:%STR	Some non-clock elements are not assigned to the child Pblock.
HDPR - 134	Reconfigurable Partition Pblock %ELG has no child Pblock. When disjoint Pblocks are used for DFX, a single child Pblock is required, to define the primary region. Please add a single child Pblock under %ELG to define the contiguous area.	The child Pblock is missing.
HDPR -128	Child Pblock %ELG of Reconfigurable Partition Pblock %ELG occupies a non-contiguous area. When disjoint Pblocks are used for DFX, the child Pblock must occupy a contiguous area. It is strongly recommended that this area aligns with the primary area of the parent Pblock.	The child Pblock is disjoint.

Table 15: **Disjoint Pblock DRC Violations** *(cont'd)*

DRC	Message	Violation
HDPR - 127	Reconfigurable Partition Pblock %ELG has multiple child Pblocks %STR. When disjoint Pblocks are used for DFX, only a single child Pblock is permitted, to define the primary region. Please remove all but one of these child Pblocks under %ELG to define the contiguous area.	There is more than one child Pblock with the same top reconfigurable partition Pblock as the parent Pblock.

Floorplan Guidelines for Static Logic

When static logic in the design has disjointed placement and needs to cross over the RP, you should provide guidance for static placement by using static Pblocks to avoid routing problems. Static Pblocks can be created to prevent static logic from interacting with the RM in the same SLR, resulting in better partitioning. This reduces SLR crossings and can improve the QoR. Moreover, if the static Pblock is defined within the expanded routing footprint of the RP Pblock, it can help in reducing PPLOC.

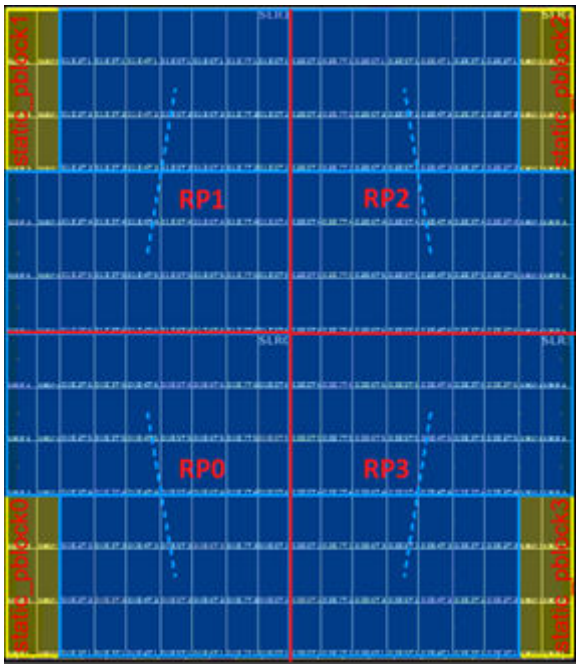
It is recommended to keep these static Pblocks hard (`IS_SOFT = FALSE`). Otherwise, the tool might ignore the Pblock constraint, which would negatively impact routability and timing closure. It is always recommended to use rectangular Pblocks. Other shapes, such as L or T shapes, are not recommended because they have more corners and could result in congestion and unroutability.

The following are the scenarios where static Pblocks are recommended.

Example 1: 4-SLR VP1902 Device with Four RPs

In this example, four static Pblocks (highlighted in yellow) are created to guide the static placement. These static Pblocks cover certain parts of the programmable logic region, along with the processor subsystem (PS), in all four SLRs. These static Pblocks contain static logic that interacts with the RM within the same SLR. This setup can help avoid ineffecient static logic placement and reduce SLR crossings.

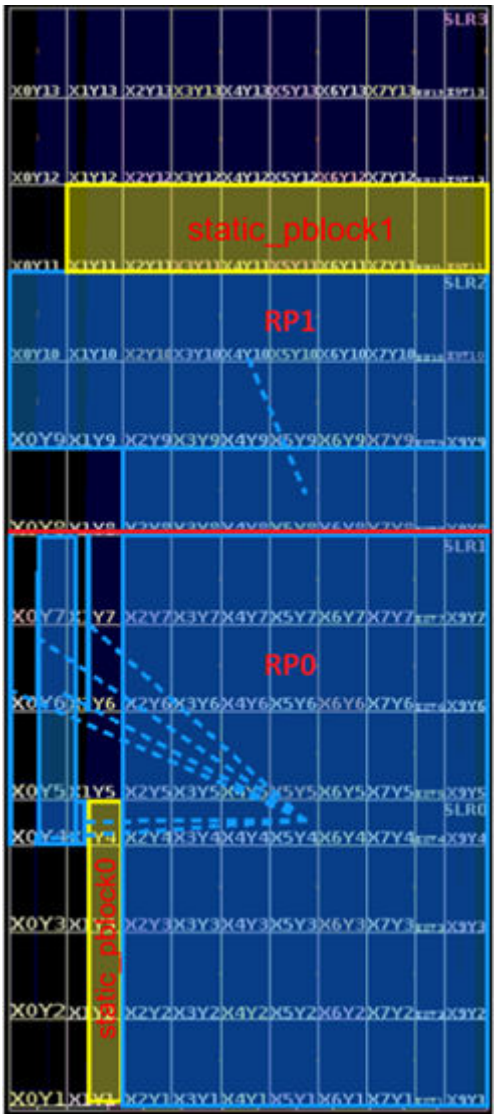
Figure 187: **4-SLR VP1902 Device with Four RPs and Four Static Pblocks**



Example 2: 4-SLR VP1802 Device with Two RPs

In this example, two static Pblocks (highlighted in yellow) are created to provide guidance for static placement. These static Pblocks are positioned near the RP, which helps keep the static logic interacting with the RM closer. Without static Pblocks, the static logic might be placed far apart, negatively impacting QoR and routability. Here, the static logic interacting with RP0 is assigned to `static_pblock0`, and the static logic interacting with RP1 is assigned to `static_pblock1`.

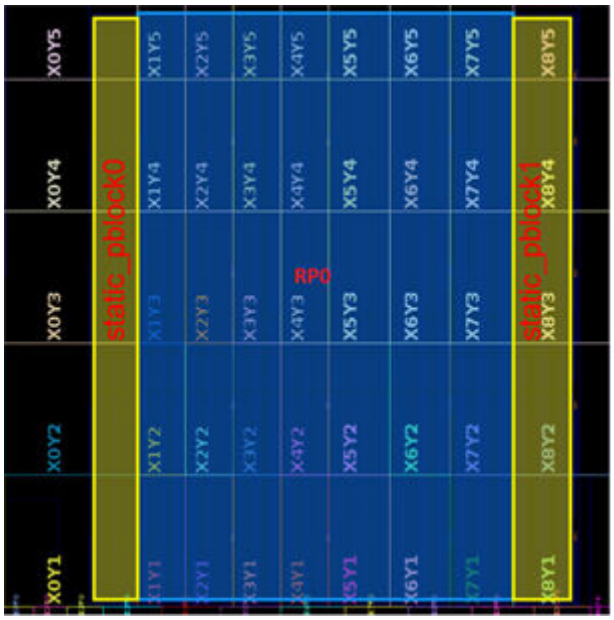
Figure 188: **4-SLR VP1802 Device with Two RPs and Two Static Pblocks**



Example 3: 1-SLR VN3716 device with One RP

In this example, two static Pblocks (highlighted in yellow) are created to provide guidance for static placement. The static Pblocks can be created on the left and right sides of the RP for static placement when there is static logic connected to hard blocks such as CPM or gigabit transceivers. While creating these static Pblocks, ensure that the interacting logic is kept together within the static Pblock so that it is not spread between the left and right static Pblocks.

Figure 189: **1-SLR VN3716 Device with One RP and Two Static Pblocks**



Using Report DFX Summary

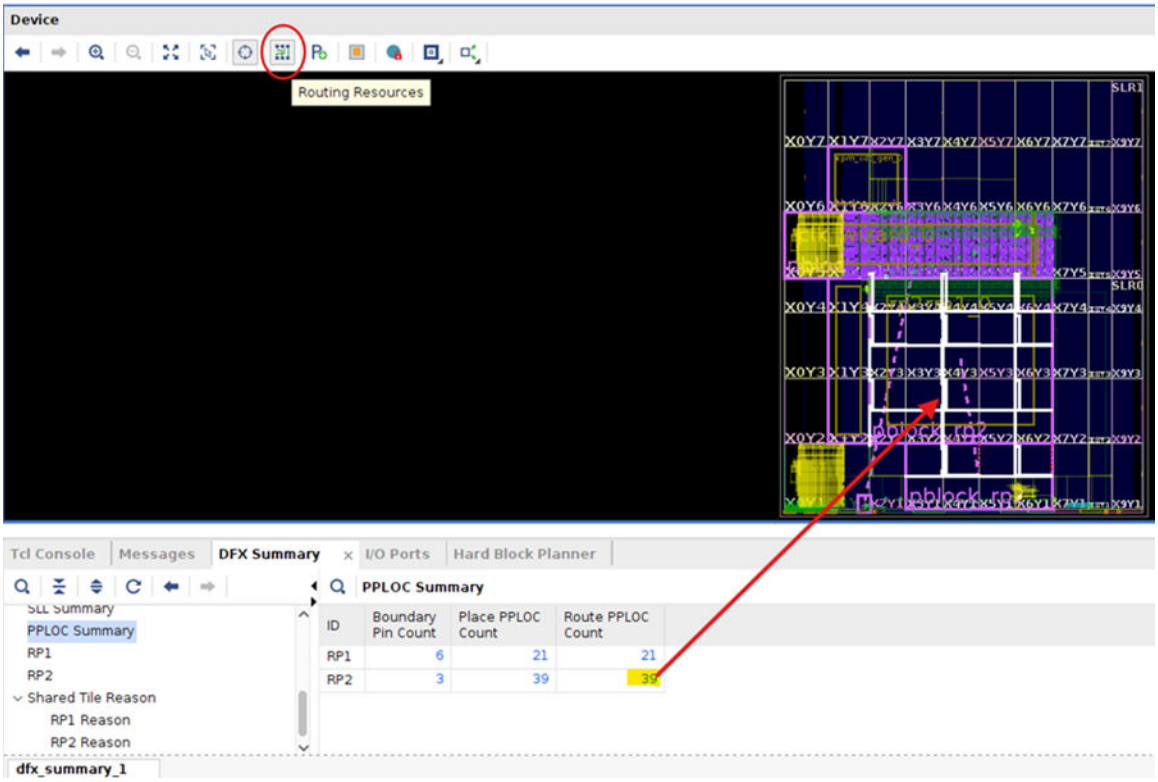
Use the `report_dfx_summary` command to analyze DFX designs. The DFX Summary Report provides information on the following topics:

- Design Configuration
- Details on Reconfigurable Pblocks
- Static and RP device utilization
- Clock utilization of static and RP
- Details on the number of SLL used and available for static and RP routing
- Total number of PPLOC deposited during implementation and boundary pin count of each RP

In addition, the DFX Summary Report in the Vivado IDE supports netlist and device objects selection for highlighting placement information and creating schematics.

The following figure shows selected post route PPLOC nodes of RP2 in device view by clicking on the hyperlink in the IDE report.

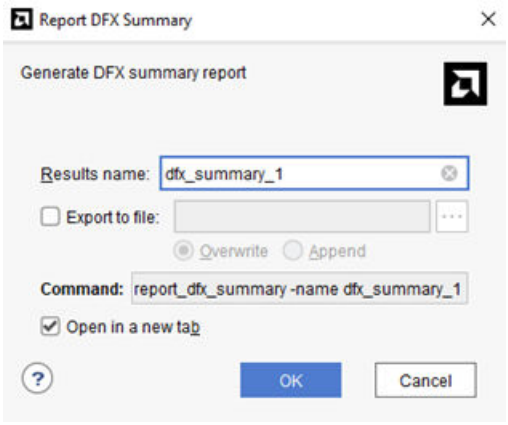
Figure 190: **Example of PPLOC Nodes Selected From the Hyperlink**



TIP: *Turn on **Routing Resources** in the device view window to view the routing resources.*

Generating the DFX Summary Report

1. To generate the DFX Summary Report in the Vivado IDE, select **Reports → Report DFX summary**.



The equivalent Tcl command is:

```
report_dfx_summary -name dfx_summary_1
```

2. In the Results name field specify the name of the graphical window for the report. The equivalent Tcl command is:

   ```
   -name <windowName>
   ```

3. Select **Export to file** to write the results to a file in addition to generating a GUI report. Specify a file name in the field on the right. Click the elipses button to browse to select a different directory. The equivalent Tcl command is:

   ```
   -file <arg>
   ```

4. Select **Overwrite** to overwrite an existing file with new analysis results. Select **Append** to append the new results.

Usage

The DFX Summary Report is created by calling the `report_dfx_summary` command in the Tcl Console with a DFX design open in memory. Reconfigurable Partition Pblocks are required for accurate resource utilization information. The command can be issued after any stage of implementation from post-synthesis to post-route, with the accuracy of the report improving the further along the design is within this process. Use the `-file` option to send the report to a text file. The `-force` option overwrites an existing report, and the `-append` option adds to an existing report. With no options selected, the report is issued in the Tcl Console.

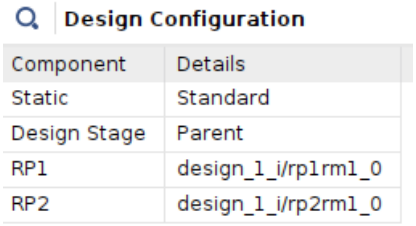
The DFX Summary Report includes the following sections:

- Design Configuration
- Pblock by Hierarchy
- Design Utilization Summary
- Design Clock Utilization Summary
- SLL Summary
- PPLOC Summary
- RP Details
- Shared Tile Reason (For multi RP design)

Design Configuration

The Design Configuration section is divided into two tables. The first table has information about the design stage, including whether the stage is a parent or child run. The static information is generated using the standard DFX flow or the Abstract Shell flow. Each RP is assigned a unique Partition ID (RP1, RP2, etc.), which is used consistently across all report tables. The second table shows details on each RP.

Figure 191: **Design Configuration Table**



Design Configuration

Component	Details
Static	Standard
Design Stage	Parent
RP1	design_1_i/rp1rm1_0
RP2	design_1_i/rp2rm1_0

Figure 192: **RP Summary**



RP Summary

RP ID	Cell Name	Pblock Name	Disjoint Pblock	Child Pblock	Sub-Child Pblock	Boundary Pins	Boundary Clocks	Placement Footprints	Routing Footprints
RP1	design_1_i/rp1rm1_0	pblock_rp1	Yes	1	0	6	1	29802	41004
RP2	design_1_i/rp2rm1_0	pblock_rp2	No	0	0	3	1	84140	102398

Table 16: **Design Configuration Table**

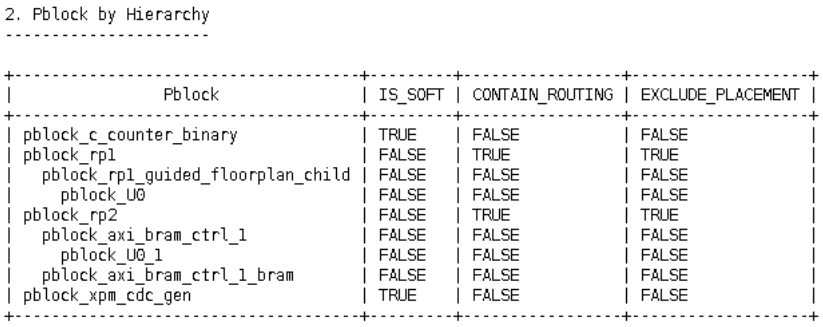
Column	Description
Reconfiguration Partition ID	Unique ID assigned to RP
Cell Name	RM cell name
Pblock Name	Name of RP Pblock
Disjoint Pblock	Is the RP Pblock disjoint? Yes / No
Child Pblock	Number of child Pblocks under RP Pblock. Parent of child Pblock is RP Pblock.
Sub-Child Pblock	Number of sub-child Pblock. Parent of sub-child Pblock is child Pblock
Boundary Pins	Number of boundary pins of the RM
Boundary Clocks	Number of boundary clocks of the RM
Placement Footprint	Number of tiles included in placement footprint of RM
Routing Footprint	Number of tiles included in routing footprint of RM

Pblock by Hierarchy

The Pblock by Hierarchy table presents the hierarchical structure of both static and reconfigurable partition (RP) Pblocks in a tree-like format. This hierarchical view helps you visualize the parent-child relationships among Pblocks. Each level in the hierarchy is indented to reflect its depth and relationship to other Pblocks. For each Pblock level, you can see the properties `IS_SOFT`, `CONTAIN_ROUTING`, and `EXCLUDE_PLACEMENT`.

Note: Vivado reports the Pblock by Hierarchy table only in the text file output. The table does not appear when you run `report_dfx_summary` in GUI mode.

Figure 193: **Pblock by Hierarchy Table**



2. Pblock by Hierarchy

Pblock	IS_SOFT	CONTAIN_ROUTING	EXCLUDE_PLACEMENT
pblock_c_counter_binary	TRUE	FALSE	FALSE
pblock_rp1	FALSE	TRUE	TRUE
pblock_rp1_guided_floorplan_child	FALSE	FALSE	FALSE
pblock_U0	FALSE	FALSE	FALSE
pblock_rp2	FALSE	TRUE	TRUE
pblock_axi_bram_ctrl_1	FALSE	FALSE	FALSE
pblock_U0_1	FALSE	FALSE	FALSE
pblock_axi_bram_ctrl_1_bram	FALSE	FALSE	FALSE
pblock_xpm_cdc_gen	TRUE	FALSE	FALSE

Design Utilization Summary

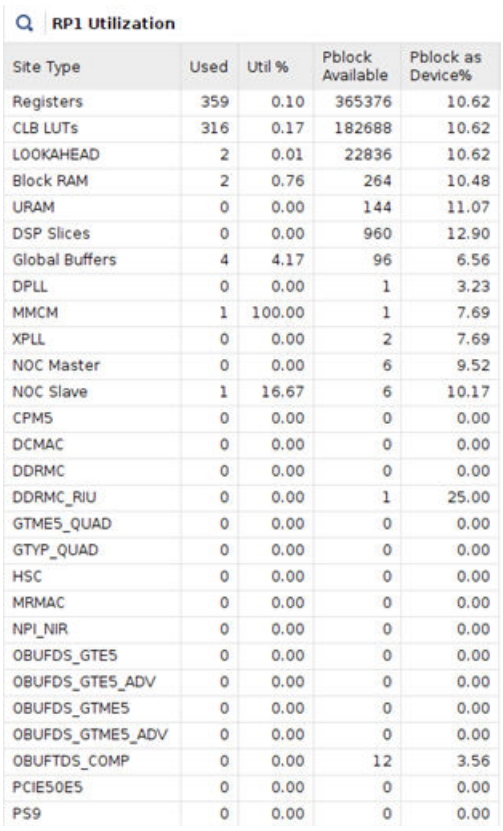
The Design Utilization Summary table provides the utilization for each RP and the static domain. The data in this table is based on resource information determined by the design floorplan, so this section of the Report DFX Summary output is only valid when a Pblock has been supplied for each reconfigurable partition (RP). The number of LUT1 buffers inserted in the static region is provided under the static utilization table. The number of LUT1 buffers inserted in the RP is provided in the RP details table.

***Note*:** For more information about design utilization, use the `report_utilization` Tcl command.

Table 17: **RP Utilization Table**

Column	Description
Used	Number of resources used in the Pblock
Util %	Percentage of resources used out of the available resources in the Pblock
Pblock Available	Total resources available in the area defined by the Pblock
Pblock Avail % in Device	Portion of the total device resource included in the RP Pblock

Figure 194: **RP Utilization Table**



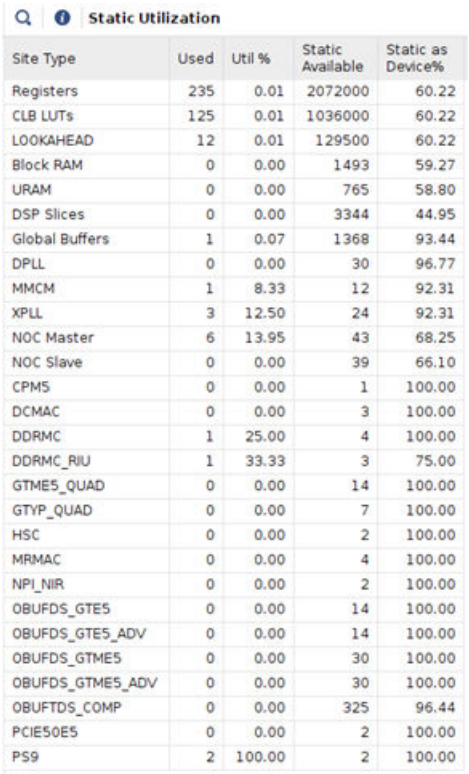
RP1 Utilization

Site Type	Used	Util %	Pblock Available	Pblock as Device%
Registers	359	0.10	365376	10.62
CLB LUTs	316	0.17	182688	10.62
LOOKAHEAD	2	0.01	22836	10.62
Block RAM	2	0.76	264	10.48
URAM	0	0.00	144	11.07
DSP Slices	0	0.00	960	12.90
Global Buffers	4	4.17	96	6.56
DPLL	0	0.00	1	3.23
MMCM	1	100.00	1	7.69
XPLL	0	0.00	2	7.69
NOC Master	0	0.00	6	9.52
NOC Slave	1	16.67	6	10.17
CPM5	0	0.00	0	0.00
DCMAC	0	0.00	0	0.00
DDRMC	0	0.00	0	0.00
DDRMC_RIU	0	0.00	1	25.00
GTME5_QUAD	0	0.00	0	0.00
GTYP_QUAD	0	0.00	0	0.00
HSC	0	0.00	0	0.00
MRMAC	0	0.00	0	0.00
NPI_NIR	0	0.00	0	0.00
OBUFDS_GTE5	0	0.00	0	0.00
OBUFDS_GTE5_ADV	0	0.00	0	0.00
OBUFDS_GTME5	0	0.00	0	0.00
OBUFDS_GTME5_ADV	0	0.00	0	0.00
OBUFTDS_COMP	0	0.00	12	3.56
PCIE50E5	0	0.00	0	0.00
PS9	0	0.00	0	0.00

Table 18: **Static Utilization Table**

Column	Description
Used	Resources used in the static domain
Util %	Percentage of resources used in static
Static Available	Total resources available for the static domain excluding the sum of all resources of all RP Pblocks from the total available device resources
Static in Device %	Portion of total device resources available in the static domain

Figure 195: **Static Utilization Table**



Static Utilization

Site Type	Used	Util %	Static Available	Static as Device%
Registers	235	0.01	2072000	60.22
CLB LUTs	125	0.01	1036000	60.22
LOOKAHEAD	12	0.01	129500	60.22
Block RAM	0	0.00	1493	59.27
URAM	0	0.00	765	58.80
DSP Slices	0	0.00	3344	44.95
Global Buffers	1	0.07	1368	93.44
DPLL	0	0.00	30	96.77
MMCM	1	8.33	12	92.31
XPLL	3	12.50	24	92.31
NOC Master	6	13.95	43	68.25
NOC Slave	0	0.00	39	66.10
CPM5	0	0.00	1	100.00
DCMAC	0	0.00	3	100.00
DDRMC	1	25.00	4	100.00
DDRMC_RIU	1	33.33	3	75.00
GTME5_QUAD	0	0.00	14	100.00
GTYP_QUAD	0	0.00	7	100.00
HSC	0	0.00	2	100.00
MRMAC	0	0.00	4	100.00
NPI_NIR	0	0.00	2	100.00
OBUFDS_GTE5	0	0.00	14	100.00
OBUFDS_GTE5_ADV	0	0.00	14	100.00
OBUFDS_GTME5	0	0.00	30	100.00
OBUFDS_GTME5_ADV	0	0.00	30	100.00
OBUFTDS_COMP	0	0.00	325	96.44
PCIE50E5	0	0.00	2	100.00
PS9	2	100.00	2	100.00

Design Clock Utilization Summary

The Design Clock Utilization Summary shows the summary of global clocks used in the design. The table gives clock net names and related details. The table includes clock information for both static and dynamic domains of the design.

***Note*:** For more information about clock usage, use the `report_clock_utilization` Tcl command.

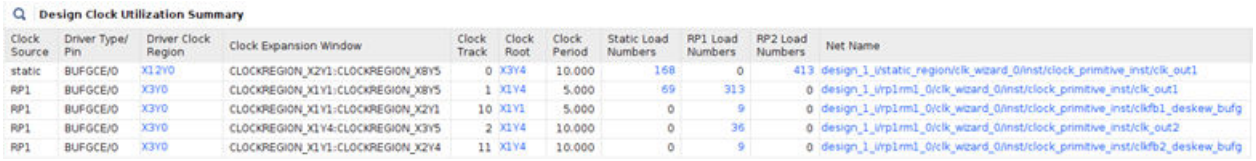
Table 19: **Design Clock Utilization Summary**

Column	Description
Clock Source	Clock source in static domain or RP
Driver Type / Pin	Output primitive pin that generates the clock
Driver Clock Region	Device clock region where the clock source is located
Clock Expansion Window	Rectangular area that includes all clock regions where clock net loads are placed
Clock Track	Clock track ID is used to route the clock net
Clock Root	Clock region where the clock net `CLOCK_ROOT` is located
Clock Period	Period in nanoseconds of timing clock that propagates on the clock net
Static Load Numbers	Number of static cells connected to clock driver pin

Table 19: **Design Clock Utilization Summary** *(cont'd)*

Column	Description
RP Load Numbers	Number of RP cells connected to clock driver pin (one column per RP)
Net Name	Logical name of clock net segment connected to the clock driver pin

Figure 196: **Design Clock Utilization Summary Table**



Design Clock Utilization Summary

Clock Source	Driver Type/ Pin	Driver Clock Region	Clock Expansion Window	Clock Track	Clock Root	Clock Period	Static Load Numbers	RP1 Load Numbers	RP2 Load Numbers	Net Name
static	BUFGCE/O	X12Y0	CLOCKREGION_X2Y1:CLOCKREGION_X8Y5	0	X3Y4	10.000	168	0	413	design_1_i/static_region/clk_wizard_0/inst/clock_primitive_inst/clk_out1
RP1	BUFGCE/O	X3Y0	CLOCKREGION_X1Y1:CLOCKREGION_X8Y5	1	X1Y4	5.000	69	313	0	design_1_i/rp1rm1_0/clk_wizard_0/inst/clock_primitive_inst/clk_out1
RP1	BUFGCE/O	X3Y0	CLOCKREGION_X1Y1:CLOCKREGION_X2Y1	10	X1Y1	5.000	0	9	0	design_1_i/rp1rm1_0/clk_wizard_0/inst/clock_primitive_inst/clkfb1_deskew_bufg
RP1	BUFGCE/O	X3Y0	CLOCKREGION_X1Y4:CLOCKREGION_X3Y5	2	X1Y4	10.000	0	36	0	design_1_i/rp1rm1_0/clk_wizard_0/inst/clock_primitive_inst/clk_out2
RP1	BUFGCE/O	X3Y0	CLOCKREGION_X1Y1:CLOCKREGION_X2Y4	11	X1Y4	10.000	0	9	0	design_1_i/rp1rm1_0/clk_wizard_0/inst/clock_primitive_inst/clkfb2_deskew_bufg

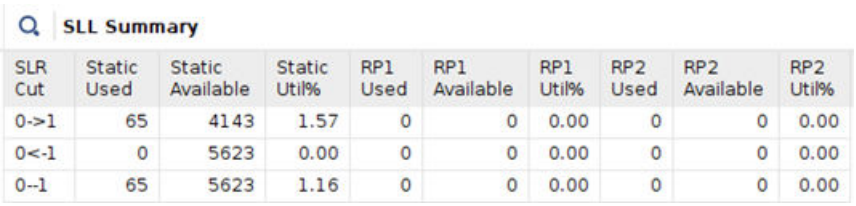
SLL Summary

The SLL summary table provided details on the number of SLLs used and available for each SLR crossing for static routing, as well as for each RP routing based on the DFX floorplan. For each SLR boundary the number of SLL used and available are detailed per direction (north and south) and combined for both directions.

Table 20: **SLL Summary Table**

Column	Description
SLR Cut	Direction of SLL nodes on each SLR
Static Used	Number of SLL nodes used by static nets
Static Available	Number of SLL nodes available for static nets
Static Util%	Utilization percentage of static nets: (Used/available)*100
RP Used	Number of SLL nodes used by RM nets
RP Available	Number of SLL nodes available for RM nets
RP Util%	Utilization percentage of RM nets: (Used/available)*100

Figure 197: **SLL Summary Table**



SLL Summary

SLR Cut	Static Used	Static Available	Static Util%	RP1 Used	RP1 Available	RP1 Util%	RP2 Used	RP2 Available	RP2 Util%
0->1	65	4143	1.57	0	0	0.00	0	0	0.00
0<-1	0	5623	0.00	0	0	0.00	0	0	0.00
0--1	65	5623	1.16	0	0	0.00	0	0	0.00

PPLOC Summary

The PPLOC Summary table shows the partition pin count of each RP and the boundary pin count of each RM associated with each RP.

There can be differences in the number of partition pins between the placer and router stages. For example, if static leaf cells are placed in the expanded routing footprint, PPLOCs are not needed. These PPLOCs can be assigned by the placer but later removed by the router.

Figure 198: **PPLOC Summary Table**

PPLOC Summary

ID	Boundary Pin Count	Place PPLOC Count	Route PPLOC Count
RP1	6	21	21
RP2	3	39	39

RP Details

The RP Details tables provide information on each RP has. The name of table is based on the Reconfiguration Partition ID listed in the Design Configuration section.

Table 21: **RP Details**

Column	Description
RM Instance name	RM cell name
Parent Pblock	Name of RP Pblock
Child Pblock	Number of child Pblocks of RP Pblock
Pblock Components	Number of Pblock elements in FSR and HSR. This is reported only for disjoint Pblocks
Sub-Child Pblock	Is Sub-Child Pblock present? Yes / No
Pblock Components	Number of Pblock elements in FSR and HSR. This is reported only for disjoint Pblocks
Prohibited Site Count	Number of sites prohibited by the implementation tools
Shared Tiles Count	Number of tiles shared with another RP. This includes the shared clock tiles for the RM
Overlap Tiles Count	Number of overlapping tiles in routing footprint of RM
Inserted LUT1 Buffers	Number of LUT1 buffers inserted by implementation tools in RM
PPLOC Count	Number of PPLOC at placer and router stages
PPLOC Change Reason	This appears if the PPLOC is removed, added, or modified

Figure 199: **RP Details Table**

RP1

Characteristics	Value
RM Instance Name	design_1_i/rp1rm1_0
Parent Pblock	pblock_rp1
Child Pblocks	1
Sub-Child Pblock	Yes
Pblock Components	HSR# 1 : FSR# 1
Prohibited Site Count	12
Unused sites of IO bank in RP pblock	12
Shared Tiles Count	30
Overlap Tiles Count	0
Inserted LUT1 Buffers	1
PPLOC Count	
Post Place Count	21
Post Route Count	22
PPLOC Change Reason	
Removed PPLOCs	0
Static Driver in Expanded Routing Footprint	0
Modified PPLOCs	3
Two Pin Nets	3
Added PPLOCs	0
Dedicated Nets	0
Clock Driver in RM	0

Shared Tile Reason

In a multi-RPs design, the shared tile reason table provides information on various categories and the corresponding number of tiles shared with other RPs.

Figure 200: **Shared Tile Reason**

RP1 Reason

Shared tile type	RP2
Interconnect tiles	0
Clocking tiles	30
Unconnected tiles	0

RP2 Reason

Shared tile type	RP1
Interconnect tiles	0
Clocking tiles	30
Unconnected tiles	0

Configuring the Device

This chapter describes the system design considerations when configuring your device with a partial BIT file, as well as architectural features in the FPGA that facilitate Dynamic Function eXchange (DFX). Because most aspects of Dynamic Function eXchange are no different than standard full configuration, this section concentrates on the details that are unique to DFX.

Configuration Modes

Dynamic Function eXchange is supported using the following configuration modes:

- **ICAP:** A good choice for user configuration solutions. Requires the creation of an ICAP controller as well as logic to drive the ICAP interface.
- **AXI32:** The ICAP equivalent for AMD Spartan™ UltraScale+™ devices only. This primitive provides a hardened 32-bit AXI4 interface from the programmable logic (PL) to the platform management controller (PMC).
- **MCAP:** (AMD UltraScale™ and AMD UltraScale+™ devices only) Provides a dedicated connection to the configuration engine from one specific PCIe® block per device.
- **PCAP:** The primary configuration mechanism for AMD Zynq™ 7000 SoC and Zynq UltraScale+ MPSoC designs.
- **JTAG:** A good interface for quick testing or debug. Can be driven with the AMD Vivado™ Logic Analyzer.
- **Slave SelectMAP or Slave Serial:** A good choice to perform full configuration and dynamic reconfiguration over the same interface.

Master modes are not directly supported because IPROG housecleaning clears the configuration memory.

Table 22: **Supported Configuration Ports**

Configuration Mode	7 Series	Zynq	UltraScale	Artix, Kintex, and Virtex UltraScale+	Spartan UltraScale+	Zynq UltraScale+ MPSoC
JTAG[2]	Yes	Yes	Yes	Yes	Yes	Yes

Table 22: **Supported Configuration Ports** *(cont'd)*

Configuration Mode	7 Series	Zynq	UltraScale	Artix, Kintex, and Virtex UltraScale+	Spartan UltraScale+	Zynq UltraScale+ MPSoC
ICAP	Yes	Yes	Yes	Yes	Yes	Yes
AXI32	N/A	N/A	N/A	N/A	Yes	N/A
PCAP	N/A	Yes	N/A	N/A	N/A	Yes
MCAP	N/A	N/A	Yes	Yes	Yes	Yes
Slave Serial	Yes	N/A	Yes	Yes	Yes	N/A
Slave SelectMap	Yes	N/A	Yes	Yes	Yes	N/A
SPI (any width)[1]	No	N/A	No	Yes	Yes	N/A
BPI sync mode	No	N/A	No	Yes	N/A	N/A
BPI async mode	Yes	N/A	Yes	Yes	N/A	N/A
Master modes	No	N/A	No	No	No	N/A

Notes:

1. SPI and BPI flash can be used to store partial bitstreams, but the STARTUP primitive cannot be used to deliver partial bitstreams to the configuration engine for devices prior to UltraScale+. The static design would need to be connected to the flash via user IO, and a controller would be used to fetch bitstreams for delivery to the ICAP.
2. JTAG mode is always available independent of the Mode pin settings. Setting the Mode pins to JTAG-only (M[2:0]=101) is not recommended, as this disrupts partial bitstream delivery to other configuration ports, including ICAP.

To use external configuration modes (other than JTAG) for loading a partial BIT file, these pins must be reserved for use after the initial device configuration. This is achieved by using the `BITSTREAM.CONFIG.PERSIST` property to keep the dual-purpose I/O for configuration usage and to set the configuration width. Refer to the *Vivado Design Suite User Guide: Programming and Debugging* (UG908). The Tcl command syntax to set this property is:

```
set_property BITSTREAM.CONFIG.PERSIST <value> [current_design]
```

where `<value>` is either `No` or `Yes`.

Note: When configuration pins are persisted, the ICAP is disabled; the two features are mutually exclusive. For more information on the ICAP, see *7 Series FPGAs Configuration User Guide* (UG470) or *UltraScale Architecture Configuration User Guide* (UG570), depending on your device.

Partial bitstreams contain all the configuration commands and data necessary for Dynamic Function eXchange. The task of loading a partial bitstream into an FPGA does not require knowledge of the physical location of the RM because configuration frame addressing information is included in the partial bitstream. A valid partial bitstream cannot be sent to the wrong part of the FPGA.

A DFX controller retrieves the partial bitstream from memory, then delivers it to a configuration port. The DFX control logic can either reside in an external device (for example, a processor) or in the programmable logic of the FPGA to be reconfigured. A user-designed internal DFX controller loads partial bitstreams through the ICAP interface. As with any other logic in the static design, the internal DFX control circuitry operates without interruption throughout the reconfiguration process.

Internal configuration can consist of either a custom state machine, or an embedded processor such as MicroBlaze. For a Zynq 7000 SoC and Zynq UltraScale+ MPSoC, the Processor Subsystem (PS) can be used to manage partial reconfiguration events.

Note: For Zynq 7000 SoC devices, the Programmable Logic (PL) can be partially reconfigured, but the Processing System cannot.

As an aid in debugging Dynamic Function eXchange designs and DFX control logic, the Vivado Logic Analyzer can be used to load full and partial bitstreams into an FPGA by means of the JTAG port.

For more information on loading a bitstream into the configuration ports, see the Configuration Interfaces chapter in these documents:

- *7 Series FPGAs Configuration User Guide* (UG470)
- *UltraScale Architecture Configuration User Guide* (UG570)
- *Zynq 7000 SoC Technical Reference Manual* (UG585)
- *Spartan UltraScale+ FPGAs Configuration User Guide* (UG860)

Bitstream Type Definitions

When designs are compiled for Dynamic Function eXchange in AMD devices, different types of bitstreams are created. This section defines terminology and explains the details for each type of bitstream for 7 series and UltraScale devices. The types of bitstreams are:

- Full Configuration Bitstreams
- Partial Bitstreams
- Blanking Bitstreams
- Clearing Bitstreams

Full Configuration Bitstreams

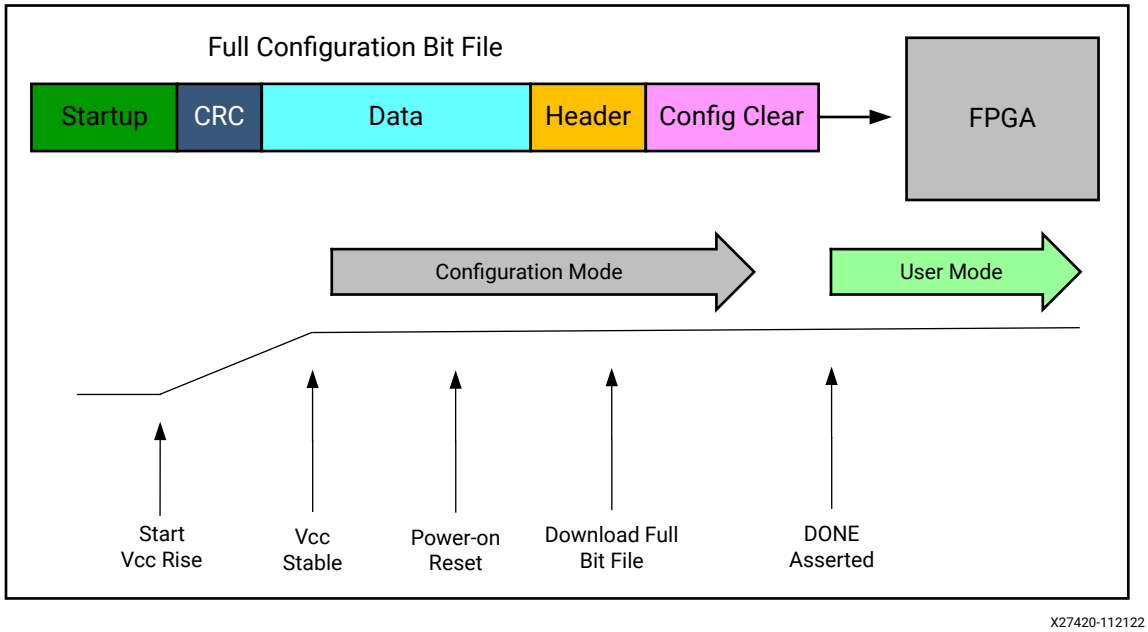
All DFX designs start with standard configuration of the full device using a full configuration bitstream. The format and structure is no different than for a flat design solution (with one exception), and there is no difference in how this bitstream can be used to initially program the FPGA. The one exception is that the global signal mask for a DFX design is closed; it is opened with each partial (or clearing) bitstream to affect only the reconfigurable region. Because of this exception, chip-wide GSR events (after the initial configuration) cannot be issued. However, note that the design itself has been processed in preparation for partial reconfiguration of the device after the full programming has been done. Standard features, such as encryption and compression, are supported.

RPs set as black boxes are supported, so RMs with no functionality can be delivered as part of the initial configuration, to be replaced later with a desired RM. Bitstream compression can be effective in this case, reducing bitstream size and initial configuration time.

Downloading a Full BIT File

The FPGA in a digital system is configured after power on reset by downloading a full BIT file, either directly from a PROM or from a general purpose memory space by a microprocessor. A full BIT file contains all the information necessary to reset the FPGA, configure it with a complete design, and verify that the BIT file is not corrupt. The figure below illustrates this process.

Figure 201: **Configuring with a Full BIT File**



After the initial configuration is completed and verified, the FPGA enters user mode, and the downloaded design begins functioning. If a corrupt BIT file is detected, the DONE signal is never asserted, the FPGA never enters user mode, and the corrupt design never starts functioning.

Partial Bitstreams

Partial bitstreams are delivered during normal device operation to replace functionality in a pre-defined device region. These bitstreams have the same structure as full bitstreams but are limited to specific address sets to program a specific portion of the device. Dedicated DFX features such as per-frame CRC checks (to ensure bitstream integrity) and automatic initialization (so the region starts in a known state) are available, as well as full bitstream features such as encryption and compression.

The size of a partial bitstream is directly proportional to the size of the region it is reconfiguring. For example, if the RP is composed of 20% of the device resources, the partial bitstream is roughly 20% the size of the full design bitstream.

Partial bitstreams are fully self-contained, so they are delivered to an appropriate configuration port. All addressing, header, and footer details are contained within these bitstreams, just as they would be for full configuration bitstreams. You deliver partial bitstreams to the FPGA through any external non-master configuration mode, such as JTAG, Slave Serial, or Slave SelectMap. Internal configuration access includes the ICAP (all devices), PCAP (Zynq 7000 SoC devices), and MCAP (UltraScale and UltraScale+ devices through PCIe).

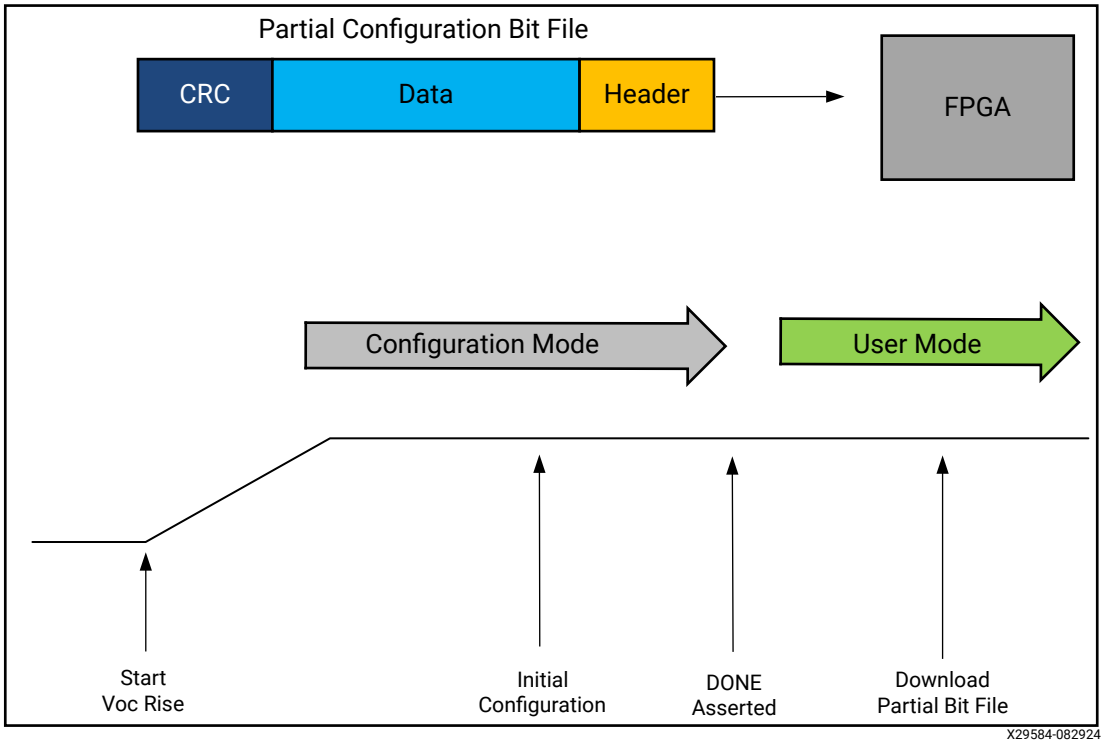
Partial bitstreams are automatically created when `write_bitstream` is run on a DFX configuration. Each partial bitstream file name references your top-level design name, plus the Pblock name for the RP, plus `_partial`. For example, for a full design bit file `top_first.bit`, a partial bit file could be named `top_first_pblock_red_partial.bit`.

The Pblock instance is always the same, regardless of the RM contained within, so it is recommended that you use a descriptive base configuration name or rename the partial bit files to clarify which module it represents.

Downloading a Partial BIT File

A partially reconfigured FPGA is in user mode while the partial BIT file is loaded. This allows the portion of the FPGA logic not being reconfigured to continue functioning while the RP is modified. The following figure illustrates this process.

Figure 202: **Configuring with a Partial BIT File**



The partial BIT file has a simplified header, and there is no startup sequence that brings the FPGA into user mode. The BIT file contains (essentially, and with default settings) only frame address and configuration data, plus a final checksum value. Additional CRC checks can be inserted, if desired, to perform bitstream integrity checking.

If Reset After Reconfiguration is used, the DONE pin pulls LOW when reconfiguration begins and pulls HIGH again when partial reconfiguration successfully completes, although the partial bitstream can still be monitored internally as well.

Note: With UltraScale devices, the DONE pin pulls LOW at the beginning of the clearing bitstream and remains low until the end of the partial bitstream because the two bitstreams together constitute a complete partial reconfiguration sequence. The DONE pin does NOT return high at the end of the clearing bitstream.

If Reset After Reconfiguration is not selected, you must monitor the data being sent to know when configuration has completed. The end of a partial BIT file has a DESYNCH word (0000000D) that informs the configuration engine that the BIT file has been completely delivered. This word is given after a series of padding NO OP commands, ensuring that once the DESYNCH has been reached, all the configuration data has already been sent to the target frames throughout the device. As soon as the complete partial BIT file has been sent to the configuration port, it is safe to release the reconfiguration region for active use.

Blanking Bitstreams

A blanking bitstream is a specific type of partial bitstream, one that represents a logical black box. In Vivado this is referred to as a greybox, as it is not completely empty. It removes the functionality of an existing RM by replacing it with new functionality, which consists simply of tie-off LUTs on all appropriate module I/O.

To create a greybox RM, you remove the logical and physical representation of a fully placed and routed design configuration and replace it with tie-off LUTs. Starting with a routed configuration (with the static design locked) in active memory, run these steps:

```
update_design -cell <foo> -black_box
update_design -cell <foo> -buffer_ports
place_design
route_design
```

The design must be placed and routed to implement the LUTs that have been inserted into the design. Outputs of the greybox RM are tied to ground by default, but can be set to Vcc by setting the HD.PARTPIN_TIEOFF on desired ports.

Compression can be used to greatly reduce the size of blanking bitstreams. Note that these bitstreams still contain, not only the tie-off LUTs, but also any static routing that happens to pass through this region of the FPGA. Blanking bitstreams are generated and named in the same way as standard partial bitstreams, as the greybox variation is saved as another configuration checkpoint.

Advisories had been given for prior versions of Vivado software recommending the use of blanking bitstreams for 7 series and Zynq devices to avoid potential glitching conditions. Starting with Vivado 2016.1, these rare glitching scenarios are automatically avoided by embedding specific blanking events in each partial bitstream. Blanking bitstreams, while still available to remove logic from a RP, are no longer required to avoid any potential glitch events. Automatically embedding blanking events results in an increased size of the partial bit files; compression can be used to reduce these effects.

Clearing Bitstreams

Unlike the bitstream types noted above, this type is for UltraScale devices only (UltraScale+ does not have this requirement). A new requirement for this architecture is to clear an existing module before loading a new module. This clearing bitstream prepares the device for the delivery of any subsequent partial bitstream for that RP by establishing the global signal mask for the region to be reconfigured. Although the existing module is technically not removed (the current logical module remains), it is easiest to think of it this way. If a clearing bitstream is not delivered, the subsequent RM will not be initialized.

Clearing bitstreams are not partial bitstreams. They comprise less than 10% of the frames for the target region and are therefore less than 10% the size of the corresponding partial bitstreams. They do not change the functionality but shut down clocks driving logic in the region. They must be delivered between partial bitstreams and should always be followed as soon as possible by the next partial bitstream.

Each clearing bitstream is built for a specific RM and must be applied after that module has been used, and must be sent to the configuration engine immediately before the next partial bitstream is delivered. For example, to transition from module A to module B, the clearing bitstream for A must be delivered just before the partial bitstream for B is delivered. To transition from module B back to module A, the clearing bitstream for B must be delivered just before the partial bitstream for A is delivered. This is the case even if any partial bitstream in question is a blanking bitstream.

Clearing bitstreams are automatically generated and have the same name as partial bitstreams with `_clear` at the end. Looking at the example above, if `top_first` is an UltraScale device design, the clearing bit file name would be `top_first_pblock_red_partial_clear.bit`.

Dynamic Function eXchange through ICAP for Zynq Devices

The primary configuration mechanism for the programmable logic (PL) of Zynq 7000 and Zynq UltraScale+ MPSoC devices is through the processing system (PS), which delivers bitstreams to the PCAP. The most common mechanism for partial reconfiguration is also through this path. However, to manage partial reconfiguration completely within the PL (either through the PR Controller IP or through a custom-designed controller module), partial bitstreams can also be delivered to the ICAP, similar to FPGA devices.

The PCAP and ICAP interfaces are mutually exclusive and cannot be used simultaneously. Switching between ICAP and PCAP is possible, but you must ensure that no commands or data are being transmitted or received before changing interfaces. Failure to do this could lead to unexpected behavior.

To enable the ICAP for Zynq 7000 devices, set bit 27 (`PCAP_PR`) of the Control Register (`devc.CTRL`). This bit selects between ICAP and PCAP for PL reconfiguration. The default is `PCAP (1)`, but that can be changed to `ICAP (0)` to enable this configuration port. Additionally, bit 28 (`PCAP_MODE`) must also be set to 1, which is the default. For more details, see the *Zynq 7000 SoC Technical Reference Manual* (UG585).

To enable the ICAP for Zynq UltraScale+ MPSoC devices, set the `PCAP_PR` field of the `pcap_ctrl` (CSU) register. This bit selects between ICAP (or MCAP) and PCAP for PL reconfiguration. The default is `PCAP (1)`, but that can be changed to `ICAP` / `MCAP (0)` to enable this configuration port. For more details, see the *Zynq UltraScale+ Device Technical Reference Manual* (UG1085) and the *Zynq UltraScale+ Device Register Reference* (UG1087).

The Zynq UltraScale+ MPSoC Xilfpga library supports the delivery of partial bit files for Linux and bare-metal applications. In the current release, only non-secure bitstreams (without encryption or authentication) are supported. For more information and examples, visit the Xilfpga Wiki page (www.wiki.xilinx.com/xilfpga).

Dynamic Function eXchange for Spartan UltraScale+ Devices

AMD Spartan™ UltraScale+™ devices support DFX solutions with a similar set of capabilities as other AMD UltraScale+™ devices, as the bulk of the programmable logic resources are the same. Configuration details, however, are a bit different and these differences must be considered when partially reconfiguring the device. For extensive information regarding programming these devices, see the *Spartan UltraScale+ FPGAs Configuration User Guide* (UG860).

Spartan UltraScale+ Bitstream Format

For Spartan UltraScale+ FPGAs, bitstream (`.bit`) files are not used. Instead, the programmable device image (PDI) file defines the application-specific FPGA functionality. The full-device PDI file contains the platform loader manager (PLM) firmware `.elf` for boot and configuration and the programmable logic (PL) data `.rcdo` (equivalent to the bitstream data). Partial images do not include the PLM firmware `.elf`, as it has already been loaded in the platform management controller (PMC), but the format of these partial images are also PDI files, containing the PL `.rcdo` data for the user-defined logical region.

When generating full and partial programming images in the Vivado IDE, the Generate Bitstream step creates the full and partial programming images by default, as it does for all DFX designs. However, this command is comprised of two explicit steps: `write_bitstream` and `bootgen`. The `write_bitstream` step creates full and partial `.bit` files, but these files must NOT be used to program the device. The outputs of Bootgen are the desired `.pdi` files that are the ones to send to the Spartan UltraScale+ device.

If you are not using the Vivado IDE in project mode, you can generate full and partial programming images using these two explicit steps. The `write_bitstream` command is used exactly as it is for any other UltraScale+ device, but instead of using the full or partial .bit files, you need to use the `.bif` and `.rcdo` files that are generated (along with the fixed `plm.elf` delivered with Vivado). Use `bootgen` with the `-arch` option set to `spartanup` to create the necessary `.pdi` files.

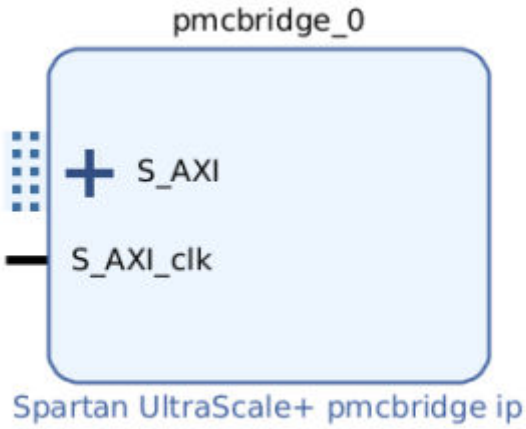
```
bootgen -arch spartanup -image <design.bif> -o <design.pdi>
```

Repeat this command for each full or partial programming image required.

Spartan UltraScale+ Device Programming

Spartan UltraScale+ devices do have ICAP resources, but they are reserved for the soft error mitigation (SEM) IP. Instead, the internal configuration access port is the AXI32 primitive, which provides a hardened AXI4 interface to the platform management controller. The AXI32 primitive can be accessed in IP integrator (IPI) with the PMC Bridge IP or using the XPM_PMC_BRIDGE module to instantiate AXI32 in the user design.

Figure 203: **Spartan UltraScale+ PMC Bridge IP**



In addition to the AXI32 resource, the set of supported configuration modes are listed in Table 22.

***Note*:** The MCAP connection is only available in larger Spartan UltraScale+ devices (xcsu65p and larger) that contain a PCIe® hard block.

Tandem Configuration and Dynamic Function eXchange

UltraScale devices introduced the MCAP, a dedicated connection from one specific PCIe block on a device to the configuration engine, providing an efficient mechanism for delivering partial bitstreams. No explicit routes are required to connect the PCIe block to the configuration engine, saving considerable resources.

The MCAP is enabled by customizing an AMD PCIe IP with Dynamic Function eXchange or Tandem Configuration features. These features are available for three IP cores that support PCI Express:

- *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156)
- *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194)
- *DMA/Bridge Subsystem for PCI Express Product Guide* (PG195)
- *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)

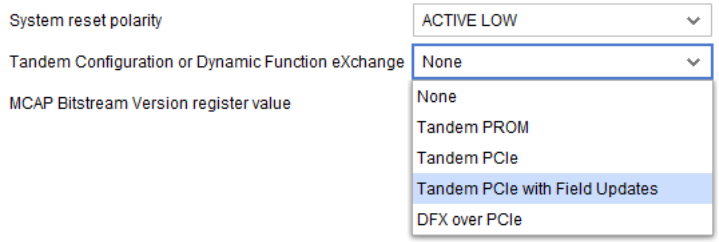
Tandem Configuration uses a two-stage methodology that enables the IP to meet the configuration time requirements indicated in the PCI Express Specification. The following use cases are supported with this technology:

- **Tandem PROM:** Loads the single two-stage bitstream from the flash.
- **Tandem PCIe:** Loads the first stage bitstream from flash, and deliver the second stage bitstream over the PCIe link to the MCAP.
- **Tandem with Field Updates:** After a Tandem PROM (UltraScale only) or Tandem PCIe (UltraScale or UltraScale+) initial configuration, update the entire user design while the PCIe link remains active. The update region (floorplan) and design structure are pre-defined, and Tcl scripts are provided.
- **Tandem + Dynamic Function eXchange:** This is a more general case of Tandem Configuration followed by Dynamic Function eXchange of any size or number of dynamic regions.
- **DFX over PCIe:** This is a standard configuration followed by DFX, using the PCIe / MCAP as the delivery path of partial bitstreams.

The Tandem and DFX combined solutions have few additional requirements. This approach requires that the Pblocks for the `HD.TANDEM_IP_PBLOCK` and `HD.RECONFIGURABLE` parts of the design do not overlap. Otherwise, like a standard DFX design, any number or size RPs can be defined.

To enable any of these capabilities, select the appropriate option when customizing the core. In the **Basic** tab:

1. Change the **Mode** to **Advanced**.
2. Change the Tandem Configuration or Partial Reconfiguration option according to your desired case in UltraScale:
   - **Tandem** for Tandem PROM, Tandem PCIe or Tandem + Dynamic Function eXchange use cases
   - **Tandem with Field Updates** ONLY for the pre-defined Field Updates use case
   - **DFX over PCIe** to enable the MCAP link for Dynamic Function eXchange, without enabling Tandem Configuration
3. Change the Tandem Configuration or Partial Reconfiguration option according to your desired case in UltraScale+:
   - **Tandem PROM** for Tandem PROM or Tandem + Dynamic Function eXchange use cases
   - **Tandem PCIe** for Tandem PCIe or Tandem + Dynamic Function eXchange use cases
   - **Tandem PCIe with Field Updates** ONLY for the pre-defined Field Updates use case; Tandem PROM does not support Field Updates in UltraScale+
   - **DFX over PCIe** to enable the MCAP link for Dynamic Function eXchange, without enabling Tandem Configuration



The PCIe block that must be selected in most cases is the lowest instance in the device, except for SSI devices with three super logic regions (SLRs), in which case it is the lowest PCIe instance in the center SLR. A complete listing of the specific supported blocks is shown in Table 23. All other PCIe blocks do not have the dedicated MCAP feature.

Tandem with Field Updates: Tandem (PCIe) with Field Updates is a predefined design structure that combines Tandem and DFX in a single design.

- **In UltraScale devices:** both stage 1 and stage 2 must come from the same design image. When it is time for the field update event to occur, clearing bitstreams are required and partial bitstreams, not stage 2 bitstreams, are delivered to change the application.
- **In UltraScale+ devices:** Reconfigurable Stage 2 are supported. This means that after stage 1 is loaded, any compatible stage 2 bitstream can be delivered over the PCIe link to complete the initial configuration. When it is time for the field update event to occur, any compatible stage 2 bitstream can be used as a partial bitstream to change the application.



TIP: *It is expected that any designer using this Field Updates solution starts with the example design generated by the customized IP as the starting point.*

For complete information about Tandem Configuration, including required PCIe block locations, design flow examples, requirements, restrictions, flow details for Field Updates, and other considerations, see the Tandem Configuration section in *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156) for UltraScale devices. For UltraScale+ devices, see *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213).

Table 23: **UltraScale: PCIe Block and Reset Locations Supporting DFX, by Device**

Device	Package PCIe Block	PCIe Reset Location	Status
Kintex UltraScale			
KU025	PCIE_3_1_X0Y0	IOB_X1Y103	Production
KU035	PCIE_3_1_X0Y0	IOB_X1Y103	Production
KU040	PCIE_3_1_X0Y0	IOB_X1Y103	Production
KU060	PCIE_3_1_X0Y0	IOB_X2Y103	Production
KU085	PCIE_3_1_X0Y0	IOB_X2Y103	Production
KU095	PCIE_3_1_X0Y0	IOB_X1Y103	Production
KU115	PCIE_3_1_X0Y0	IOB_X2Y103	Production
AMD Virtex™ UltraScale™			
XVU065	PCIE_3_1_X0Y0	IOB_X1Y103	Production
VU080	PCIE_3_1_X0Y0	IOB_X1Y103	Production
VU095	PCIE_3_1_X0Y0	IOB_X1Y103	Production
VU125	PCIE_3_1_X0Y0	IOB_X1Y103	Production
VU160	PCIE_3_1_X0Y1	IOB_X1Y363	Production
VU190	PCIE_3_1_X0Y2	IOB_X1Y363	Production
VU440	PCIE_3_1_X0Y2	IOB_X1Y363	Production

To easily find the package pin for the dedicated PCIe Reset for UltraScale devices, issue the following Tcl command:

```
get_package_pins -of_objects [get_sites IOB_X1Y103]
```

Table 24: **UltraScale+: PCIe Block Locations Supporting DFX, by Device**

Device	Package PCIe Block	Status[1]
Kintex UltraScale+		
KU3P	PCIE40E4_X0Y0	Production
KU5P	PCIE40E4_X0Y0	Production
KU11P	PCIE40E4_X1Y0	Production
KU15P	PCIE40E4_X1Y0	Production
KU19P	no MCAP enabled PCIe site	Unsupported for PCIe delivery[2]
Virtex UltraScale+		
VU3P	PCIE40E4_X1Y0	Production
VU5P	PCIE40E4_X1Y0	Production

Table 24: **UltraScale+: PCIe Block Locations Supporting DFX, by Device** *(cont'd)*

Device	Package PCIe Block	Status[1]
VU7P	PCIE40E4_X1Y0	Production
VU9P	PCIE40E4_X1Y2	Production
VU11P	PCIE40E4_X0Y0	Production
VU13P	PCIE40E4_X0Y1	Production
VU19P	PCIE4CE4_X0Y2	Production
VU23P	PCIE4CE4_X0Y0	Production
VU27P	PCIE40E4_X0Y	Production
VU29P	PCIE40E4_X0Y0	Production
VU31P	PCIE4CE4_X1Y0	Production
VU33P	PCIE4CE4_X1Y0	Production
VU35P	PCIE4CE4_X1Y0	Production
VU37P	PCIE4CE4_X1Y0	Production
VU45P	PCIE4CE4_X1Y0	Production
VU47P	PCIE4CE4_X1Y0	Production
VU57P	PCIE4CE4_X1Y0	Production
AMD Zynq™ UltraScale+™ MPSoC		
ZU4EV/EG/CC	PCIE40E4_X0Y1	Production
ZU5EV/EG/CC	PCIE40E4_X0Y1	Production
ZU7EV/EG/CC	PCIE40E4_X0Y1	Production
ZU11EG	PCIE40E4_X1Y0	Production
ZU17EG	PCIE40E4_X1Y0	Production
ZU19EG	PCIE40E4_X1Y0	Production

Notes:

1. For the most up-to-date information on core and device support status, consult the product guide for the specific version of the IP you wish to use.
2. The KU19P has no master PCIe block instance; none of the three PCIe blocks in the device contain the MCAP connection to the configuration engine. Tandem Configuration is not supported for this device, and any partial bitstream delivery via PCIe must be sent to the ICAP.

***Note*:** Any device not listed in this table does not have a PCIe site in the programmable logic portion of the device, or, like Zynq RFSoC devices, does not have an MCAP-enabled PCIe site in the programmable logic. Unlike UltraScale, UltraScale+ does not have a dedicated connection to a PCIe Reset pin, but AMD recommends using a pin in Bank 65.

The MCAP is capable of operating at 200 MHz with a 32-bit data path. Traditionally bitstreams are loaded into the MCAP from a host PC through PCI Express configuration packets. In these systems the host PC and host PC software are the main factors which limit MCAP performance and bitstream throughput. Because PCIe performance of specific host PC and host PC software can vary widely, overall MCAP performance throughput might vary.

For more information and sample drivers, see Answer Record 64761.

If the performance of partial bitstream delivery using the MCAP port is insufficient, the ICAP can be used instead. While this approach does require additional logic to funnel configuration data from the PCIe end point to this internal configuration port, the ICAP can be saturated with 32-bit configuration data at the maximum clock rate (200 MHz for monolithic devices, 125 MHz for SSI devices). See *Fast Partial Reconfiguration Over PCI Express Application Note* (XAPP1338) for more information and an example design.

Tandem Configuration and Versal Devices

Versal devices support tandem configuration for PCIe end points within the CPM or MMI. While the device image structure is quite different, the concept of a two-stage load remains. Both tandem PROM and tandem PCIe variations are available for devices with CPM instances; tandem is not supported for the PL-based PCIe sites.

For more information, refer to the following product guides:

- *Versal Adaptive SoC CPM Mode for PCI Express Product Guide* (PG346)
- *Versal Adaptive SoC CPM DMA and Bridge Mode for PCI Express Product Guide* (PG347)

With Segmented Configuration, the boot PDI natively includes tandem configuration when you configure the CPM or MMI as a PCIe end point. You do not have to select the tandem mode option in the CIPS IP or PS wizard IP.

Formatting BIN Files for Delivery to Internal Configuration Ports

Partial bit files have the same basic format as full bit files, but they are reduced to the set of configuration frames for the target region and restricted to the set of events that make sense for active devices. Partial bit files can be:

- Delivered to external interfaces, such as JTAG or slave configuration ports.
- Reformatted as BIN files to be delivered to the internal configuration ports: ICAP (7 series or UltraScale devices), PCAP (Zynq devices only) or MCAP (UltraScale devices only).

Generate BIN files using the `write_cfgmem` utility. Three options are critical:

- Set `-format` as BIN to generate that file type.
- Use `-interface` to select the SelectMap width, and use `SMAPx32` for PCAP or `MCAP` for UltraScale ICAP.
  - `SMAPx16` and `SMAPx8` (default) can also be used for the 7 series ICAP.
  - `SMAPx8` is required for 7 series encrypted partial bitstreams.

- You must use `-disablebitswap` to target the PCAP or MCAP.

Examples

ICAP (for 7 series devices)

```
write_cfgmem -format BIN -interface SMAPx8 -loadbit "up 0x0
<partial_bitfile>"
```

ICAP (for UltraScale devices)

```
write_cfgmem -format BIN -interface SMAPx32 -loadbit "up 0x0
<partial_bitfile>"
```

PCAP (for Zynq 7000 SoC devices) or MCAP (for one specific PCIe block per UltraScale device)

```
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit "up
0x0
<partial_bitfile>"
```

Summary of BIT Files for UltraScale Devices

This section applies specifically to UltraScale devices and does not apply to 7 series, UltraScale+, Zynq, or Zynq MPSoC devices. With the finer granularity of global signals (that is, GSR) and the ability to reconfigure new element types, a new configuration process is necessary. Prior to loading in a partial bitstream for a new RM, the existing RM must be cleared. This clearing bitstream prepares the device for delivery of any subsequent partial bitstream for that RP by establishing the global signal mask for the region to be reconfigured. Although the existing module technically is not removed, it is easiest to think of it this way.

When running `write_bitstream` on a design configuration with RPs, a clearing BIT file per RP is created. For example, take a design in which two RPs (RP1 and RP2), with two RMs each, A1 and B1 into RP1, and A2 and B2 into RP2, have been implemented. Two configurations (configA and configB) have been run through place and route, and `pr_verify` has passed. When bitstreams are generated, each configuration produces five bitstreams. For configA, these could be named:

- **`configA.bit`:** This is the full design bitstream that is used to configure the device from power-up. This contains the static design plus functions A1 and A2.
- **`configA_RP1_A1_partial.bit`:** This is the partial BIT file for function A1. This is loaded after another RM has been cleared from this RP.
- **`configA_RP1_A1_partial_clear.bit`:** This is the clearing BIT file for function A1. Before loading in any other partial BIT file into RP1 *after function* A1, this file must be loaded.

- `configA_RP2_A2_partial.bit`: This is the partial BIT file for function A2. This is loaded after another RM has been cleared from this RP.
- `configA_RP2_A2_partial_clear.bit`: This is the clearing BIT file for function A2. Before loading in any other partial BIT file into RP2 *after function* A2, this file must be loaded.

Likewise, configB produces five similar bitstreams:

- `configB.bit`: This is the full design bitstream that is used to configure the device from power-up. This contains the static design plus functions B1 and B2.
- `configB_RP1_B1_partial.bit`: This is the partial BIT file for function B1. This is loaded after another RM has been cleared from this RP.
- `configB_RP1_B1_partial_clear.bit`: This is the clearing BIT file for function B1. Before loading in any other partial BIT file into RP1 *after function* B1, this file must be loaded.
- `configB_RP2_B2_partial.bit`: This is the partial BIT file for function B2. This is loaded after another RM has been cleared from this RP.
- `configB_RP2_B2_partial_clear.bit`: This is the clearing BIT file for function B2. Before loading in any other partial BIT file into RP2 *after function* B2, this file must be loaded.

The sequence for any reconfiguration is to first load a clearing BIT file for a current RM, immediately followed by a new RM. For example, to transition RP RP1 from function A1 to function B1, first load the BIT file `configA_RP1_A1_partial_clear.bit`, then load `configB_RP1_B1_partial.bit`. The first bitstream prepares the region by opening the mask, and the second bitstream loads the new function, initializes only that region, then closes the mask.

If a clearing bit file is not loaded, initialization routines (GSR) have no effect. If a clearing file for a different RP is loaded, then that RP is initialized instead of the one that has been just reconfigured. If the incorrect clearing file for the proper RP is used, the current RM or possibly even the static design could be disrupted until the following partial bit file has been loaded.

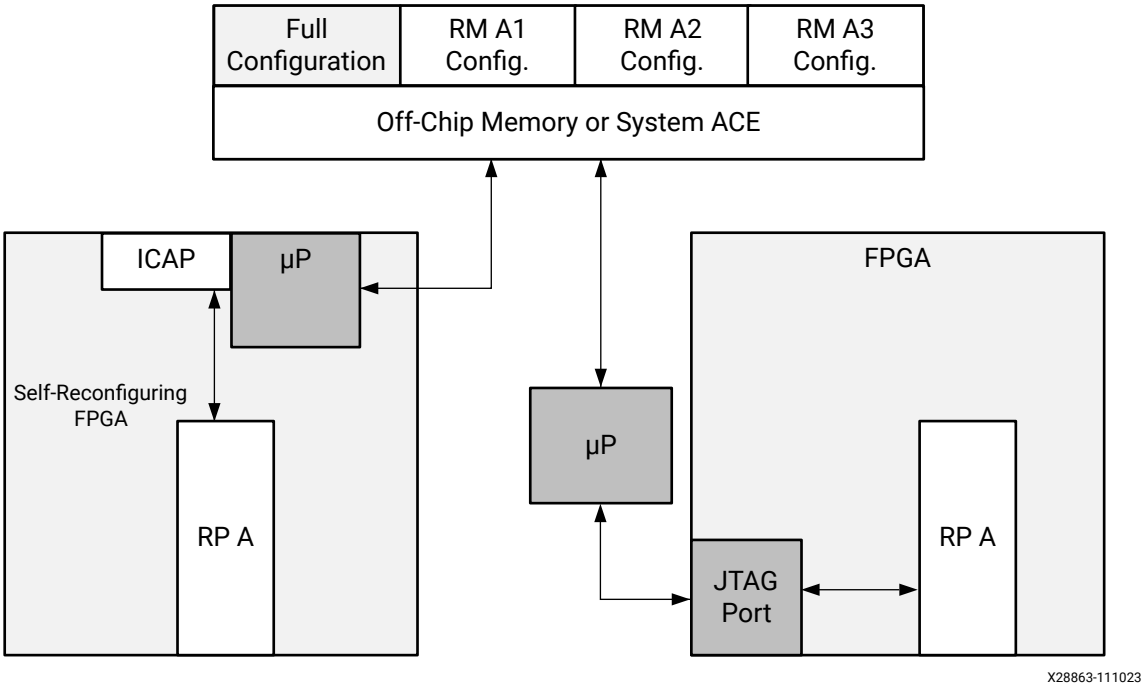
System Design for Configuring an FPGA

A partial BIT file can be downloaded to the FPGA in the same manner as a full BIT file. An external microprocessor determines which partial BIT file should be downloaded, where it exists in an external memory space, and directs the partial BIT file to a standard FPGA configuration port such as JTAG, Select MAP or serial interface. The FPGA processes the partial BIT file correctly without any special instruction that it is receiving a partial BIT file.

It is common to assert the INIT or PROG signals on the FPGA configuration interface before downloading a full BIT file. This must not be done before downloading a partial BIT file, as that would indicate the delivery of a full BIT file, not a partial one.

Any indication to the working design that a partial BIT file will be sent (such as holding enable signals and disabling clocks) must be done in the design—and not by means of dedicated FPGA configuration pins. The following figure shows the process of configuring through a microprocessor.

Figure 204: **Configuring Through a Microprocessor**



In addition to the standard configuration interfaces, Dynamic Function eXchange supports configuration by means of the Internal Configuration Access Port (ICAP). The ICAP protocol is identical to SelectMAP and is described in the Configuration User Guide for the target device. The ICAP library primitive can be instantiated in the HDL description of the FPGA design, thus enabling analysis and control of the partial BIT file before it is sent to the configuration port. The partial BIT file can be downloaded to the FPGA through general purpose I/O or gigabit transceivers and then routed to the ICAP in the FPGA programmable logic.

Rules for DFX ports and formats:

- Encrypted partial bitstreams can be delivered to any port for FPGA devices, but only when the initial configuration was also encrypted. The same key must be used for all bitstreams.
  - Encrypted partial bitstreams cannot be delivered to the ICAP or MCAP for Zynq SoC devices,
- If the initial configuration of a device is encrypted, unencrypted partial bitstreams can be used only if they are delivered to the ICAP, which is a trusted port.

- Bitstream authentication for partial bitstreams is not supported for Virtex or Kintex devices, only for Zynq devices, delievered to the PCAP. However, authentication can be used for the initial full-device programming of any device.
- The ICAP must be used with an 8-bit bus only for Dynamic Function eXchange for encrypted 7 series BIT files.
- Reconfiguration through external configuration ports is permitted only when bitstream readback security is not set to Level 2.

Partial BIT File Integrity

Error detection and recovery of partial BIT files have unique requirements compared to loading a full BIT file. If an error is detected in a full BIT file when it is being loaded into an FPGA, the FPGA never enters user mode. The error is detected after the corrupt design has been loaded into configuration memory, and specific signals are asserted to indicate an error condition. Because the FPGA never enters user mode, the corrupt design never becomes active. You must determine the system behavior for recovering from a configuration error such as downloading a different BIT file if the error condition is detected.

When you download partial BIT files, you cannot use this methodology for error detection and recovery. The FPGA is by definition already in user mode when the partial BIT file is loaded. Because the configuration circuitry supports error detection only after a BIT file has been loaded, a corrupt partial BIT file can become active, potentially damaging the FPGA if left operating for an extended period of time.

If a CRC error is detected during a partial reconfiguration, it asserts the INIT_B pin of the FPGA (INIT_B goes Low to indicate a CRC error). In UltraScale devices, this behavior is echoed on the PRERROR output pin of the ICAP. It is important to note that if a system monitors INIT_B for CRC errors during the initial configuration, a CRC error during a partial reconfiguration might trigger the same response. To detect the presence of a CRC error from within the FPGA, the CRC status can be monitored through the ICAP block. The Status Register (STAT) indicates that the partial BIT file has a CRC error, by asserting the CRC_ERROR flag (bit 0).

There are two types of partial BIT file errors to consider: data errors and address errors (the partial BIT file is essentially address and data information). Given that static routes are free to pass through reconfigurable regions, both types of errors can corrupt the static design, although the likelihood is very small. The only method for completely safe recovery is to download a new full BIT file to ensure the state of the static logic, which requires the entire FPGA to be reset.

Many systems do not need a complex recovery mechanism because resetting the entire FPGA is not critical, or the partial BIT file is stored locally. In that case, the chance of BIT file corruption is not appreciable. Systems in which the BIT files are at risk of becoming corrupted (such as sending the partial BIT file over a radio link) should use a dedicated silicon feature that avoids the problem.

The configuration engines of all AMD devices from 7 series through Versal devices, have the ability to perform a frame-by-frame CRC check and do not load a frame into the configuration memory if that CRC check fails. A failure is reported on the INIT_B pin (it is pulled Low) and gives you the opportunity to take the next steps: retry the partial bit file, fall back to a golden partial bit file, etc. The partially loaded reconfiguration region does not have valid programming in it, but the CRC check ensures the remainder of the device (static region and any other RMs) stays operational while the system recovers from the error.

To enable this feature for these devices, set the `PerFrameCRC` property prior to running `write_bitstream`. The default is No, and Yes inserts the extra CRC checks. The size of an uncompressed bit file increases four to five percent with this option enabled. Note that this feature is not compatible with bitstream compression. No other specific design considerations are necessary to select this option, but your partial reconfiguration controller solution should be designed to choose the course of action should the INIT_B pin indicate a failure has occurred.

The syntax for setting the `PerFrameCRC` property is:

```
set_property bitstream.general.perFrameCRC yes [current_design]
```

This property inserts per-frame CRC checks in all bitstreams created from the current checkpoint, not just partial bitstreams. Full device bitstreams for the initial configuration of the device would also contain the extra CRC checks.

After a partial bit file has been loaded (with or without the per-frame CRC checks), the overall configuration of the device has changed. If the POST_CRC feature for SEU mitigation is enabled, the SEU mitigation engine automatically recalculates the embedded SEU CRC value after the partial bitstream has been loaded and after you have de-synced the configuration interface. Upon completion of the CRC recalibration, the FRAME_ECCE2 FRAME_VALID output toggles again to indicate that SEU detection has resumed.

Configuration Frames

All user-programmable features inside AMD FPGA and SoC devices are controlled by volatile memory cells that must be configured at power-up. These memory cells are collectively known as configuration memory. They define the LUT equations, signal routing, IOB voltage standards, and all other aspects of the design.

AMD FPGA and SoC architectures have configuration memory arranged in frames that are tiled about the device. These frames are the smallest addressable segments of the device configuration memory space, and all operations must therefore act upon whole configuration frames.

Reconfigurable Frames are built upon these configuration frames, and these are the minimum building blocks for performing dynamic reconfiguration. All dimensions based on the fundamental element described.

- Base Regions in 7 series FPGAs are a full clock region high:
  - **CLB:** 50 high by 1 wide
  - **DSP48:** 10 high by 1 wide
  - **Block RAM:** 10 high by 1 wide

The granularity of global signals (GSR) in 7 series is at the clock region level, so full columns of elements are initialized upon reconfiguration. Even if RESET_AFTER_RECONFIGURATION is not used, partial bitstream composition is based on clock region high columns.

- Base Regions in UltraScale and UltraScale+ FPGAs are a full clock region high:
  - **CLB:** 60 high by 1 wide
  - **DSP48:** 24 high by 1 wide
  - **Block RAM:** 12 high by 1 wide
  - **I/O and Clocking:** 52 I/O (one bank), plus related XiPhy, MMCM, and PLL resources
  - **Gigabit Transceivers:** Four high (one quad, plus related clocking resources)

The granularity of global signals (GSR) in UltraScale and UltraScale+ is at the element level, so individual elements are initialized upon reconfiguration, and RESET_AFTER_RECONFIGURATION is not needed. However, partial bitstream composition is still based on clock region high columns, and any static logic elements in these regions are NOT initialized after reconfiguration even though they are contained within the partial bitstream.

Configuration Time

The speed of configuration is directly related to the size of the partial BIT file and the bandwidth of the configuration port. The different configuration ports in each device family have the maximum bandwidths shown in the following tables.

Table 25: **Maximum Bandwidths for Configuration Ports in 7 Series Devices**

Configuration Mode	Max Clock Rate (MHz)	Data Width (Bits)	Maximum Bandwidth
ICAP	100	32	3.2 Gb/s
SelectMAP	100	32	3.2 Gb/s
Serial Mode	100	1	100 Mb/s
JTAG	66	1	66 Mb/s

Table 26: **Maximum Bandwidths for Configuration Ports in UltraScale Devices**

Configuration Mode	Max Clock Rate (MHz)	Data Width (Bits)	Maximum Bandwidth
ICAP/MCAP	200	32	6.4 Gb/s
ICAP/MCAP (SSI)[1]	125	32	4.0 Gb/s
SelectMAP	125	32	4.0 Gb/s
Serial Mode	150	1	150 Mb/s
Serial (SSI Devices)	100	1	100 Mb/s
JTAG	50	1	50 Mb/s
JTAG (SSI Devices)	20	1	20 Mb/s

Notes:

1. When delivered to the master SLR to program any SLR; when programming to the same SLR as the ICAP, the faster rate can be used. Configuration clock frequency maximums might be less than these values depending on the speed grade or operating voltage. Consult the Data Sheet for your target device for more information.

Table 27: **Maximum Bandwidths for Configuration Ports in UltraScale+ Devices**

Configuration Mode	Max Clock Rate (MHz)	Data Width (Bits)	Maximum Bandwidth
ICAP/MCAP	200	32	6.4 Gb/s
ICAP/MCAP (SSI)[1]	125	32	4.0 Gb/s
BPI	125	16	2.0 Gb/s
QSPI	125	4	500 Mb/s
SelectMAP	125	32	4.0 Gb/s
Serial Mode	125	1	125 Mb/s
JTAG	66	1	66 Mb/s
JTAG (SSI Devices)	20	1	20 Mb/s

Notes:

1. When delivered to the master SLR to program any SLR; when programming to the same SLR as the ICAP, the faster rate can be used. Configuration clock frequency maximums might be less than these values depending on the speed grade or operating voltage. Consult the Data Sheet for your target device for more information.

In addition to being reported by `write_bitstream`, The exact bitstream length is available in the created `.rbt` file by using the `-raw_bitfile` option for `write_bitstream`. Use this number along with the bandwidth to calculate the total configuration time. Here is an example of the header in a raw bit file:

```
Xilinx ASCII Bitstream
Created by Bitstream 2023.1
Design name: led_shift_count;UserID=0XFFFFFFFF
Architecture: kintex7
Part:      7k325tffg900
Date:      Mon Mar 16 16:42:05 2015
Bits:      1211072
11111111111111111111111111111111
```

Configuration Debugging

The ICAP interface can be use used to monitor the configuration process when it is used as the port for delivering bitstreams. The "O" port of the ICAP block is a 32-bit bus, but only the lowest byte is used. The mapping of the lower byte is as follows:

Table 28: **ICAP "O" Port Bits**

ICAP "O" Port Bits	Status Bit	Meaning
O[7]	CFGERR_B	Configuration error (active-Low) 0 = A configuration error has occurred. 1 = No configuration error.
O[6]	DALIGN	Sync word received (active-High) 0 = No sync word received. 1 = Sync word received by interface logic.
O[5]	RIP	Readback in progress (active-High) 0 = No readback in progress. 1 = A readback is in progress.
O[4]	IN_ABORT_B	ABORT in progress (active-Low) 0 = Abort is in progress. 1 = No abort in progress.
O[3:0]	1	Reserved

The most significant nibble of this byte reports the status. These Status bits indicate whether the Sync word been received and whether a configuration error has occurred. The following table displays the values for these conditions.

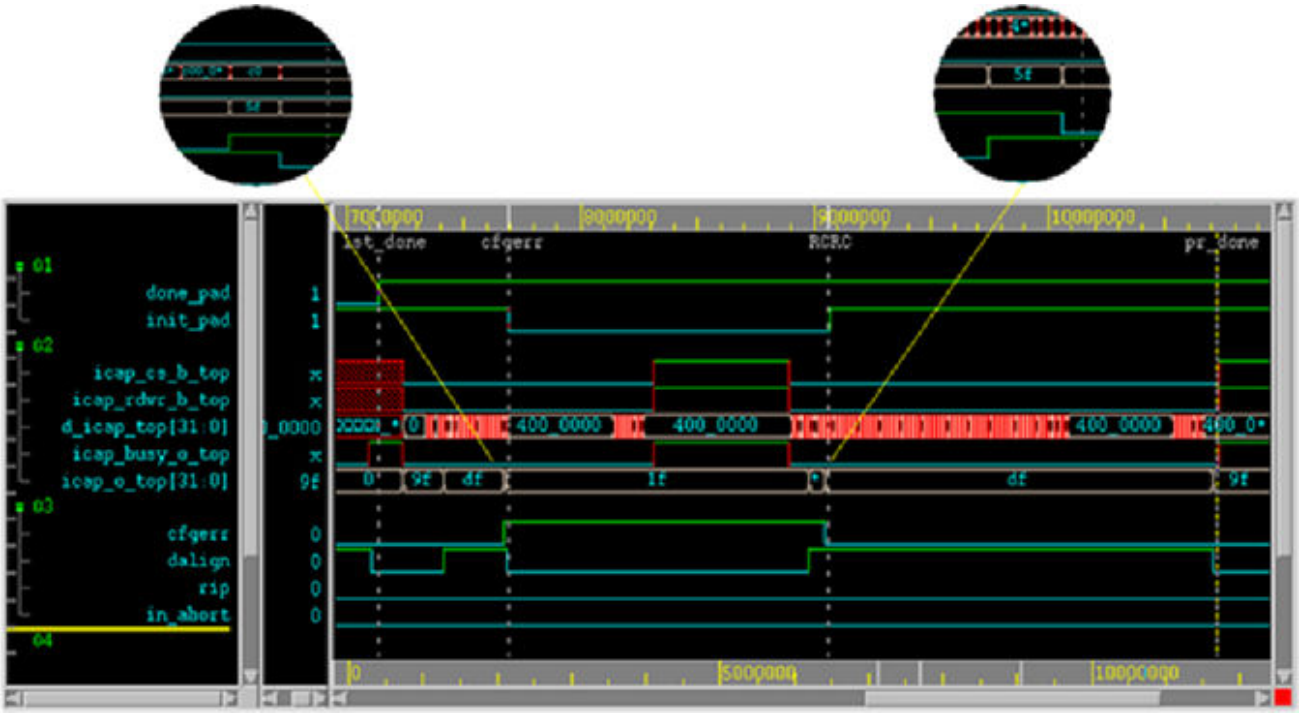
Table 29: **ICAP Sync Bits**

O[7:4]	Sync Word?	CFGERR?
9	No Sync	No CFGERR
D	Sync	No CFGERR
5	Sync	CFGERR
1	No Sync	CFGERR

***Note*:** In the above table, the entries in the first column are different depending on the board. For 7 series devices, they end with "F", so 9F, DF, etc. For UltraScale+ devices, they end with "B", so 9B, DB, etc.

The following figure shows a completed full configuration, followed by a partial reconfiguration with a CRC error, and finally a successful partial reconfiguration. Using the table above, and the description below, you can see how the "O" port of the ICAP can be used to monitor the configuration process. If a CRC error occurs, these signals can be used by a configuration state machine to recover from the error. These signals can also be used by Vivado Logic Analyzer to capture a configuration failure for debug purposes. With this information Vivado Logic Analyzer can also be used to capture the various points of a partial reconfiguration.

Figure 205: **Vivado Logic Analyzer Display for Dynamic Function eXchange**



The markers in the Vivado Logic Analyzer display indicate the following:

- **1st_done:** This marker indicates the completion of the initial full bitstream configuration. The DONE pin (`done_pad` in this waveform) goes HIGH.
- **cfgerr:** This marker indicates a CRC error is detected while loading partial bitstream. The status can be observed through O[31:0] (`icap_o_top[31:0]` in the waveform).
  - `Icap_o_top[31:0]` starts at 0x9F
  - After seen SYNC word, `Icap_o_top[31:0]` change to 0xDF
  - After detect CRC error, `Icap_o_top[31:0]` change to 0x5F for one cycle, and then switches to 0x1F
  - INIT_B pin is pulled Low (`init_pad` in the waveform)

- **RCRC:** This marker indicates when the partial bitstream is loaded again. The RCRC command resets the `cfgerr` status, and removes the pull-down on the INIT_B pin (`init_pad` in this waveform).
  - `Icap_o_top[31:0]` change from 0x1F to 0x5F when the SYNC word is seen
  - `Icap_o_top[31:0]` change from 0x5F to 0xDF when RCRC command is received
- **pr_done:** This marker indicates a successful partial reconfiguration.
  - `Icap_o_top[31:0]` change from 0xDF to 0x9F when the DESYNC command is received and no configuration error is detected.

In addition to the techniques described above, the UltraScale architecture introduced two new dedicated ports on the ICAP to aid in Dynamic Function eXchange:

- The PRDONE pin is intended to echo the external DONE pin. However, it should only be used for the following:
  - Monolithic devices
  - SSI devices when the RP is on the master SLR
  - When the ICAP used is on the same SLR as the RP.

  Any partial bitstream that configures a slave SLR from the master ICAP has multiple PRDONE events due to the construction of these partial bitstreams. Instead, use the EOS pin on the STARTUP block on the master SLR (or on the target SLR if the RP is not on the master SLR) as a reliable indication of the completion of partial reconfiguration.
- The PRERROR pin echoes the external INIT_B pin. It drops LOW when a CRC error occurs, either with the standard full CRC value at the end of the bit file, or with any per-frame CRC value. The value resets after each segment of the partial bitstream so the failure pulse must be captured by the user design.

For more information on configuration behavior, see the *UltraScale Architecture Configuration User Guide* (UG570).

***Note*:** The netlist insertion flow is not supported for UltraScale and UltraScale+ DFX designs, including designs that contain a Debug Bridge IP component. Use the ILA instantiation method instead to add debugging probes for DFX.

Using Vivado Debug Cores

Vivado tool debug cores (such as ILA and VIO) can be placed in any part of a Dynamic Function eXchange design, including within RMs. A specific design methodology is necessary to connect these cores to a central Debug Hub for communication throughout the device.

The connectivity between the central Debug Hub in static can be set up automatically, and this is triggered by a specific naming convention for the port names on the RP. These twelve pins are required, and the hub is inferred if these exact names are used. Examples in Verilog, VHDL, and block design are below.

***Note*:** The automatic connectivity using BSCAN ports is specific to architectures previous to Versal. Versal designs use an AXI4 connection for Debug instead of BSCAN, and it is suggested that Debug Cores be added via a block design or through the modular NoC feature.

The following is the Verilog instantiation in the static design:

```
my_count counter_inst (
.clk(my_clk),
.dout(dout),
.S_BSCAN_drck(),
.S_BSCAN_shift(),
.S_BSCAN_tdi(),
.S_BSCAN_update(),
.S_BSCAN_sel(),
.S_BSCAN_tdo(),
.S_BSCAN_tms(),
.S_BSCAN_tck(),
.S_BSCAN_runtest(),
.S_BSCAN_reset(),
.S_BSCAN_capture(),
.S_BSCAN_bscanid_en()
);
```

The following is the VHDL component declaration and instantiation in the static design:

```
component my_count is
Port ( clk : in STD_LOGIC;
dout : out STD_LOGIC;
S_BSCAN_drck: IN std_logic := '0';
S_BSCAN_shift: IN std_logic := '0';
S_BSCAN_tdi: IN std_logic := '0';
S_BSCAN_update: IN std_logic := '0';
S_BSCAN_sel: IN std_logic := '0';
S_BSCAN_tdo: OUT std_logic;
S_BSCAN_tms: IN std_logic := '0';
S_BSCAN_tck: IN std_logic := '0';
S_BSCAN_runtest: IN std_logic := '0';
S_BSCAN_reset: IN std_logic := '0';
S_BSCAN_capture: IN std_logic := '0';
S_BSCAN_bscanid_en: IN std_logic := '0'
);
end component;

counter_inst: my_count
port map (clk => my_clk,
dout => dout,
S_BSCAN_drck => open,
S_BSCAN_shift => open,
S_BSCAN_tdi => open,
S_BSCAN_update => open,
S_BSCAN_sel => open,
S_BSCAN_tdo => open,
S_BSCAN_tms => open,
S_BSCAN_tck => open,
```

```
S_BSCAN_runtest => open,
S_BSCAN_reset => open,
S_BSCAN_capture => open,
S_BSCAN_bscanid_en => open
);
```

***Note*:** These input ports must receive an initial value to use the `open` keyword, and that initial value must be `0`. Ports tied to `1` do not connect to the local debug hub.

Within the RM top-level RTL, leave these twelve ports unconnected. Debug Hubs are inserted as black boxes, one in static and one in each RM during synthesis. These inserted IP are then expanded during `opt_design`. This is done for each RM, even if there are no debug cores within that RM (including greybox RMs).

If these exact port names cannot be used for any reason, such as the need to explicitly connect to multiple BSCAN instances, attributes can be used to drive the Debug Hub insertion. This approach must also be used if the first configuration processed does not have any debug cores present. Inference of Debug Hubs is not done if no debug cores within the RM can be found by the Vivado implementation tools.

In the syntax shown below, do not change anything other than the port names to assign Debug ports. These attributes are to be used in all RM top level source files.

The following are the Verilog attributes in the RM top level:

```
(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN drck" *) (*
DEBUG="true" *)
input my_drck;

(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN shift" *) (*
DEBUG="true" *)
input my_shift;

(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN tdi" *) (*
DEBUG="true" *)
input my_tdi;

(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN update" *) (*
DEBUG="true" *)
input my_update;

(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN sel" *) (*
DEBUG="true" *)
input my_sel;

(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN tdo" *) (*
DEBUG="true" *)
output my_tdo;

(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN tms" *) (*
DEBUG="true" *)
input my_tms;

(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN tck" *) (*
DEBUG="true" *)
input my_tck;

(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN runtest" *)
(* DEBUG="true" *)
input my_runtest;

(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN reset" *) (*
DEBUG="true" *)
input my_reset;

(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN capture" *)
(* DEBUG="true" *)
input my_capture;

(* X_INTERFACE_INFO = "xilinx.com:interface:bscan:1.0 S_BSCAN bscanid_en"
*)
(* DEBUG="true" *) input my_bscanid_en;
```

The following are the VHDL attributes in the RM top level:

```
attribute X_INTERFACE_INFO : string;

attribute DEBUG : string;

attribute X_INTERFACE_INFO of my_drck: signal is
"xilinx.com:interface:bscan:1.0 S_BSCAN
drck";

attribute DEBUG of my_drck: signal is "true";

attribute X_INTERFACE_INFO of my_shift: signal is
"xilinx.com:interface:bscan:1.0 S_BSCAN
shift";

attribute DEBUG of my_shift: signal is "true";

attribute X_INTERFACE_INFO of my_tdi: signal is
"xilinx.com:interface:bscan:1.0 S_BSCAN
tdi";

attribute DEBUG of my_tdi: signal is "true";

attribute X_INTERFACE_INFO of my_update: signal is
"xilinx.com:interface:bscan:1.0 S_BSCAN
update";

attribute DEBUG of my_update: signal is "true";

attribute X_INTERFACE_INFO of my_sel: signal is
"xilinx.com:interface:bscan:1.0 S_BSCAN
sel";

attribute DEBUG of my_sel: signal is "true";

attribute X_INTERFACE_INFO of my_tdo: signal is
"xilinx.com:interface:bscan:1.0 S_BSCAN
tdo";

attribute DEBUG of my_tdo: signal is "true";

attribute X_INTERFACE_INFO of my_tms: signal is
"xilinx.com:interface:bscan:1.0 S_BSCAN
tms";

attribute DEBUG of my_tms: signal is "true";

attribute X_INTERFACE_INFO of my_tck: signal is
"xilinx.com:interface:bscan:1.0 S_BSCAN
tck";

attribute DEBUG of my_tck: signal is "true";

attribute X_INTERFACE_INFO of my_runtest: signal is
```

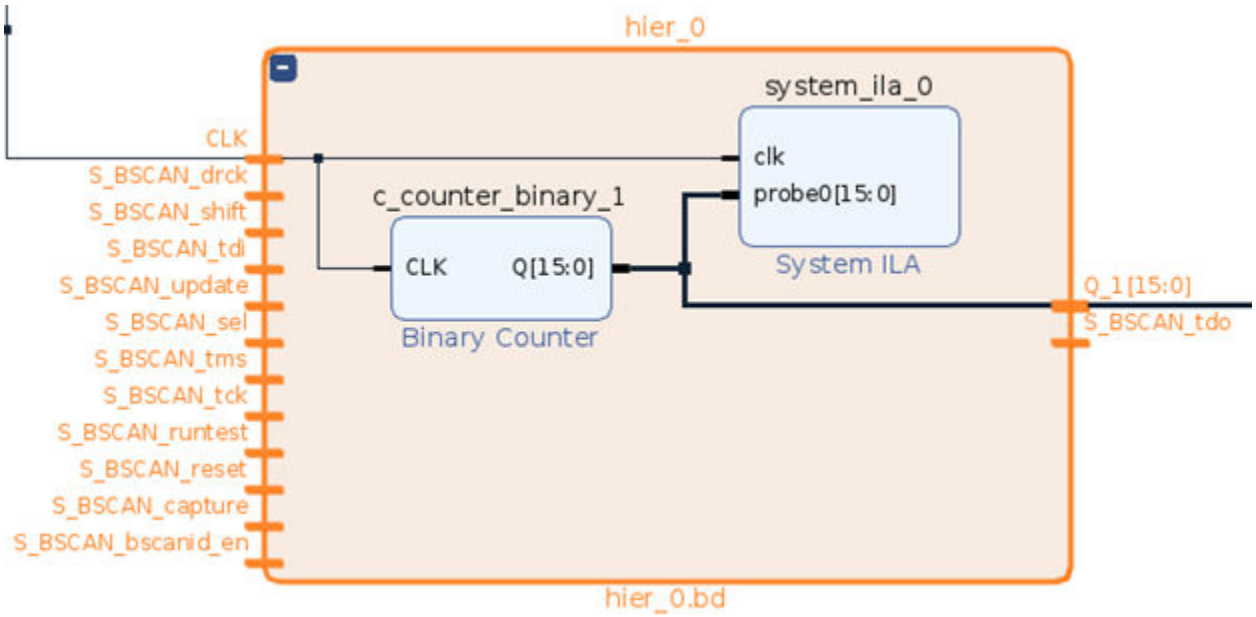
```
"xilinx.com:interface:bscan:1.0 S_BSCAN
runtest";

attribute DEBUG of my_runtest: signal is "true";

attribute X_INTERFACE_INFO of my_reset: signal is
"xilinx.com:interface:bscan:1.0 S_BSCAN
reset";

attribute DEBUG of my_reset: signal is "true";

attribute X_INTERFACE_INFO of my_capture: signal is
"xilinx.com:interface:bscan:1.0 S_BSCAN
capture";

attribute DEBUG of my_capture: signal is "true";

attribute X_INTERFACE_INFO of my_bscanid_en: signal is
"xilinx.com:interface:bscan:1.0
S_BSCAN bscanid_en";

attribute DEBUG of my_bscanid_en: signal is "true";
```

You can use this approach for IP integrator as well. Use the following Tcl commands to add the necessary ports to a block design in an IP integrator flow:

```
create_bd_port -dir I S_BSCAN_drck
create_bd_port -dir O S_BSCAN_tdo
create_bd_port -dir I S_BSCAN_shift
create_bd_port -dir I S_BSCAN_tdi
create_bd_port -dir I S_BSCAN_update
create_bd_port -dir I S_BSCAN_sel
create_bd_port -dir I S_BSCAN_tms
create_bd_port -dir I S_BSCAN_tck
create_bd_port -dir I S_BSCAN_runtest
create_bd_port -dir I S_BSCAN_reset
create_bd_port -dir I S_BSCAN_capture
create_bd_port -dir I S_BSCAN_bscanid_en
```

The following figure shows the IP integrator canvas with the appropriate connectivity.

Figure 206: **IP Integrator Canvas**



IMPORTANT! *There is currently one additional requirement for this solution. A greybox configuration cannot be the first one processed. The debug bridge must be established within an RM with debug cores to establish connectivity with the debug hub before moving to versions that do not contain debug cores.*

For an example of this core insertion as well as functionality within the Vivado Hardware Manager, see the *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947).

Chapter 10

Configuration Solutions for Versal Devices

Dynamic Function eXchange is managed by platform loader and manager (PLM) services running in the PMC. These services control the events needed before, during, and after dynamic reconfiguration of the NPI and CFI resources throughout the device. These events include the following:

- Controlling the isolation of the target region
- Unloading (and loading) of software drivers as appropriate for modified applications
- Delivery of programming images from any secondary boot interface
- Image authentication and integrity checking before the programming is done

TIP: *All primary and secondary boot modes can be used for partial device image delivery.*

For more information on this topic, see *Versal Adaptive SoC System Software Developers Guide* (UG1304) for detailed discussion on primary and secondary boot modes.

Partial PDI Creation and Details

Unlike AMD UltraScale+™ and prior architectures, bitstreams are not used to program AMD Versal™ devices. The Versal device PMC uses a proprietary boot and configuration file format called the programmable device image (PDI) to program and configure the Versal device. The PDI consists of headers, the PLM image, and design data image partitions to be loaded into the Versal device. The PDI also contains configuration data, ELF files, and NoC register settings. The PDI image is programmed through the PMC block by the Boot ROM and PLM. For more information on the PDI file format and generation, see the *Bootgen User Guide* (UG1283).

To generate partial PDI for Dynamic Function eXchange, use the `write_device_image` Tcl command. By default, `write_device_image` generates a full device PDI as well as a partial PDI for each RP found in the design. The two key options for `write_device_image` are:

- `-cell <arg>`, which creates only a partial PDI for the requested RP
- `-no_partial_pdi`, which creates only a full device PDI.

For more information on Tcl commands, see *Vivado Design Suite Tcl Command Reference Guide* (UG835). For more information on the structure and details of PDI, configuration data object (CDO) files and commands, and supported boot devices, refer *Boot and Configuration* chapter of *Versal Adaptive SoC System Software Developers Guide* (UG1304).

Supported Boot Modes

Any primary or secondary boot modes can be used to deliver partial images. Partial PDI can be fetched from boot devices, over Ethernet or USB, or from DDR memory controllers. They can also be passed over slave boot interfaces for JTAG, PCIe®, or SelectMAP use cases.

Table 30: **Primary and Secondary Boot Interfaces**

Configuration Mode	Type	Maximum Bandwidth	Authentication and Encryption
Primary			
JTAG	Slave	12.5 MB/s	No
QSPI	Master	150 MB/s	Yes
OSPI	Master	400 MB/s	Yes
SD	Master	100 MB/s	Yes
eMMC	Master	200 MB/s	Yes
SelectMap	Slave	800 MB/s	Yes
Secondary			
PCIe	Master	6.4 GB/s	Yes
DDR4	Slave	6.4 GB/s	Yes
PL (NoC)	Master	6.4 GB/s	Yes

Secure Boot Modes

Versal devices support secure boot capabilities. You can use encryption (AES-GCM) and authentication (RSA or ECDSA) features with DFX designs, within monolithic or SSI technology devices.

For more information on Secure Boot and the rules for implementing DFX with Secure Boot in Versal, refer to the *Versal Adaptive SoC Security Manual* (UG1508). This manual requires an active NDA to download from the Design Security Lounge.

Design Version Compatibility Checks

A critical detail of Dynamic Function eXchange (DFX) is ensuring that incoming partial images are compatible with the currently operating static design. Although `pr_verify` validates compatibility between a pair of routed design checkpoints, you must keep the programming images in sync after they are generated. Versal devices have safeguards in place to help with this effort.

In UltraScale+ and older devices, no compatibility checks are available as a dedicated silicon programming feature. Device IDs embedded in bitstreams ensure the correct target device is selected, but there is no equivalent *Design* ID to check compatibility of a partial bitstream to the currently loaded full design. If a partial bitstream is programmed over the incorrect static design, failures can occur. These failures can range from simple issues, such as the new RM not operating (because it does not connect to the static domain) or disruption of the static logic itself (due to static elements or routes not being maintained), to more critical issues, such as contention if static and dynamic resource usage overlaps. The latter is much less likely, but it highlights the need for designers to build solutions to ensure that incompatible images are never mixed.

Versal devices have safeguards in the form of unique identifiers (UID) that are checked by the PLM when partial programming images are delivered. These checks are enabled for DFX, tandem configuration, and Segmented Configuration solutions. Four 32-bit fields are embedded in the PDI as shown in the following table.

Table 31: **UIDs Checked by PLM**

UID Name	Description	Defined by	PDI Mapping
Node ID	Defines the configuration node in the PL or AI Engine	Vivado tools	id (0x18)
Unique ID	A unique hash value to identify the module	Vivado tools	unique_id (0x24)
Parent ID	A reference to the Unique ID of the module above the target module	Vivado tools	parent_unique_id (0x28)
Function ID	Identification of the function residing in the target module	User	function_id (0x2c)

Three of the four IDs are automatically generated by the Vivado tools, and you define the fourth ID known as the Function ID,. Following are details on each ID:

- **Node ID:** Fixed value that is incremented for each partition in the design. The static image represents the initial configuration of the device and is given an ID of 0x18700000 for the PL and 0x18800000 for the AI Engine. Each node (RM) included in the design has an incremented value starting with 0x18700002, as shown in the example below. A value of 0x18700001 is reserved for Tandem Configuration, representing the stage 2 design, or for Segmented Configuration, representing the PLD design. This value is useful in differentiating the partitions in the design.

- **Unique ID:** Hash value automatically generated by the Vivado tools. The value is deterministic, calculated by a number of factors within the design, so any change to a module's results (for example, code change, new synthesis or implementation options, design constraints, etc.) produces a new Unique ID. If the static design is recompiled such that prior versions of reconfigurable module partial bitstreams do not align, a new Unique ID is issued.
- **Parent ID:** Reference to the Unique ID of the module above the current one. The Parent ID of the static design is always zero (0x00000000), because it is the top level of any DFX design. The Parent ID of a reconfigurable module is the Unique ID of the static design it was compiled with. If Tandem PCIe is enabled, the parent ID of each RM is a UID of the stage 2 PDI, because this is the programming image that must be loaded before partial images can be used. Similarly, if Segmented Configuration is enabled, the parent ID of each RM is a UID of the PLD PDI.
- **Function ID:** User space to embed any custom unique identifier that can be parsed in the PDI header. You can apply a 32-bit value by setting a property on the reconfigurable module instance. This enables you to quickly differentiate functions contained in the programming image

***Note*:** The Function ID feature is not currently supported and remains at the default 0x00000000.

These unique identifiers are automatically stored in the full and partial PDI files generated by `write_device_image`. You can read these values using various methods, but the easiest method is to parse them using Bootgen. Call Bootgen with the `-read` option to see the details of the PDI contents. Find the four UID fields in the pl_cfi portion of the output.

```
bootgen -arch versal -read <pdi file>
```

Information seen in the full design PDI that contains the static partition:

```
...
--------------------------------------------------------------------------------
   IMAGE HEADER (pl_cfi)
--------------------------------------------------------------------------------
          pht_offset (0x00) : 0x000160c0       section_count (0x04) : 0x00000002
      mHdr_revoke_id (0x08) : 0x00000000          attributes (0x0c) : 0x00001800
                 name (0x10) : pl_cfi
                   id (0x18) : 0x18700000           unique_id (0x24) : 0x589cd7dd
    parent_unique_id (0x28) : 0x00000000         function_id (0x2c) : 0x00000000
   memcpy_address_lo (0x30) : 0x00000000   memcpy_address_hi (0x34) : 0x00000000
             checksum (0x3c) : 0x2b91d98a
 attribute list -
                    owner [plm]               memcpy [no]
                     load [now]              handoff [now]
   dependentPowerDomains [spd][pld]
```

Information seen in a partial PDI that represents a dynamic partition:

```
...
--------------------------------------------------------------------------------
   IMAGE HEADER (pl_cfi)
--------------------------------------------------------------------------------
          pht_offset (0x00) : 0x00000034       section_count (0x04) : 0x00000008
      mHdr_revoke_id (0x08) : 0x00000000          attributes (0x0c) : 0x00001800
                 name (0x10) : pl_cfi
                   id (0x18) : 0x18700002           unique_id (0x24) : 0xb61496d2
    parent_unique_id (0x28) : 0x589cd7dd         function_id (0x2c) : 0x00000000
   memcpy_address_lo (0x30) : 0x00000000   memcpy_address_hi (0x34) : 0x00000000
             checksum (0x3c) : 0x757ea33c
 attribute list -
                    owner [plm]               memcpy [no]
                     load [now]              handoff [now]
   dependentPowerDomains [spd][pld]
```

The Unique ID of the first output matches the Parent Unique ID of the second output. These PDI are from the same configuration and are therefore compatible.

This data can also be found via `cdoutil` by grepping for the string `ldr_set_image_info` in a full or partial PDI image. The four UID fields are always listed in order: Node ID, Unique ID, Parent Unique ID, Function ID. Full design PDI have the UID information for the static domain, which is clearly identified by the Parent ID (the third field set to 0), followed by all partials. Whereas partial PDI only shows UID information for that partial image.

```
> cdoutil design_1_wrapper.pdi | grep ldr_set_image_info

ldr_set_image_info 0x18700000 0x589cd7dd 0 0

ldr_set_image_info 0x18700002 0x86a292a3 0x589cd7dd 0

> cdoutil design_1_i_rp1_rp1rm2_inst_0_partial.pdi | grep
ldr_set_image_info

ldr_set_image_info 0x18700002 0xb61496d2 0x589cd7dd 0
```

During the load of a partial programming image, PLM examines these contents and confirms these identifiers match. If they do not, a failure is reported and the partial PDI is rejected before it is programmed, allowing you to take corrective action. Following is an example of the PLM log for a failing condition.

```
[4232.091]***********Boot PDI Load: Done***********

[4236.635]4200.941 ms: ROM Time

[4239.471]Total PLM Boot Time

[24261.805]SBI PDI Load: Started

[24261.894]Loading PDI from SBI

[24264.739]Monolithic/Master Device

[24267.946]6.046 ms: PDI initialization time

[24271.897]Image is not compatible with Parent Image

[24276.556]Error loading PL data:

CFU_ISR: 0x00000000, CFU_STATUS: 0x00002A8C

PMC ERR1: 0x00000000, PMC ERR2: 0x00000000

[24287.449]PLM Error Status: 0x03530000
```

Tandem Configuration and PCIe

PCI Express® is a plug-and-play protocol, that is, at power up the PCIe host enumerates the system, including assigning base addresses to the PCIe devices. Therefore, the PCIe interfaces must be ready when the host queries the interfaces, or the interfaces are not assigned a base address. The PCI Express specification states that PERST# can deassert 100 ms after the power good of the systems has occurred, and a PCI Express port must be ready to link train 20 ms after PERST# has deasserted. This is commonly referred to as the 100 ms boot time requirement, even though 120 ms is the goal.

Versal devices can meet this 120 ms link training requirement by using Tandem Configuration, a solution that splits the programming image into two stages. The first stage quickly configures the PCIe Endpoints so that the Endpoint is ready for link training within 120 ms. The second stage then configures the rest of the device. Two variants are supported:

- **Tandem PROM:** Loads both stages from a single programming image from a standard primary boot interface.
- **Tandem PCIe:** Loads the first stage from a primary boot interface. Then, the second stage is delivered via one of the CPM PCIe controllers.

Within the Versal adaptive SoC, the CPM consists of two PCIe controllers, DMA features, CCIX features, and network on chip (NoC) integration. The Versal Adaptive SoC CPM Mode for PCI Express enables direct access to the two high-performance, independently customizable PCIe controllers. You can specify whether to have one or both of these controllers enumerate within 120 ms using Tandem Configuration.

IMPORTANT! *Not every device has a CPM, so if 120 ms enumeration is a requirement for your system, be sure to select a device that can meet this goal.*

Tandem Configuration and Dynamic Function eXchange (DFX) are solutions for different phases of device operation. Tandem Configuration is only used at the initial power-up of the device or after a full device reconfiguration request to bring up the device in stages. DFX is used during normal operation to deliver programming images that modify a portion of the programmable logic while the rest of the device remains active. Versal devices support both technologies in the same design.

The CPM is a resource customized within the CIPS IP and must reside in the static domain of a DFX design. You must take care to keep any DFX Pblocks away from the PS-PL boundary where soft logic associated with the CPM is placed. Failure to do so might result in an unrouteable design.

When using DFX in Versal devices with CPM resources, you can use the DMA functionality to deliver partial PDI images from a PCIe host to the PMC for partial reconfiguration. You can manage DMA transfers (QDMA for CPM4 or CPM5, XDMA for CPM4 only) using drivers provided by AMD. The aperture size must be set so the DMA transfer acts on a keyhole. A configurable example design (CED) is available within the AMD Vivado™ tools, showing the details of the Tandem PCIe solution, including the use of the QDMA driver to deliver the stage 2 programming image. This CED also covers details on how to deliver partial images for DFX during normal operation of the design.

For more information on PCIe functionality, Tandem Configuration, and PDI delivery via CPM DMA, see the *Versal Adaptive SoC CPM DMA and Bridge Mode for PCI Express Product Guide* (PG347), which lists device support for Tandem features.

Reconfiguration via U-Boot

Partial PDI can also be delivered via U-Boot. Non-secure and secure images can be sent using the `versal loadpdi` command to a system that has been configured with the boot PDI and has at least PLM running on the target.

For further details and syntax examples, see the Loading PL and Partial PDI on Versal Platform Using U-Boot Wiki.

Partial PDI can be delivered from PCIe, DDR memory, or a primary boot device. For loading the partial PDI, the XilLoader CDO commands can be used using the IP integrator interface. XilFPGA provides the required APIs to load DFx PDIs from APU.

Reconfiguration via Linux

For managing partial PDI delivery via Linux, you can use the Linux FPGA Manager framework. You can call the fpgautil utility with the `-f Partial` option to load partial bitstreams through the sysfs interface. Non-secure and secure (via encryption and/or authentication) image delivery modes are supported.

Take care to offload and reload drivers associated with functions in the target dynamic region. You can build device tree overlays to modify the existing device tree to reflect changes in hardware after partial reconfiguration has occurred.

For details and syntax examples, see the Solution Versal PL Programming Wiki and the Versal Adaptive SoC Linux-Based Partial Image Delivery Tutorial available from the GitHub repository.

Programming Image Compression

By default, all full and partial PL programming images are compressed. Versal devices use a new compression algorithm that is a different approach than the prior architectures. Versal device programmable device image (PDI) compression reduces the configuration time in most cases.

Reduced configuration time and decreased configuration storage size requirements are observed when compression is used for the following interfaces for QSPI, OSPI, SD, eMMC, JTAG, and SelectMAP:

- Less than 2.56 GB/s (in the slowest speed grade)
- 3.2 GB/s (in medium and fast speed grades)

For this reason, compression is enabled by default.

***Note*:** The configuration time is affected when the Versal device PDI (`.pdi` format) is compressed and if the interface exceeds 2.56 GB/s (slow speed grades) or 3.2 GB/s (medium and fast speed grades).

The maximum bandwidth of the Versal device PMC configuration frame unit (CFU) is 5.12 GB/s (for slow speed grade) or 6.4 GB/s (for medium and fast speed grades). When an interface exceeds either 2.56 GB/s or 3.2 GB/s, the bandwidth will be throttled.

- **Use uncompressed image:** For high-speed interfaces that require the shortest programming time when images are delivered from DDR memory or through PCIe.
- **Use compression:** For high-speed interfaces if the primary goal is minimizing storage space.

To disable compression, apply this property prior to running `write_device_image`:

```
set_property BITSTREAM.GENERAL.COMPRESS FALSE [current_design]
```

***Note*:** Bandwidth is calculated based on the uncompressed data size. For example, if a full xcvc1902 PDI (medium speed grade) contains approximately 90 MB of CFI data then the time required to load this data is 90 MB/3.2 GB/s = 27.9 ms.

NoC Clock Gating Issue

Versal devices have a clock gating capability within the NoC to reduce power consumption due to unused clock buffers. This feature is not yet supported for certain DFX use cases.

- For DFX designs that contain a single RP that contains NoC resources, the feature is supported and it is ON by default.
- For DFX designs that contain two or more RP that contain NoC resources, this feature is automatically disabled.

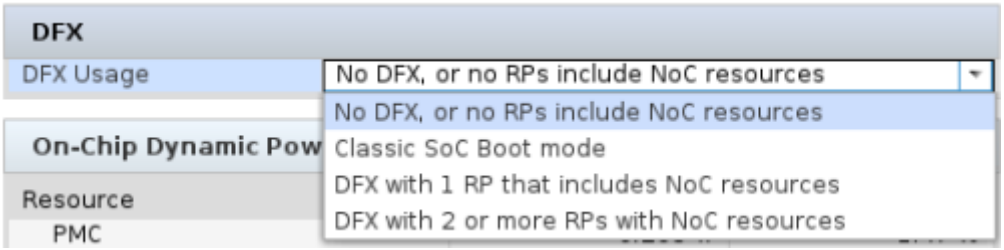
- For DFX designs that target devices that use stacked silicon interconnect (SSI) technology, the feature is automatically disabled.

Figure 207: **NoC Clock Gating Summary**

Device	1 RP with NoC	2+ RP with NoC
Versal Monolithic	NoC Clock Gating **ON**	NoC Clock Gating **OFF**
Versal SSI	NoC Clock Gating **OFF**	NoC Clock Gating **OFF**

When this feature is disabled, the partial PDI can function properly, but the ungated clock buffers consumes 37 mW per buffer, and this additional power needs to be accounted for in `Vcc_SoC` rail and power supply design. When calculating power consumption within the Power Design Manager, be sure to set the DFX operating conditions to match your use case for the most accurate results.

Figure 208: **NoC Clock DFX**



Segmented Configuration

Segmented Configuration is a solution that enables you to boot the processors in a device and access DDR memory before the programmable logic (PL) is configured. This allows DDR-based software, such as Linux, to boot first, followed by the PL, which you can configure later if needed via any primary or secondary boot device or through a DDR image store. The Segmented Configuration feature is designed to provide the Versal boot sequence with flexibility similar to AMD Zynq™ UltraScale+™ MPSoCs methods of configuring the PL . This feature is optional when targeting first-generation Versal devices, but it is always enabled for second-generation Versal devices, including AMD Versal™ AI Edge Series Gen 2 and AMD Versal™ Prime Series Gen 2.

This solution uses a standard Vivado tool flow through implementation, with the only additional annotation required being the identification of NoC path segments to include in the initial boot image. This identification occurs automatically after you set the project property that enables the feature. Programming image generation (`write_device_image`) automatically splits the programming images into two PDI files, which are stored and delivered separately. The entire PL is dynamic and can be completely reloaded while any operating system and DDR memory access remain active.

Segmented configuration is not DFX, but it can be used to meet similar goals. After your design is up and running, Segmented Configuration enables you to dynamically reload the entirety of the PL domain, similar to how you can reload a portion of the PL using DFX. When using Segmented Configuration, the dynamic region is determined by the silicon, so you do not have the ability to change the boundary between static and dynamic.

The capability within Segmented Configuration that pairs multiple PL images with a fixed boot image is referred to as *PL Reload*. The explicit DFX design flow is not required, and you design in separate projects. However, you must take specific steps to ensure the contents of the boot image are fixed so each new PL image can connect properly. This is done by locking and reusing the NoC solution from a golden design image. The concept is similar to DFX in that the PS boot image must remain fixed, with all details locked, including PS-PL boundary interface usage and the physical and addressing information for the NoC paths that cross between domains.

In general, supported use cases in Vivado are relatively narrow in scope. Different PL images are expected to be subtle variations of a base design, at least in terms of the boundary conditions. The initial *golden* design run must be a superset of all connectivity required for any PL variant that connects to this fixed boot image. All characteristics of the interface: pin usage, address apertures, NoC connectivity (number and types of connections), and so on, must not exceed this initial image. Subsets of connectivity are permitted (tie off unused ports) but you might introduce new PS-PL boundary connections.

For more information on Segmented Configuration, including design requirements and a tutorial walk-through, see the documentation available in the AMD Vivado GitHub repository.

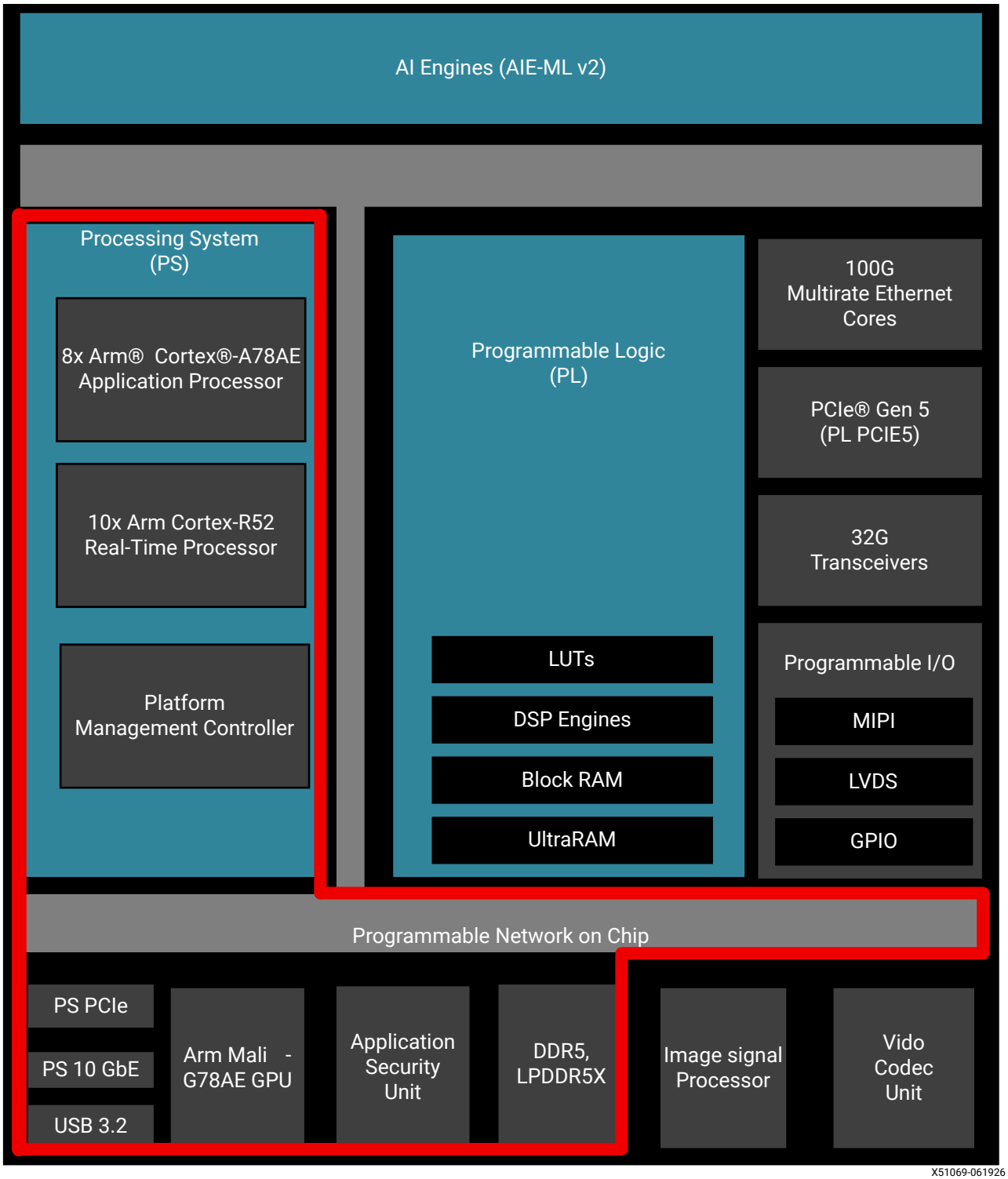
Segmented Configuration and DFX

The AMD Vivado™ 2026.1 release supports designs that use Segmented Configuration and Dynamic Function eXchange (DFX) together. This release supports AMD Versal™ AI Edge Series Gen 2 and Versal Prime Series Gen 2 devices listed in the supported devices table in the appendix. Future Vivado releases are expected to expand access to the remaining Versal devices that support Segmented Configuration and DFX individually.

Vivado Design Flow

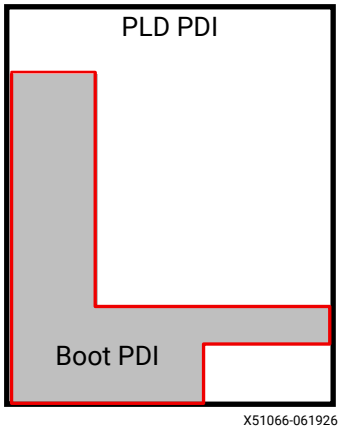
Within Vivado, this combined support is essentially the union of both solutions. Segmented Configuration splits the programming instructions into two files: `Boot.PDI` and `PLD.PDI`. DFX creates partial programming images that are subsets of the `PLD.PDI`. The result is a flexible programming environment similar to tandem with field updates or nested DFX, which AMD UltraScale+™ devices support. In the following simplified figure, the region enclosed in the red box represents the boot domain, and the remaining features represent the PLD domain.

Figure 209: **Boot Domain and PLD Domain**



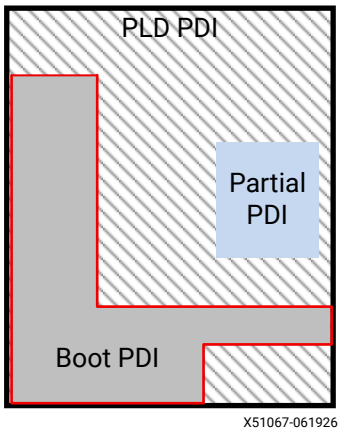
Segmented configuration splits the initial device programming between the PS and PL domains. The PL Reload feature lets you load different versions of the PLD for a given boot PDI. Different PLD images are typically generated from different Vivado projects and linked by a common NoC solution for consistency.

Figure 210: **Segmented Configuration with PL Reload**



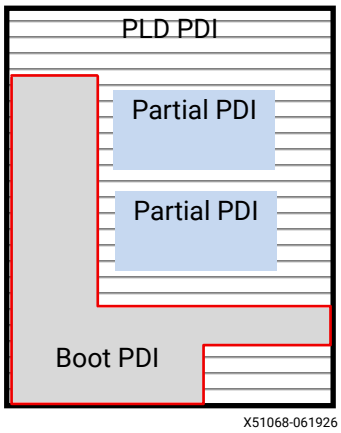
With DFX, after the Boot and PLD images that represent the complete static design are loaded, you can deliver a partial PDI on the fly to change functionality in the PL domain. You can create different partial PDIs with the DFX design flow in Vivado.

Figure 211: **Static and Partial PDI**



When you combine PL Reload and DFX, you can modify the DFX static design to create different floorplans for different design requirements. To deliver a partial PDI for a given floorplan or static PL design, you must first load the appropriate PLD image.

Figure 212: **Multiple Floorplans with Partial PDI**



Design Flows for Segmented Configuration + DFX

Within a single project, the design flow from design entry through implementation and device programming follows the standard DFX flow for Versal devices. Avoid one scenario: do not include resources in a reconfigurable partition (RP) Pblock if those resources are also in the boot partition. This conflict is most likely to occur at the DDRMC, which can appear in both an `initial_boot` path and an RP Pblock.

Device image generation in a single project combines both solutions. The `write_device_image` command, or **Generate Device Image** in project mode, creates all required programmable device images (PDIs) for both solutions. These PDIs include boot and PLD PDIs for Segmented Configuration, and one partial PDI for each RP for DFX. The flow supports abstract shells. PDI generation from an abstract run can generate only one partial PDI for that partition. You can use project flows with IP-based block design containers or RTL-based partition definitions. You can also use non-project flows with modular NoC.

When you use the PL Reload feature of Segmented Configuration, you can dynamically reconfigure the entire PL domain while the processing domain remains active. You can also dynamically reconfigure part of the PL domain while the rest of the PL, and the processing domain, remain active. You are responsible for pausing activity in, and communication to and from, the part of the device being reconfigured. For details on logical decoupling, refer to Logical Decoupling.

Design Compatibility Checks in Vivado Software

The Tcl command `pr_verify` confirms compatibility for both Segmented Configuration PL reload and DFX use cases. The command now supports the combined use case.

The command compares both Segmented Configuration and DFX rules by default when both features are enabled. In a single Vivado DFX project that uses parent and child runs, this behavior does not change. In a single parent `impl_1` run, the locked DFX static design requires the boot partition for Segmented Configuration to remain unchanged. As a result, the PL reload checks pass.

The `-segcfg_only` switch supports comparisons across different projects for the PL Reload use case. When the PL implementations differ, the static PL domain also differs. In this case, the DFX checks fail. This switch instructs `pr_verify` to compare the two designs only for PL Reload compatibility.

For PL Reload, you must run `pr_verify` manually because no project mode includes built-in compatibility checking. Follow the PL Reload flow to associate projects by sharing boot-domain IP and the NoC solution file (`.ncr`).

For PL reload, list the golden configuration first, and then list the new design configuration. The golden version must contain the superset of PS/PL connectivity. The `boot.pdi` from the golden version is the image that the primary boot device loads.

To compare two routed design checkpoint (DCP) files from the same project with locked static logic for DFX matching, use the following:

```
pr_verify -initial parent_routed.dcp -additional child_routed.dcp
```

This example checks DFX compatibility.

To compare two routed DCP files from different projects with a shared NoC solution file for PL Reload matching, use the following:

```
pr_verify -segcfg_only -initial golden_routed.dcp -additional
secondary_routed.dcp
```

This example checks Segmented Configuration compatibility.

Design Compatibility Checks in Versal Hardware

In the previous section, 32-bit unique identifiers (UIDs) are embedded in each PDI. The PLM uses these UIDs to validate incoming programming images. These checks apply to both Segmented Configuration and DFX. PLM validates each image delivered to the PMC against the currently loaded design.

Node IDs are assigned as follows:

- The segmented boot PDI always uses node ID `0x18700000`
- The segmented PLD PDI always uses node ID `0x18700001`

- DFX partial PDIs start at node ID `0x18700002` and increment for each additional RP in the design

When both Segmented Configuration and DFX are enabled in a design:

- The segmented boot PDI has a UID generated by Vivado and a parent ID of `0`
- The segmented PLD PDI has a UID generated by Vivado and a parent ID that matches the boot PDI UID
- Each DFX partial PDI has a UID generated by Vivado and a parent ID that matches the PLD PDI UID

If the parent ID of an incoming PDI does not match the UID of the resident node above it, PLM reports a failure, and it does not load the PDI.

Chapter 11

Known Issues and Limitations

This is a list of issues that might be encountered when using Dynamic Function eXchange (DFX) in the current AMD Vivado™ Design Suite release. If you encounter any of these issues, or discover any others, contact AMD Support and send an example design that shows the issue. These test cases are very helpful to improve the overall solution.

Report to AMD all cases of fatal or internal errors, incomplete routing (partial antennas), or other rule violations that prevent place and route, `pr_verify`, and `write_bitstream` from succeeding. Including a design showing the failure is critical for proper analysis and implementation of fixes.

Known Issues

- Engineering silicon (ES) for AMD UltraScale™ or AMD UltraScale+™ devices does not officially support Dynamic Function eXchange. To investigate the capabilities of DFX on ES devices, contact AMD support for advice.
- Do not drive multiple outputs of a single RM with the same source. Each output of an RM must have a unique driver.
- When using AMD Virtex™ UltraScale+™ VU29P devices, connections between the IBUFDS_GTM and GTM_DUAL sites might be unroutable if the placer does not place them on the same SLR and the same side of the device. You might encounter route_design Route 35-7 in this case. If this occurs, you must LOC both the IBUFDS_GTM and GTM_DUAL instances to appropriate locations in the same SLR on the same side of the device.
- AMD Versal™ DFX designs targeting SSI devices with reconfigurable partitions spanning multiple SLRs compiled in Vivado versions prior to 2025.1: glitching of static routes crossing these SLR boundaries can occur during reconfiguration. These designs should be reimplemented using Advanced Flow (in Vivado 2024.2 or newer) and all programming images (full and partial PDI) must be regenerated using Vivado 2025.1 or newer. The failure occurrence possibility is incredibly low but non-zero. For more information, see Answer Record 000036769.
- Certain AMD Versal™ DFX designs compiled in Vivado versions prior to 2025.1 might experience NoC Compiler errors if the locked static design checkpoint is brought to Vivado 2025.1 or newer to implement new Reconfigurable Modules. Changes in the NoC Compiler affect designs with very specific conditions:

- Two or more Reconfigurable Partitions
- Reuse of pre-2025.1 post-synthesis or locked static checkpoints
- NoC streaming endpoints with N by M connectivity

These designs might encounter NoC DRC NOCCHK-1 or NoC compilation error Ipconfig75-4217. The issue can be avoided by resynthesizing and implementing the static design to generate results with the updated NoC Compiler (version 2025.1 or newer). For more information, see Answer Record 000039002.

- If the initial configuration of a 7 series SSI device (7V2000T, 7VX1140T) is done through an SPI interface, partial bitstreams cannot be delivered to the master (or any) ICAP; they must be delivered to an external port, such as JTAG. If the initial configuration is done through any other configuration port, the master ICAP can be used as the delivery port for partial bitstreams. Contact AMD support for a workaround.

Known Limitations

Certain features are not yet developed or supported in the current release. These include:

- When selecting Pblock ranges to define the size and shape of the RP, do not use the CLOCKREGION resource type for 7 series or Zynq 7000 designs. Pblock ranges must only include types SLICE, RAMB18, RAMB36, and DSP48 resource types.
- Do not use Vivado Debug core insertion features within RPs when working with 7 series, UltraScale, or UltraScale+ devices. This flow inserts the debug hub, which includes BSCAN primitive, which is not permitted inside reconfigurable bitstreams. Vivado Debug cores must be instantiated or included within IP for these architectures, then the Debug Hub can be inferred as described in Using Vivado Debug Cores.
- The Soft Error Mitigation (SEM) IP core is supported with DFX in monolithic devices. For UltraScale devices, the SEM IP core is not supported when using Dynamic Function eXchange on SSI devices. For UltraScale+ devices, the SEM IP core is supported when using DFX on SSI devices. For more information on using the SEM IP in DFX designs, see *Demonstration of Soft Error Mitigation IP and Partial Reconfiguration Capability on Monolithic Devices* (XAPP1261).
- The STARTUP primitive does not support loading of partial bitstreams for 7 series and UltraScale devices, because its clock will stop after a partial bitstream enters the configuration engine. IP, such as the AXI SPI IP or the AXI EMC IP, should not be configured to use the STARTUP primitive to clock or deliver partial bitstreams from external flash. For these architectures, partial bitstreams may be stored in BPI or SPI flash, but they must be moved to DDR memory or another location before being shifted into the ICAP.
- Two use cases regarding encryption will not be supported when using new features within UltraScale and UltraScale+ devices:

1. If RSA authentication is selected for the initial configuration, then encrypted partial reconfiguration is not supported. RSA authentication is not supported on FPGAs for partial bitstreams.
2. If the initial configuration bitstream uses an obfuscated AES-256 key stored in either the eFUSE or BBRAM, then any encrypted partial bitstreams must use the same obfuscated key. Encrypted partial bitstreams using a different key than the initial bitstream is not supported.

   In either of these two cases, an unencrypted partial bitstream can be delivered to the ICAP to partially reconfigure the device.

- Bitstream compression and per-frame CRC checks cannot be enabled at the same time for a partial bitstream for 7 series, UltraScale, or UltraScale+ devices.
- The `update_design` command in general permits multiple targets for the `-cells` switch. However, when using this command for DFX designs (for use with `-black_box` or `-buffer_ports`), specify one cell (RP) at a time. Performing these actions on more than RP requires multiple calls to `update_design`.
- Cascaded global clocking buffers across RM boundary is not a supported use case and is not guaranteed to be successfully routed. If cascaded BUFS are unavoidable in the design, it is recommended to keep them both, either in static or RP.
- All RP ports must be strictly input or output in direction. Ports of type inout are not supported. Partition pins are established on boundary ports by selecting routing points or resource pins that are strictly unidirectional, so changing the direction at these points is physically impossible.

Related Information

Using Vivado Debug Cores

Chapter 12

Hierarchical Design Flows

The Dynamic Function eXchange (DFX) flow is an ideal hierarchical design tool for a top-down in-context use case. The ability to implement the static portion of the design, meet timing, and then reuse those results meets the needs of other hierarchical design (HD) flows.

Platform Reuse takes advantage of the tool's ability to separate out the top-level (Platform) from the lower level partitions. Platform Reuse can be used for a number of use cases including:

- Reduced design cycle and timing closure. By closing timing on the top-level, interface logic that changes once (such as memory controllers, networking IP, high speed interconnect) and then locking down and reusing the placed/routed result for the Platform which partition logic is being developed.
- Platform delivery to end users. In this case a Platform can be developed by one user, and then fully placed, routed, and locked DCP (with black boxes for partitions) and be delivered to another user for development of customer partition logic.

Both of these flows use the Dynamic Function eXchange flow, to implement the initial design, carve out the partitions, and lock down the Static/Platform logic. All features of DFX (like greybox support) can be used, and all requirements of the DFX flow must be followed. From the tool's perspective, there is no difference between the flows, and all DFX DRCs will be applied.

Because partial bit files are not required for this flow, AMD recommends using the `-no_partial_bitfile` switch of the `write_bitstream` command to avoid producing partial bitstreams.

Appendix A

Supported Devices

The following table shows the devices supported for the current release.

Table 32: **Supported Devices**

Device	Variants[1]
AMD Artix™ 7 Devices	
XC7A15T	A, L
XC7A35T	A, L
XC7A50T	A, L, Q
XC7A75T	A, L
XC7A100T	A, L, Q
XC7A200T	L, Q
AMD Kintex™ 7 Devices	
XC7K70T	L
XC7K160T	A, L
XC7K325T	L, Q, QL
XC7K355T	L
XC7K410T	L, Q, QL
XC7K420T	L
XC7K480T	L
AMD Virtex™ 7 Devices	
XC7V585T	Q
XC7V2000T	
XC7VX330T	Q
XC7VX415T	
XC7VX485T	Q
XC7VX550T	
XC7VX690T	Q
XC7VX980T	Q
XC7VX1140T	
XC7VH580T	
XC7VH870T	
AMD Zynq™ 7000 Devices	
XC7Z007S	

Table 32: **Supported Devices** *(cont'd)*

Device	Variants[1]
XC7Z010	A
XC7Z012S	
XC7Z014S	
XC7Z015	
XC7Z020	A, Q
XC7Z030	A, Q
XC7Z035	
XC7Z045	Q
XC7Z100	Q
AMD Kintex™ UltraScale™ Devices	
XCKU025	
XCKU035	
XCKU040	Q
XCKU060	Q, QR
XCKU085	
XCKU095	Q
XCKU115	Q
AMD Virtex™ UltraScale™ Devices	
XCVU065	
XCVU080	
XCVU095	
XCVU125	
XCVU160	
XCVU190	
XCVU440	
AMD Spartan™ UltraScale+™ Devices	
XCSU10P	
XCSU25P	
XCSU35P	A
XCSU45P	
XCSU60P	
XCSU150P	
XCSU200P	
AMD Artix™ UltraScale+™ Devices	
XCAU7P	A
XCAU10P	A
XCAU15P	A
XCAU20P	
XCAU25P	

Table 32: **Supported Devices** *(cont'd)*

Device	Variants[1]
AMD Kintex™ UltraScale+™ Devices	
XCKU3P	
XCKU5P	Q
XCKU9P	
XCKU11P	
XCKU13P	
XCKU15P	Q
XCKU19P	
AMD Virtex™ UltraScale+™ Devices	
XCVU2P	
XCVU3P	Q
XCVU5P	
XCVU7P	Q
XCVU9P	Q
XCVU11P	Q
XCVU13P	Q
XCVU19P	
XCVU23P	
XCVU27P	
XCVU29P	
XCVU31P	
XCVU33P	
XCVU35P	
XCVU37P	Q
XCVU45P	
XCVU47P	
XCVU57P	
AMD Zynq™ UltraScale+™ MPSoCs	
XCZU1CG	
XCZU1EG	A
XCZU2CG	
XCZU2EG	A
XCZU3CG	
XCZU3EG	A, Q
XCZU3TCG	
XCZU3TEG	A
XCZU4CG	
XCZU4EG	Q
XCZU4EV	A

Table 32: **Supported Devices** *(cont'd)*

Device	Variants[1]
XCZU5CG	
XCZU5EG	
XCZU5EV	A, Q
XCZU6CG	
XCZU6EG	
XCZU7CG	
XCZU7EG	
XCZU7EV	A, Q
XCZU9EG	
XCZU9EG	Q
XCZU11EG	A, Q
XCZU15EG	Q
XCZU17EG	
XCZU19EG	Q
AMD Zynq™ UltraScale+™ RFSoCs	
XCZU21DR	Q
XCZU25DR	
XCZU27DR	
XCZU28DR	Q
XCZU29DR	Q
XCZU39DR	
XCZU42DR	
XCZU43DR	
XCZU46DR	
XCZU47DR	
XCZU48DR	Q
XCZU49DR	Q
XCZU63DR	
XCZU64DR	
XCZU65DR	Q
XCZU67DR	Q
AMD Versal™ AI Core Series	
XCVC1502	
XCVC1702	Q
XCVC1802	
XCVC1902	Q, QR
XCVC2602	
XCVC2802	

Table 32: **Supported Devices** *(cont'd)*

Device	Variants[1]
AMD Versal™ AI Edge Series	
XCVE1752	A
XCVE2002	A
XCVE2102	A, Q
XCVE2202	A
XCVE2302	A, Q, QR
XCVC2602	A
XCVC2802	A
AMD Versal™ AI Edge Series Gen 2	
XC2VE3358	
XC2VE3504	
XC2VE3558	
XC2VE3804	
XC2VE3858	
AMD Versal™ Prime Series	
XCVM1102	Q
XCVM1302	
XCVM1402	Q
XCVM1502	Q
XCVM1802	Q
XCVM2202	
XCVM2302	
XCVM2502	
XCVM2902	
AMD Versal™ Prime Series Gen 2	
XC2VM3558	
XC2VM3858	
AMD Versal™ Premium Series	
XCVP1002	
XCVP1052	Q
XCVP1102	
XCVP1202	Q
XCVP1402	Q
XCVP1502	Q
XCVP1552	
XCVP1702	Q
XCVP1802	
XCVP1902	
XCVP2502	Q

Table 32: **Supported Devices** *(cont'd)*

Device	Variants[1]
XCVP2802	
AMD Versal™ HBM Series	
XCVH1522	
XCVH1542	
XCVH1582	
XCVH1742	
XCVH1782	

Notes:

1. Variants in addition to commercial grade (XC):
   - L = Low Power
   - A = Automotive
   - Q = Defense Grade
   - QL = Defense Grade, Low Power
   - QR = Defense Grade, Radiation Tolerant

***Note*:** All speed grades and temperature grades are supported.

Early access devices can be license-gated, and for any early access and/or engineering silicon (ES) devices, device programming information is parameter-gated.

If a device is early access and/or engineering silicon, the error message from the Vivado tools is the same. When running `write_bitstream` or `write_device_image`, the following error is reported:

```
ERROR: [Vivado 12-4164] Dynamic Function eXchange partial bitstream
generation is not supported for part <device>.
```

Contact AMD for information regarding access to pre-production device support.

AMD

Appendix B

Additional Resources and Legal Notices

Finding Additional Documentation

Technical Information Portal

The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to https://docs.amd.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav, do the following:

- From the AMD Vivado™ IDE, select **Help → Documentation and Tutorials**.
- On Windows, click the **Start** button and select **AMD Adaptive SoC and FPGA Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

***Note*:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

AMD 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, do the following:

- In DocNav, click the **Design Hubs View** tab.
- Go to the Design Hubs web page.

Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Support.

References

1. *Versal Adaptive SoC Technical Reference Manual* (AM011)
2. *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054)
3. *Virtex 7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023)
4. *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156)
5. *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194)
6. *DMA/Bridge Subsystem for PCI Express Product Guide* (PG195)
7. *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)
8. *Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313)
9. *Versal Adaptive SoC CPM Mode for PCI Express Product Guide* (PG346)
10. *Versal Adaptive SoC CPM DMA and Bridge Mode for PCI Express Product Guide* (PG347)
11. *Dynamic Function eXchange Decoupler IP LogiCORE IP Product Guide* (PG375)
12. *Dynamic Function eXchange Bitstream Monitor IP LogiCORE IP Product Guide* (PG376)
13. *Dynamic Function eXchange AXI Shutdown Manager IP LogiCORE IP Product Guide* (PG377)
14. *7 Series FPGAs Configuration User Guide* (UG470)
15. *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476)
16. *7 Series FPGAs GTP Transceivers User Guide* (UG482)
17. *UltraScale Architecture Configuration User Guide* (UG570)
18. *UltraScale Architecture Clocking Resources User Guide* (UG572)
19. *UltraScale Architecture Configurable Logic Block User Guide* (UG574)
20. *UltraScale Architecture GTH Transceivers User Guide* (UG576)
21. *UltraScale Architecture GTY Transceivers User Guide* (UG578)
22. *Zynq 7000 SoC Technical Reference Manual* (UG585)
23. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
24. *Spartan UltraScale+ FPGAs Configuration User Guide* (UG860)

25. *Vivado Design Suite User Guide: Synthesis* (UG901)
26. *Vivado Design Suite User Guide: Using Constraints* (UG903)
27. *Vivado Design Suite User Guide: Implementation* (UG904)
28. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906)
29. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
30. *Vivado Design Suite Properties Reference Guide* (UG912)
31. *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947)
32. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994)
33. *UltraFast Design Methodology Guide for FPGAs and SoCs* (UG949)
34. *Zynq UltraScale+ Device Technical Reference Manual* (UG1085)
35. *Zynq UltraScale+ Device Register Reference* (UG1087)
36. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
37. *Bootgen User Guide* (UG1283)
38. *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide* (UG1387)
39. *Versal Adaptive SoC System Integration and Validation Methodology Guide* (UG1388)
40. *Versal Adaptive SoC System Software Developers Guide* (UG1304)
41. *Versal Adaptive SoC Hardware, IP, and Platform Development Methodology Guide* (UG1387)
42. *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393)
43. *PRC/EPRC: Data Integrity and Security Controller for Partial Reconfiguration Application Note* (XAPP887)
44. *MMCM and PLL Dynamic Reconfiguration Application Note* (XAPP888)
45. *Local Partial Reconfiguration Using Embedded Processing for 3D ICs* (XAPP1099)
46. *Partial Reconfiguration of a Hardware Accelerator with Vivado Design Suite for Zynq 7000 AP SoC Processor* (XAPP1231)
47. *Bitstream Identification with USR_ACCESS using the Vivado Design Suite* (XAPP1232)
48. *Demonstration of Soft Error Mitigation IP and Partial Reconfiguration Capability on Monolithic Devices* (XAPP1261)
49. *Isolation Design Flow for UltraScale+ FPGAs and Zynq UltraScale+ MPSoCs* (XAPP1335)
50. *Fast Partial Reconfiguration Over PCI Express Application Note* (XAPP1338)
51. Vivado Design Suite Documentation
52. Loading PL and Partial PDI on Versal Platform Using U-Boot Wiki
53. Solution Versal PL Programming Wiki

Training Resources

AMD provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related training resources:

1. Vivado Design Suite QuickTake Video Tutorials
2. Vivado Design Suite QuickTake Video: Partial Reconfiguration in Vivado
3. Vivado Design Suite QuickTake Video: Partial Reconfiguration for UltraScale
4. Vivado Design Suite QuickTake Video: Partial Reconfiguration for UltraScale+
5. Partial Reconfiguration Flow on Zynq using Vivado
6. Designing with Dynamic Function eXchange (DFX) Using the Vivado Design Suite (FPGA-DFX)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
07/01/2026 Version 2026.1	
Chapter 1: Introduction	Updated the section.
Dynamic Function eXchange Commands	Updated the section.
Apply Reset After Reconfiguration	Updated the section.
Supported Boot Modes	Removed x32 from SelectMap in the table.
• Segmented Configuration and DFX • Vivado Design Flow • Design Flows for Segmented Configuration + DFX • Design Compatibility Checks in Vivado Software • Design Compatibility Checks in Versal Hardware • ELF Association Within DFX • Static Routing Across SLR Boundaries Using USER_SLL_GUIDANCE	Created new topics.
Configuration Debugging	Added a note.
Logical Decoupling at the PS-PL Interface	Updated the section.
Tandem Configuration and Versal Devices	Updated the section.
Appendix A: Supported Devices	Updated the section.
Use of Divided Outputs in MBUFGCE Primitives for Boundary Clock Nets	• Updated the title from Divided Outputs on MBUFGCE Primitives Not Allowed for Boundary Clock Nets. • Updated the entire section.

Please Read: Important Legal Notices

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes. THIS INFORMATION IS PROVIDED "AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

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 2012-2026 Advanced Micro Devices, Inc. AMD, the AMD Arrow logo, Artix, Kintex, Spartan, UltraScale, UltraScale+, Versal, Virtex, Vitis, Vivado, Zynq, and combinations thereof are trademarks of Advanced Micro Devices, Inc. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the US and/or elsewhere. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.