Vivado Design Suite User Guide

Dynamic Function eXchange

UG909 (v2021.1) June 30, 2021

XILINX®

Table of Contents

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
06/30/2021 Version 2021.1	
Abstract Shell for Dynamic Function eXchange	Added Abstract Shell and Nested DFX section.
Chapter 4: Vivado Project Flow	Added following sections: • IP Integrator Using Block Design Containers and Running the Dynamic Function eXchange Wizard. • Adding Reconfigurable Module Constraints in RTL Project flow in the Vivado IDE.
Added new chapters	• Chapter 8: Design Considerations and Guidelines for Versal Devices. • Chapter 10: Configuration Solutions for Versal Devices.
Chapter 13: Additional Resources and Legal Notices	Added List of Supported Devices.

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 that is comprised of many parts. These elements include the Xilinx® silicon ability to be dynamically reconfigured, the Vivado® software flow for compiling designs from RTL to bitstream, and the complementary features such as IP. In this release, you will see a mix of DFX and Partial Reconfiguration (PR) terminology, with DFX representing the overall solution and PR representing a component technology piece of that solution.

Complementary documentation, such as application notes, white papers, and videos, will not be recaptured with DFX terminology, but all new documentation from 2020 on will show the DFX terms.

The content of this guide includes the following:

- Description of Dynamic Function eXchange as implemented in the Vivado® Design Suite
- Assumption of familiarity with FPGA design software, particularly Vivado Design Suite
- Updates specific to the Vivado Design Suite Release 2021.1. This release supports Dynamic Function eXchange for the products listed below. For a complete list of supported devices, see List of Supported Devices.
  - 7 series Devices
    - Nearly all Virtex®-7, Kintex®-7, Artix®-7, and Zynq®-7000 SoC devices.

      Note: Spartan-7 devices, as well as Artix-7 A12T and 7A25T, are not supported.
  - UltraScale™ Devices
    - Place and route, as well as bitstream generation is enabled for all production devices.

      Note: Access to the VU440 is not restricted in this release. However, expect memory usage to be higher for this device than all others (potentially exceeding 64 MB).
    - Bitstream generation is disabled by default for ES2 devices, but place and route can still be performed.

- UltraScale+ Devices
  - Place and route, as well as bitstream generation, is enabled for all production devices, including all Zynq UltraScale+ RFSoCs, the Virtex UltraScale+ VU57P, and the Kintex® UltraScale+™ KU19P.
  - Place and route is enabled for many engineering silicon (ES1, ES2) versions of UltraScale+ devices. Bitstream generation is disabled by default for these devices.
- Versal Devices
  - DFX support for Versal devices is production for three devices in this release: the Versal AI Core VC1902, VC1802, and the Versal Prime VM1802. Also the Versal Premium VP1202 is available for ES1 silicon.

Navigating Content by Design Process

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

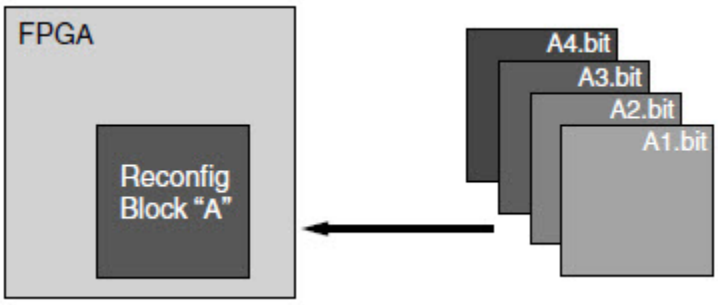
- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the Vivado® timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document 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 Xilinx Devices
  - Chapter 6: Design Considerations and Guidelines for 7 Series and Zynq Devices
  - Chapter 7: Design Considerations and Guidelines for UltraScale and UltraScale+ 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

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 gray area of the FPGA block represents static logic and the block portion 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.

Bottom-Up Synthesis

Block Design Containers (BDC) 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 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 Xilinx 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 UltraScale™ device. This entry point can be enabled when configuring the Xilinx 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 Xilinx 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 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

- Dynamic Function eXchange requires the use of Vivado 2013.3 or newer.
  - Partial Reconfiguration (PR) is supported in the ISE Design Suite as well. Use the ISE Design Suite for PR only with Virtex-6, Virtex-5 and Virtex-4 devices. See the *Partial Reconfiguration User Guide (v14.5)* (UG702) for more information.

- 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. Xilinx 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.
- The Vivado Design Suite includes support for the Dynamic Function eXchange Controller IP. This customizable IP manages the core tasks for partial reconfiguring any Xilinx device. 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 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. 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 Xilinx Isolation Design Flow (IDF) for Zynq MPSoC devices. For more information on solution details, please consult *Isolation Design Flow for UltraScale+ FPGAs and Zynq UltraScale+ MPSoCs* (XAPP1335).

Design Performance

Performance metrics vary from design to design, and the best results are achieved if you follow the Hierarchical Design techniques suggested in Chapter 12: Hierarchical Design Flows. You can find additional design recommendations in the *UltraFast Design Methodology Guide for Xilinx 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.

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 (e.g. spurious write to memory).
  - The Vivado Design Suite includes the Partial Reconfiguration Decoupler IP. This IP allows users to easily insert MUXes to efficiently decouple AXI4-Lite, AXI4-Stream, and custom interfaces. More information on the PR Decoupler IP is available on the Xilinx 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, Xilinx 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 Xilinx 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.

Dynamic Function eXchange Licensing

Dynamic Function eXchange is available as a feature within the Vivado Design Suite. Starting with Vivado 2019.1, no specific license code is necessary to use this feature for any edition of Vivado.

For older versions of Vivado, a Partial Reconfiguration license is included with every System Edition and Design Edition seat, and is available for purchase for Standard Edition seats.

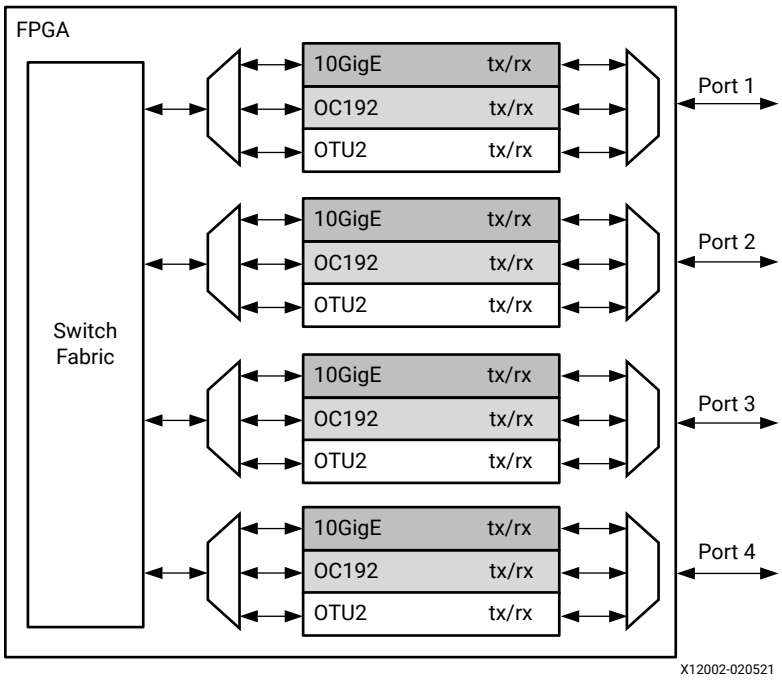
Common Applications

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

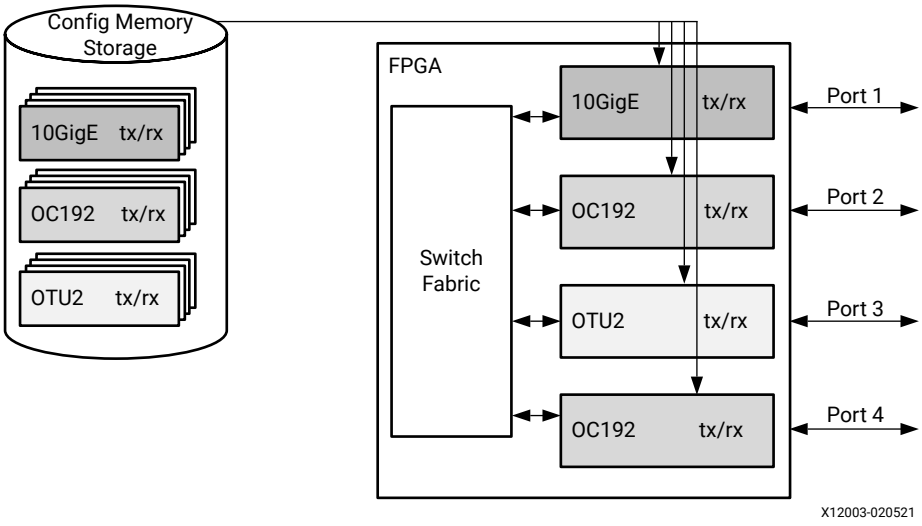
Dynamic Function eXchange optimizes traditional FPGA applications by reducing size, weight, power, and cost. Time-independent functions can be identified, isolated, and implemented as reconfigurable modules (RM) and swapped in and out of a single device as needed. A typical example is a 40G OTN muxponder 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 Networked Multiport Interface.

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 Networked Multiport Interface. 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 Networked Multiport Interface 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 Networked Multiport Interface 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.

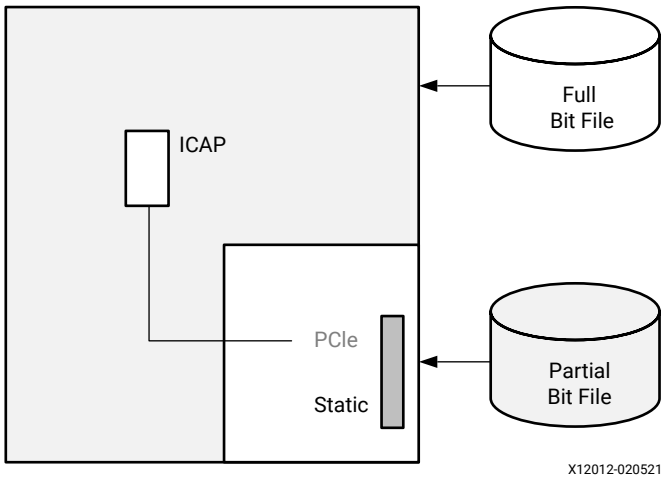
Configuration by Means of Standard Bus Interface

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 Configuration by Means of Standard Bus Interface. An example of fast configuration over PCIe is shown in *Fast Partial Reconfiguration Over PCI Express Application Note (v1.0)* (XAPP1338), with an example targeting UltraScale+™ 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 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.

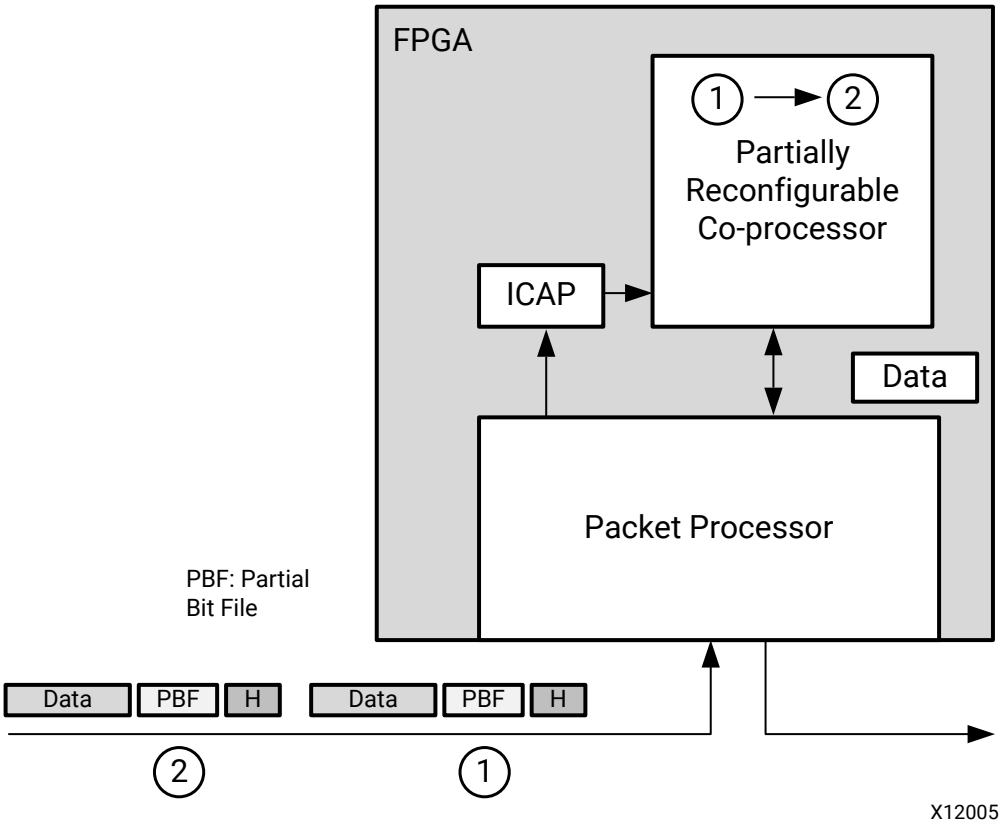
Tandem Configuration is designed to be a specific solution for a specific goal: fast configuration of a PCIe endpoint to meet enumeration requirements. For more information, see the following manuals:

- *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 Dynamically Reconfigurable Packet Processor, 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**

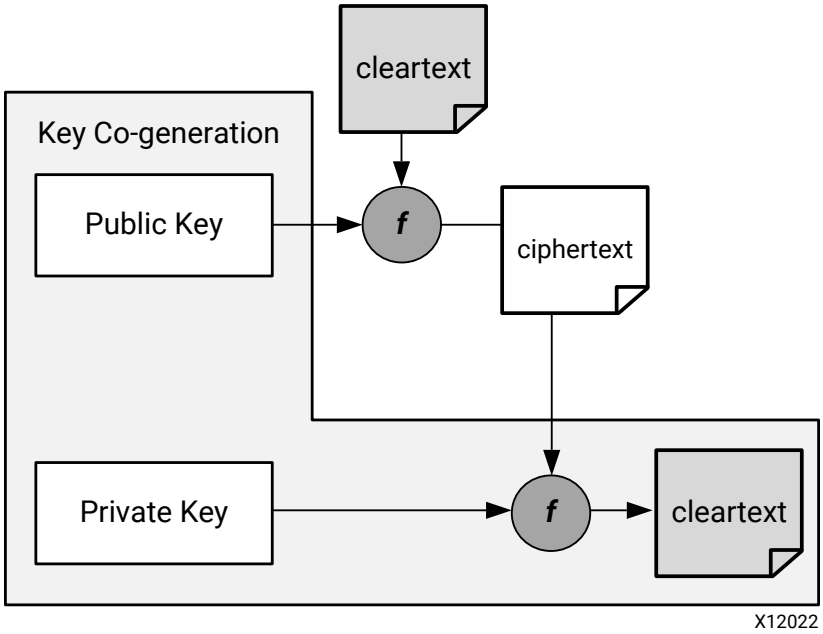


Asymmetric Key Encryption

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 Asymmetric Key Encryption, 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.

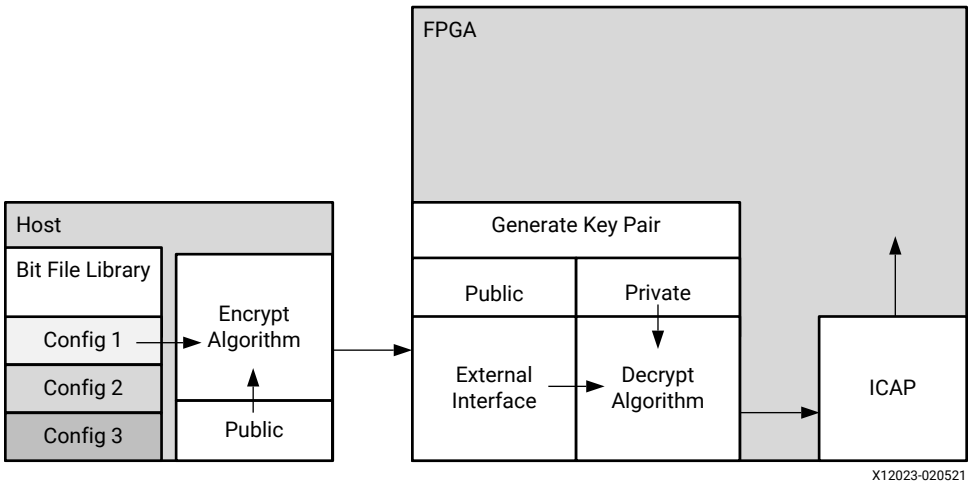
Figure 6: **Asymmetric Key Encryption**



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 Asymmetric Key Encryption.

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 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.

Dynamic Function eXchange Commands

The DFX flows are supported through the non-project batch/Tcl interface (no project based commands), as well as within an RTL-based project flow. Example scripts for the non-project flow are provided in the *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947), along with step-by-step instructions for setting up the flows. See that Tutorial for more information.

Even with the introduction of the Dynamic Function eXchange terminology, the underlying design flow remains unchanged. Fundamental Tcl commands remain unchanged so that existing projects and scripts will safely migrate forward. Designs and scripts created prior to Vivado 2019.2 require no modification when updating to this release.

The following sections describe a few specialized commands and options needed for the DFX flows. Examples of how to use these commands to run a DFX flow are given. For more information on individual commands, see 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 this link 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 in conjunction 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 Xilinx® 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 Xilinx 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: Add and Link Files.*

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.*

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 -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.

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 may 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 Creating Pblocks 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.
- 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.
- Any nested Pblocks under the reconfigurable Pblock inherit the IS_SOFT = FALSE property; this cannot be changed.

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.
`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.

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

Command/Property Name	Description
PARTPIN_SPREADING	Used to control the maximum number of PartPins per INT tile. Default is 5. Setting a lower value (i.e., 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}
```

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:

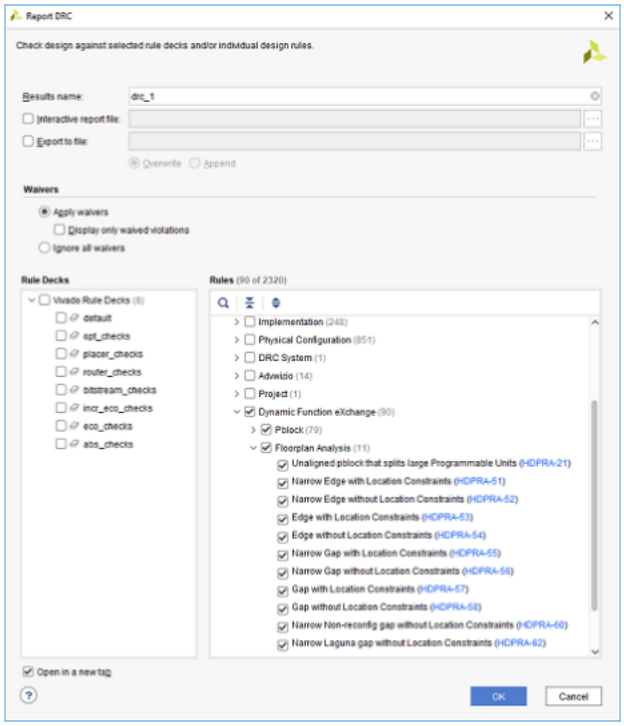
   ```
   open_checkpoint synth/Static/top_synth.dcp
   set_property HD.RECONFIGURABLE true [get_cells inst_count]
   read_checkpoint -cell [get_cells inst_count] synth/count_up/
   count_synth.dcp
   set_property HD.RECONFIGURABLE true [get_cells inst_shift]
   read_checkpoint -cell [get_cells inst_shift] 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 Partial Reconfiguration checks are present (see Floorplan in the Vivado IDE). Note that 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 Partial Reconfiguration, 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.*

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. Other scripts are created for very specific goals and are not needed in most cases.

For UltraScale and UltraScale+, unique scripts named `<rp_pblock>_Placement_AllTiles.tcl` and `<rp_pblock>_Routing_AllTiles.tcl` show the boundaries for the placement and expanded routing for the reconfigurable region. The placement script shows the range available for logic placement after snapping is finished. The routing script shows the expanded routing region and represents the contents of the partial bitstream created for that RP.

For all devices, 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.

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 this link (for defining clocks) and this link (for 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.

***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.

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.

Apply Reset After Reconfiguration

With the Reset After Reconfiguration feature, the reconfiguring region is held in a steady state during partial reconfiguration, and then all logic in the new RM is initialized to its starting values. Static routes can still freely pass unaffected through the region, and static logic (and all other dynamic regions) elsewhere in the device continue to operate normally during Partial Reconfiguration. Dynamic Function eXchange with this feature behaves in the same manner as the initial configuration of the FPGA, with synchronous elements being released in a known, initialized state.

IMPORTANT! *Release of global signals such as GSR (Global Set Reset) and GWE (Global Write Enable) are not guaranteed to be synchronized chip-wide. If functionality within a RM relies on synchronized startup of initialized sequential elements, the clock(s) driving the logic in that module or Clock Enables on these elements can be disabled during reconfiguration, then re-enabled after reconfiguration has been completed. For more details, see the "Design Advisory for techniques on properly synchronizing flip-flops and SRLs" answer record (AR#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 will be blocked (held in reset) during partial reconfiguration when RESET_AFTER_RECONFIG is enabled. The interface will be non-responsive (busy), and there will be no access during the length of the reconfiguration period. The interface will become 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, since that matches the base region for a reconfigurable frame. The width of a Pblock does not matter when using `RESET_AFTER_RECONFIG`.

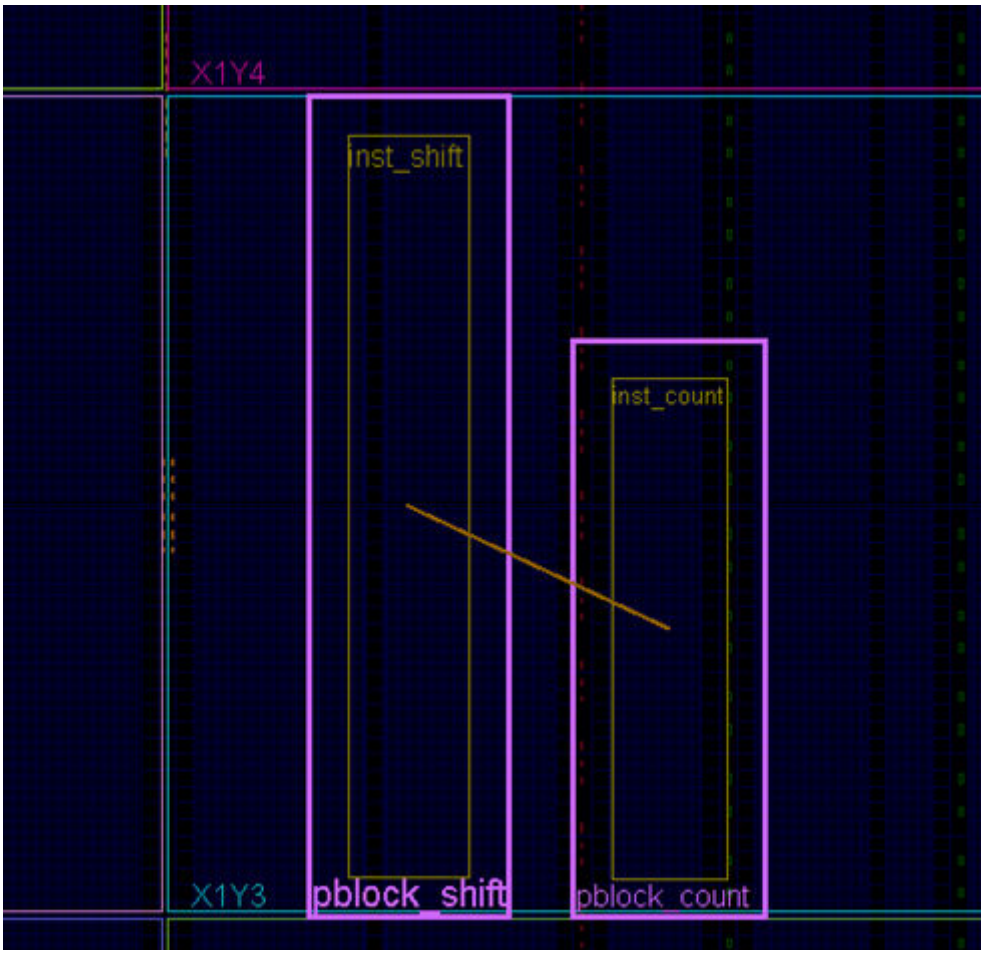
UltraScale and UltraScale+ devices do not have this clock region alignment requirement, and GSR can be applied at a fine granularity. Because of this, `RESET_AFTER_RECONFIG` is automatically applied for all RPs in the UltraScale and UltraScale+ architecture. This capability cannot be disabled.

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.

1. 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.
2. 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. See Automatic Adjustments for Reconfigurable Partition Pblocks (for 7 series devices) or Automatic Adjustments for PU on Pblocks (for UltraScale and UltraScale+ devices) for more information.

Figure 8: **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 utilizes 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 `DONEO` is not an option to prevent the `DONE` pin from changing state, since 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.

Moreover, in order 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 only, this process is split between two bitstreams -- see Clearing 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 only, an alternative approach would be to forego 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.

Software Flow

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

Synthesis

Each module (including Static) needs to be synthesized bottom-up so that a netlist or checkpoint exists for static and each RM.

1. Synthesize the top level:

   `read_verilog top.v` (and other HDL associated with the static design, including black box module definitions for RMs), then:

   ```
   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, then other configurations may 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 Xilinx 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 Compilation feature for any child configuration run. The flow is documented here in *Vivado Design Suite User Guide: Implementation* (UG904), and best results are seen when 95% of design instances match.

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. Support in this release for the second configuration and beyond; 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 in conjunction 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.

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

For more information on creating encrypted design checkpoints and the options available, please consult Chapter 6 of *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.

When just 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.

Xilinx 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 will be 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.

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 "gray box 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 (Nested 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.

While there is no formal limit to the number of levels into which a device may be subdivided, the further you go, the more difficult it will be to place and route. Moreover, 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 in this release is available for UltraScale and UltraScale+ targets, including Zynq UltraScale+ MPSoC and RFSoC devices. Versal support will begin in a future release, after base Dynamic Function eXchange (DFX) support has been added for this family. No 7 series devices will be supported for Nested DFX. Only the Tcl-based non-project flow is supported in this release. Project support, including IP integrator, will be considered in a future release.

Nested DFX Structure and Commands

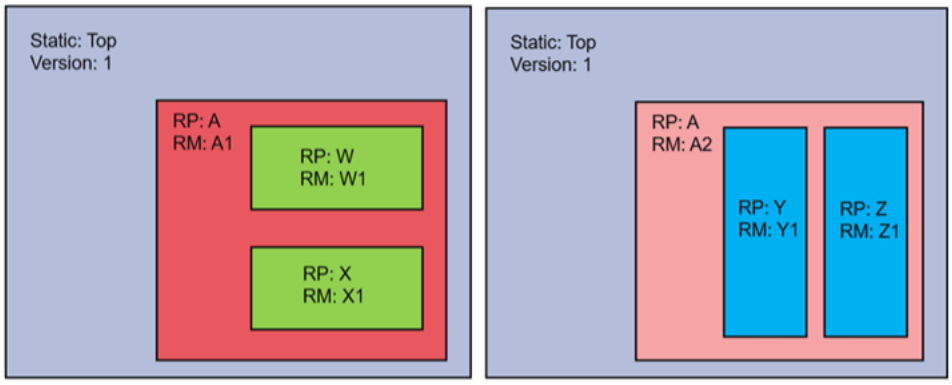
The Nested Dynamic Function eXchange flow is essentially a step and repeat through hierarchy, moving the context of static and reconfigurable boundaries with each pass through the Vivado tools. The first pass (configuration) will establish the implementation results of the static design, just as is done for the standard DFX flow. Subsequent runs will 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. We will use the images in Design Structure to describe the Vivado tool flow and design considerations:

Design Structure

Figure 9: **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` will 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 will become 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` will need 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 will prevent 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.

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 Design Structure. 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 10: **Implemented Baseline Design Prior to pr_subdivide**

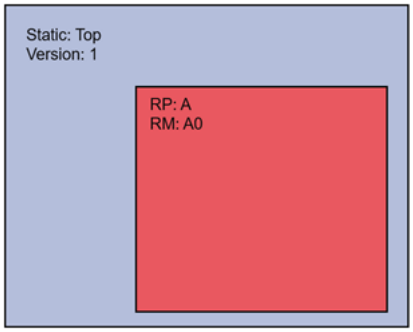
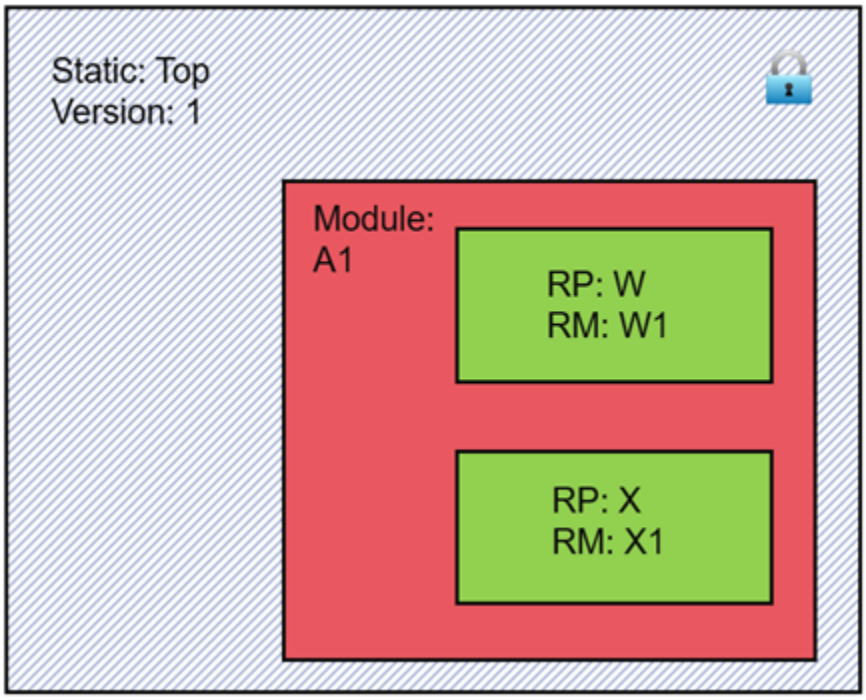


Figure 11: **Design After pr_subdivide**



Three key details are shown in the above 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.

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.

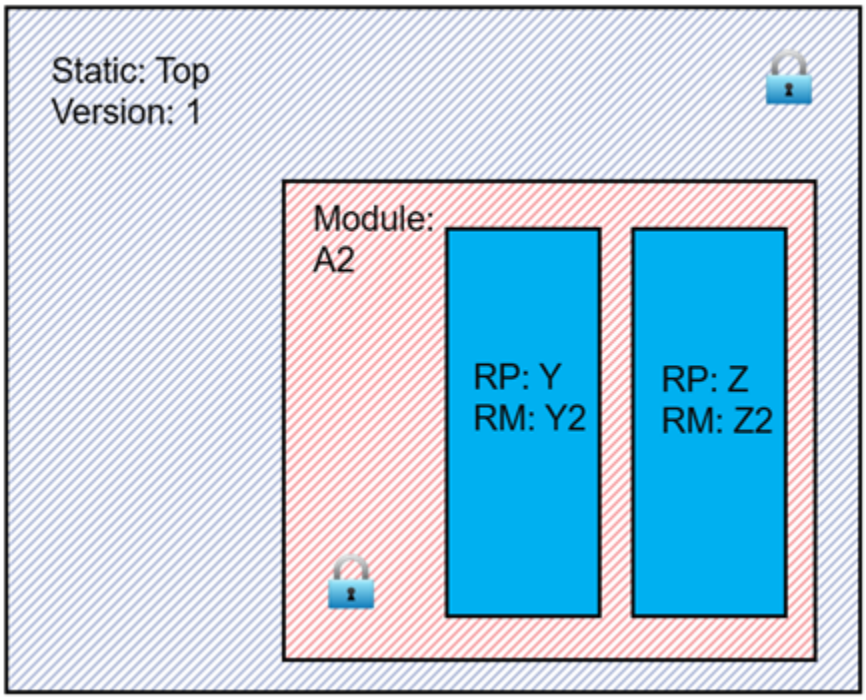
Create Reconfigurable Module Results Under A2

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 12: **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.

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 we recommend 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 we have 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

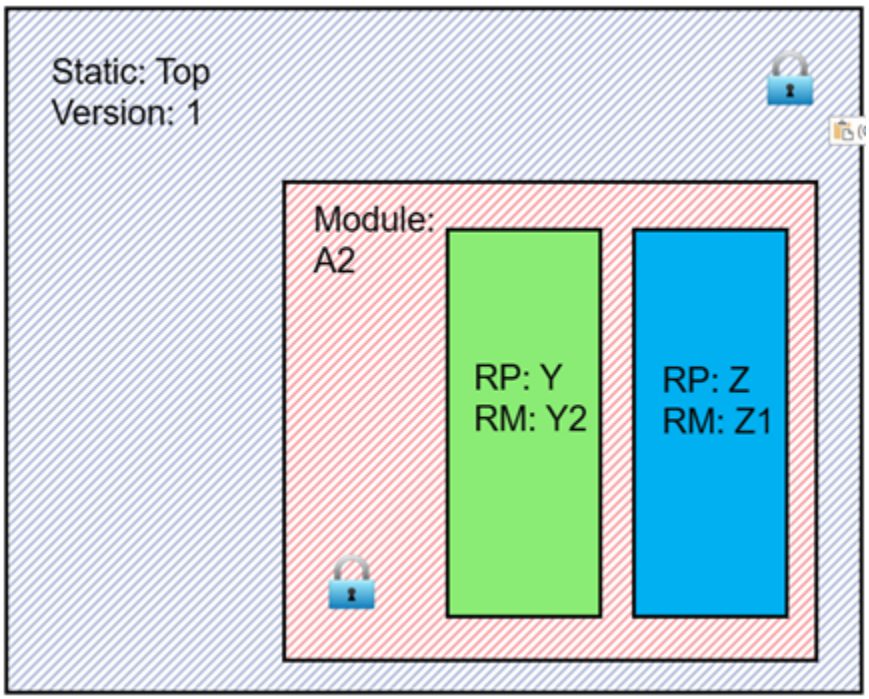
The Nested DFX design methodology moves the HD.RECONFIGURABLE property down and up through the hierarchy. Implementation tools follow standard DFX design rules based on what cells are currently defined as reconfigurable. This holds true for `write_bitstream` as well; partial bitstreams will only be created 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 will generate 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 will generate ONLY a partial bitstream for the requested cell.
2. The `-no_partial_bitfile` option will generate 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 13: **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 will create 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 are subject to the same rules and recommendations as standard Dynamic Function eXchange designs. Simply 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.

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 and 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.

Supported/Unsupported Features

This section lists the current lists of supported and unsupported features. While future enhancements may be possible, this list is not an indication that these enhancements will be delivered in a future revision.

Supported Features

- Device support: All UltraScale and UltraScale+ devices, including MPSoC and RFSoC
- Subdivide a DFX design into any depth of levels

Unsupported Features

Some features are not yet implemented but are planned for future releases.

- Device support: Versal will be supported in an upcoming release. 7 series will not be supported.
- Project support: Vivado projects, including those for IP integrator, are not yet supported.

Known Limitations

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, and remember that implementation runs can be launched in parallel once the design structure and floorplan is established.

Abstract Shell for Dynamic Function eXchange

Xilinx UltraScale+ 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:

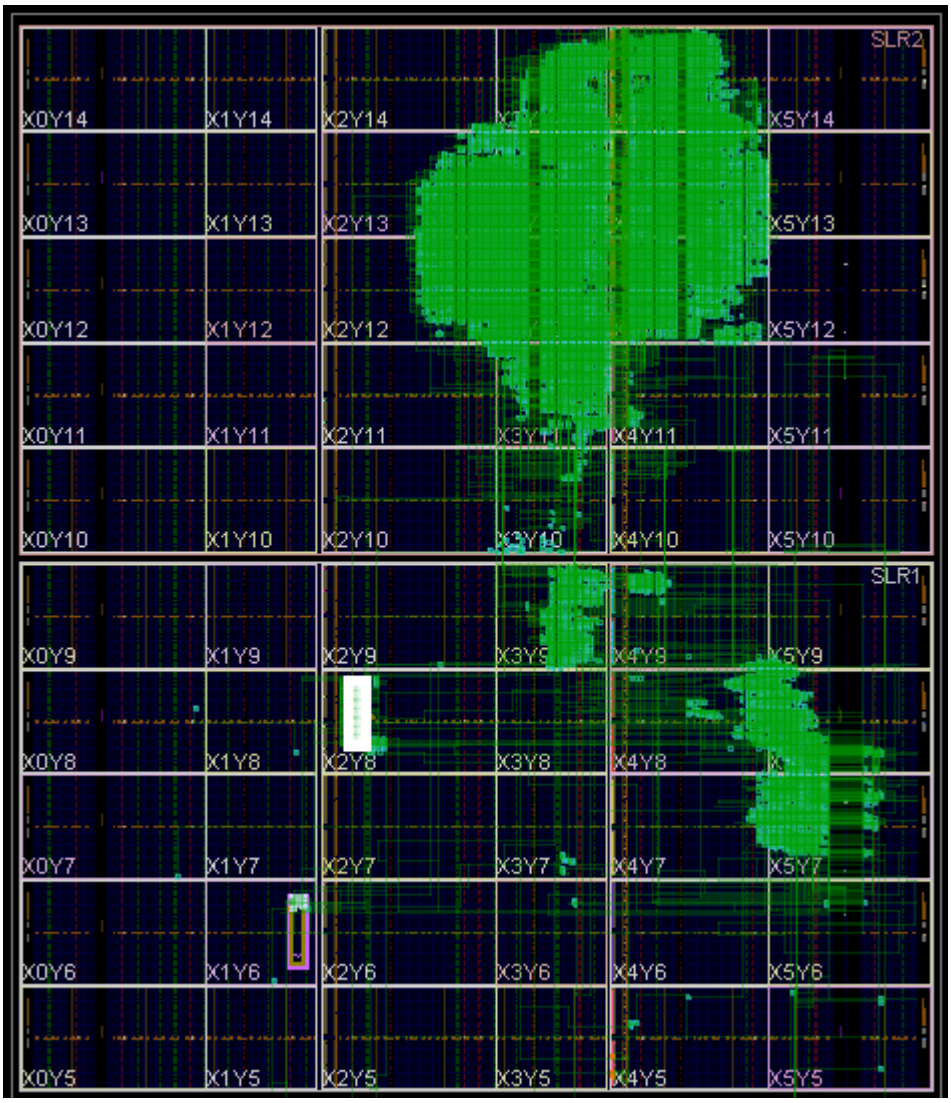
- Implement a new RM within a specific RP
- Validate timing and pass PR Verify
- Generate a partial bitstream for the RM

By using this approach, you can:

- Reduce runtime 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

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 14: **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 15: **Abstract Shell for the u_shift RP**

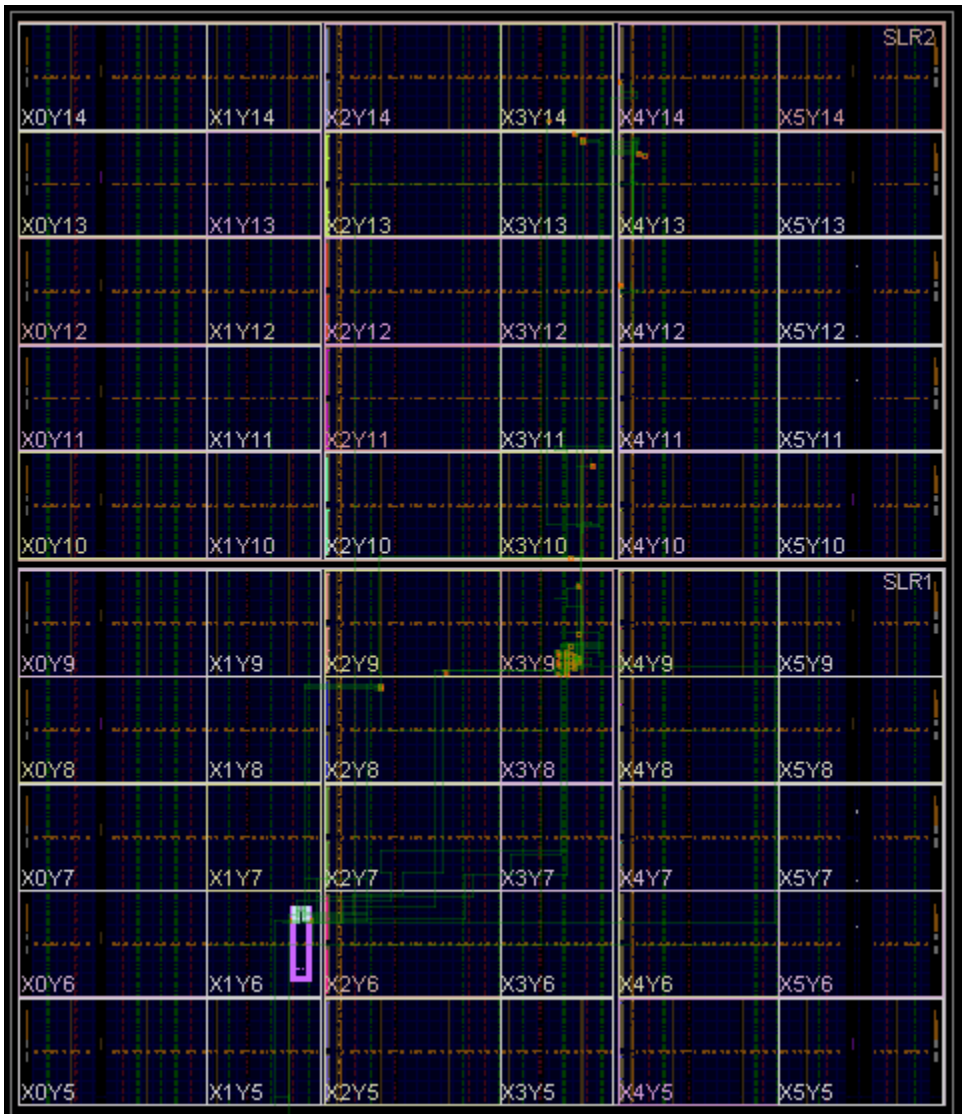
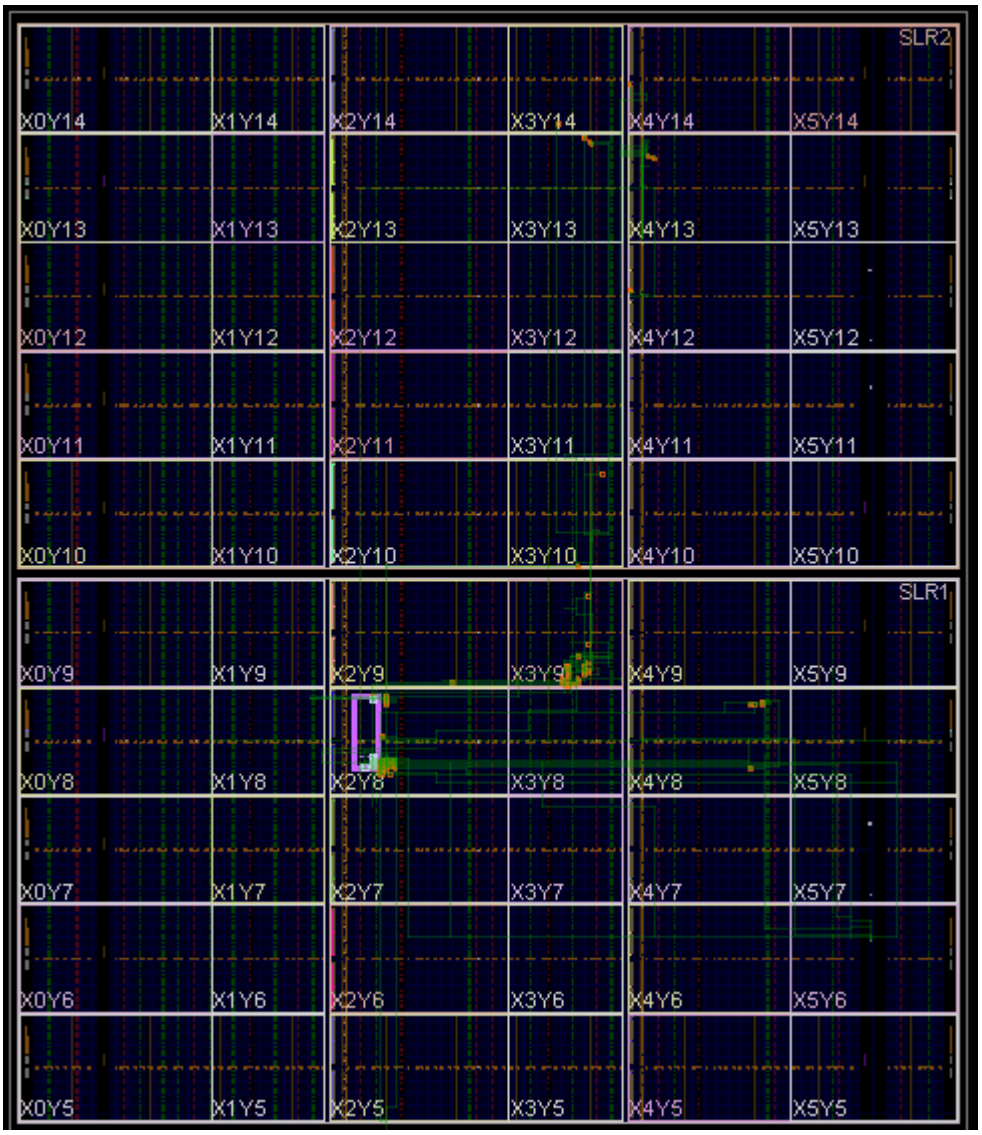


Figure 16: **Abstract Shell for the u_count RP**



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.

Even though project mode is not supported for Abstract Shell runs, the initial parent configuration can be set up and implemented using project mode within the Vivado IDE. All subsequent RM implement runs, however, must be implemented outside the project.

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:

- 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 may 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 may 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.

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, VivadoVivado returns 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 gray box 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.

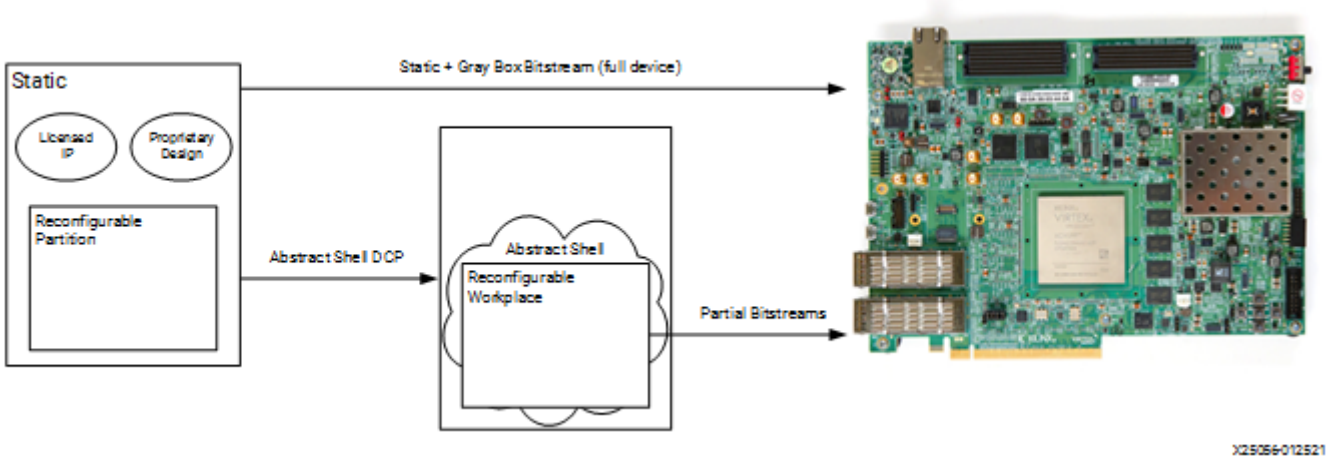
Device Programming Strategies

At the end of these implementation and bitstream generation processes, you have full and partial bitstreams just as you do with the standard DFX flow. For single-user scenarios, follow the recommendations and considerations outlined in Chapter 5: Design Considerations and Guidelines for All Xilinx Devices, Chapter 6: Design Considerations and Guidelines for 7 Series and Zynq Devices, and Chapter 7: Design Considerations and Guidelines for UltraScale and UltraScale+ Devices in this document.

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 could have a gray box (no functionality, just LUT 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 17: **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.

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 18: **Routed Reconfigurable Module within an Abstract Shell**

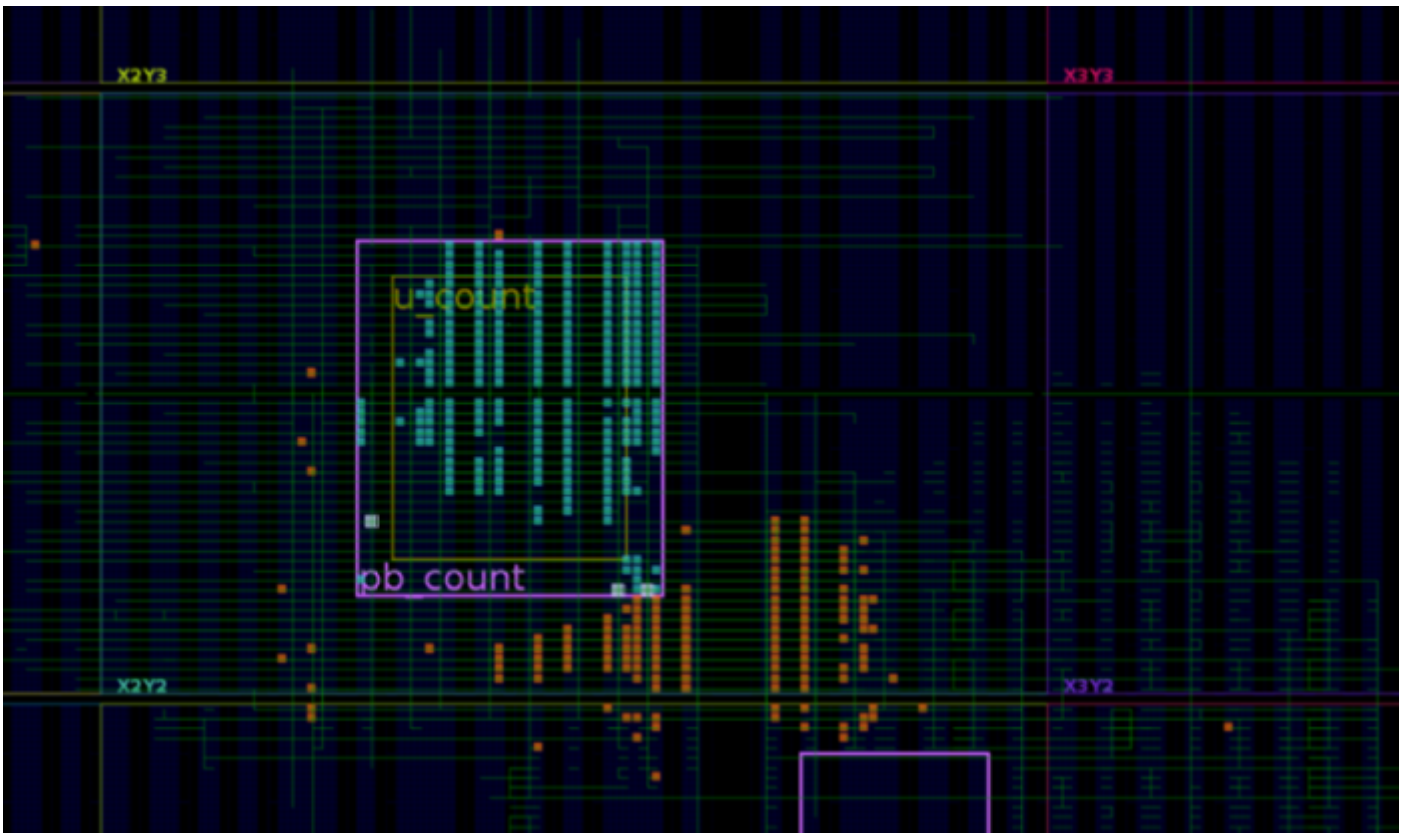
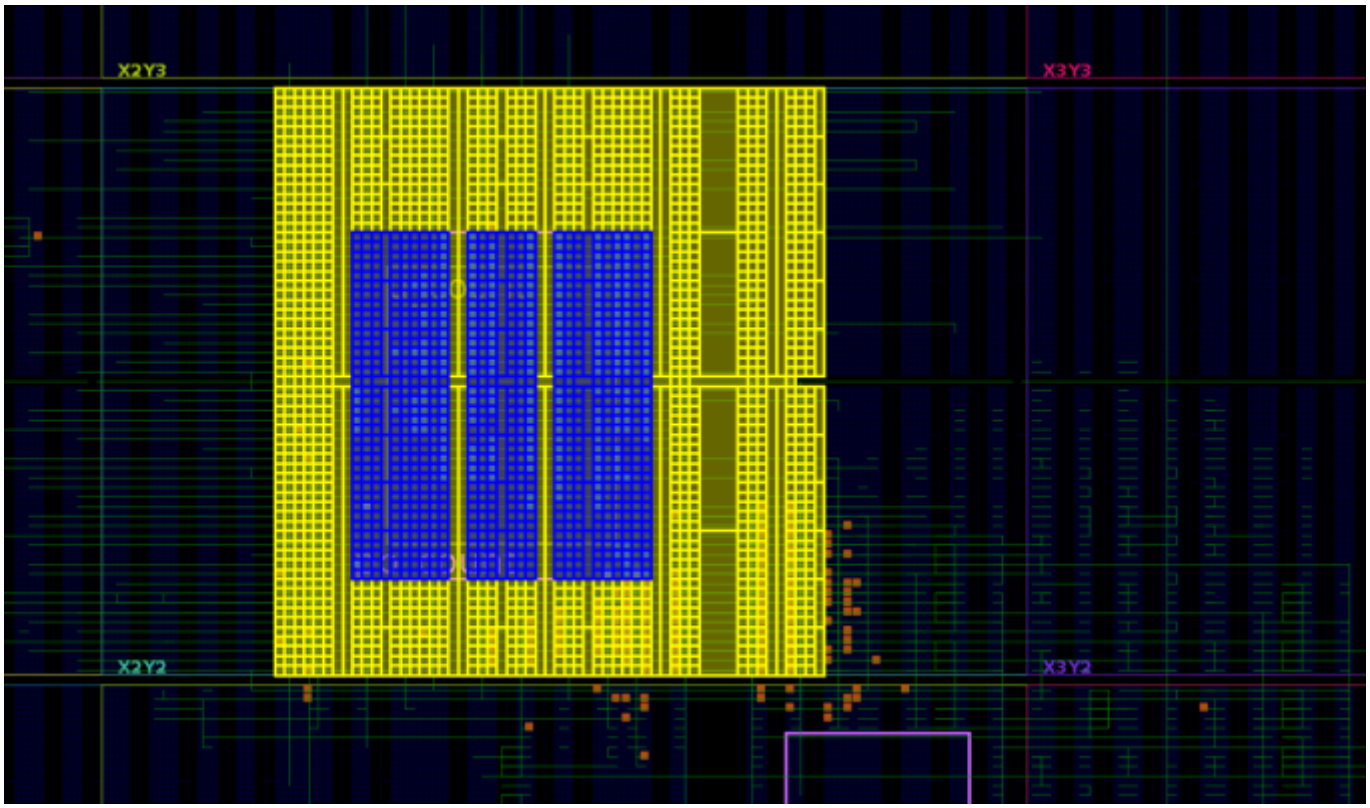


Figure 19: **Expanded Routing Region (yellow) for the Reconfigurable Partition (blue)**



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.

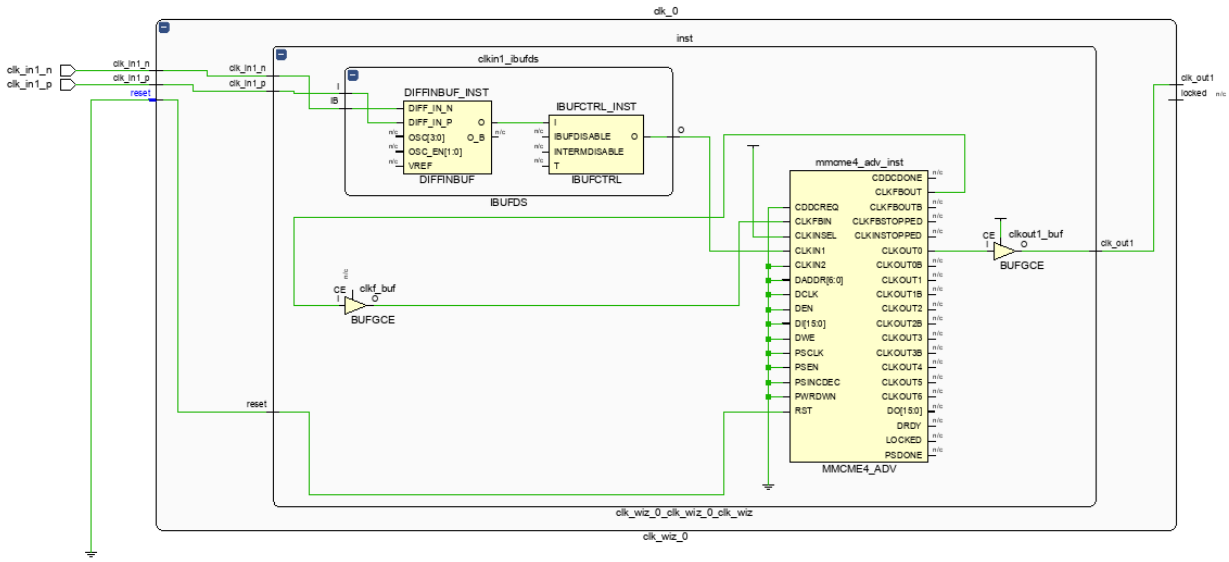
Logical Connectivity Contents

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 20: **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 (Xilinx 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@xilinx.com for any Xilinx IP, or the provider of the IP for any non-Xilinx IP.*

Abstract Shell and Nested DFX

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, Xilinx 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.

Supported/Unsupported Features

This section shows the current lists of supported and unsupported features in this version of Vivado. While future enhancements may be possible, this list is not an indication that these enhancements will be definitively delivered in a future revision.

Supported Features

- Abstract Shell creation for all UltraScale+ devices
- Implementation of any RM within an Abstract Shell for a given RP
- Validation of an implemented RM within an Abstract Shell via PR Verify
- Partial bitstream generation for any RM from an Abstract Shell

Unsupported Features

Some features are not yet implemented but may be considered for future releases.

- Project mode is not yet supported. However, users can use a DFX project to create all the post-synthesis checkpoints and implement the parent configuration. Use that routed checkpoint to jump out to a non-project flow for all the RM implementation runs.
- Abstract Shell does not support UltraScale devices. Versal support will be added in a future Vivado release, after DFX support is released for that architecture.
- Abstract Shell does not and will never support 7 series architectures.

Known Issues and Limitations

- Some issues have been seen in PR Verify due to partition pins. If PR Verify between the original Abstract Shell and the Abstract Shell with an implemented RM returns HDPRVerify-10 or HDPRVerify-11, look at the net (typically a clock, in a design with multiple RPs) they have in common (shown in the latter message). A workaround is to reset the partition pin location constraint prior to running PR Verify:

  ```
  reset_property HD.PARTPIN_LOCS [get_pins RP_Top/MyClk200]
  ```

- 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 placing and routing new RMs in an Abstract Shell checkpoint, are encountered, check the behavior using a full static design checkpoint first.

Vivado Project Flow

Dynamic Function eXchange (DFX) in Xilinx® 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 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	All architectures
Top Level design source	Verilog or VHDL	Block Design (with RTL wrapper)
Sources supported within RMs	IP, RTL, and EDIF	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 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 are implemented for this release, featuring 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 in this release, however, underlying Tcl commands has remain unchanged so that existing projects and scripts will 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 as Vivado 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 a Xilinx Evaluation Platform, see Lab 3 and Lab 4 in the *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947).

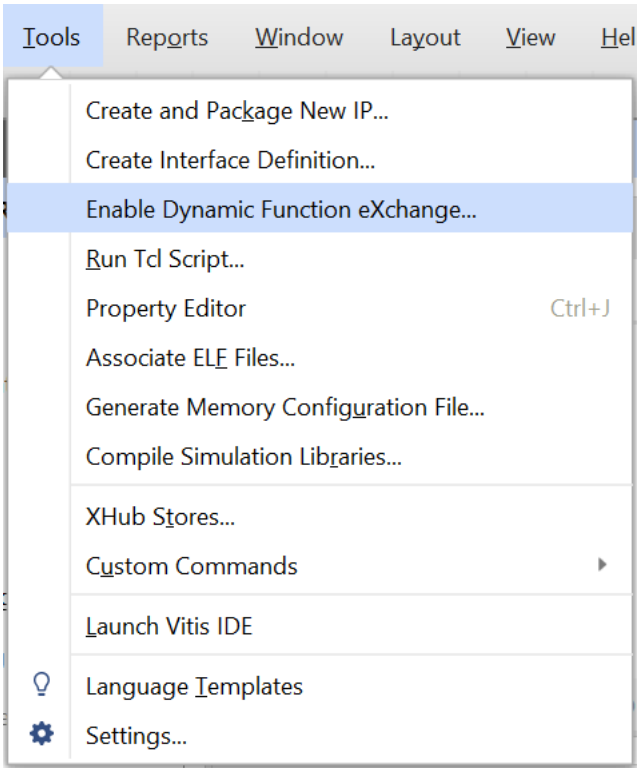
Creating a Dynamic Function eXchange Project

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 Partial Reconfiguration 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 Xilinx recommends archiving your project prior to selecting this option.

Figure 21: **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

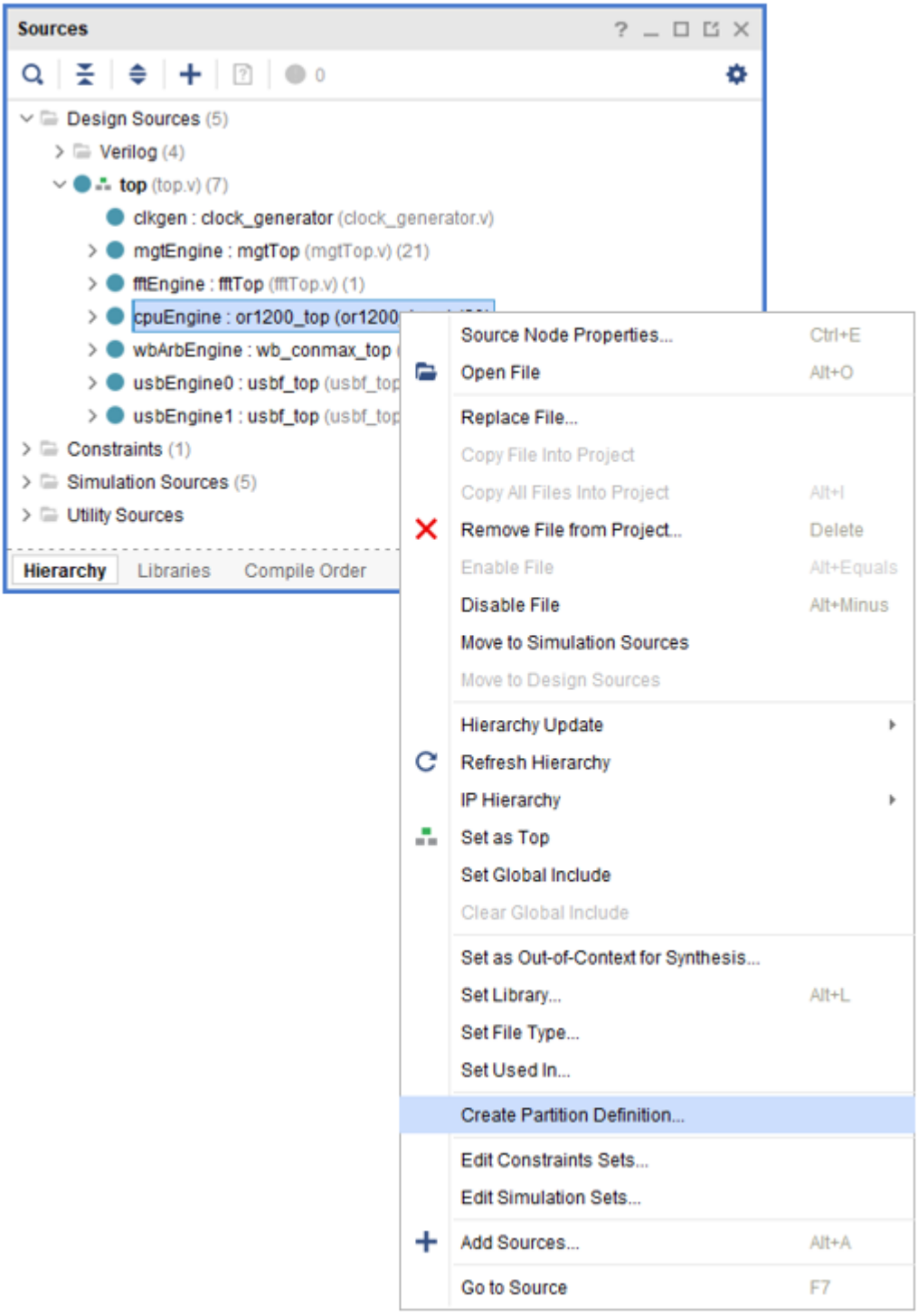
Defining Reconfigurable Partitions

Once 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 or EDIF 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) or EDIF modules in the underlying RTL
- Do not have IP, DCP or EDIF 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. This module can be a black box if design sources do not yet exist.

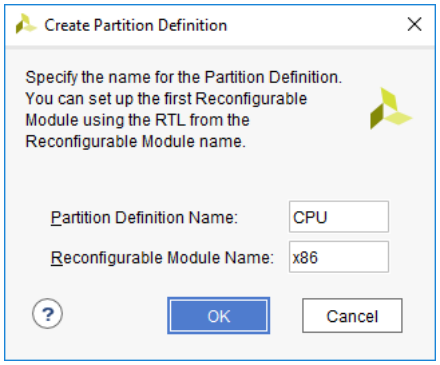
Figure 22: **Creating a Reconfigurable Partition**



2. In the next 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 or netlist sources currently residing in this level of hierarchy. More RM are added or created later in the flow.

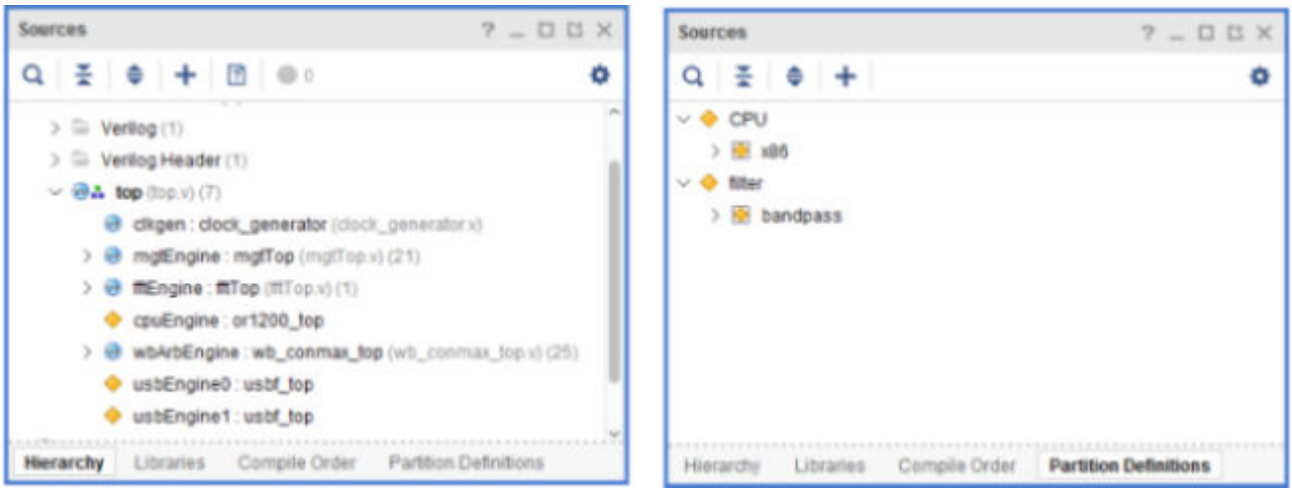
IMPORTANT! *Every instance of the selected module is turned into a RP. In order for one instance to be defined as reconfigurable and another instance to remain static, the two instances must be given unique module names.*

Figure 23: **Defining the Reconfigurable Partition and the First Reconfigurable Module**



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 a 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.

Figure 24: **Partition Definitions as Seen in the Sources Window**



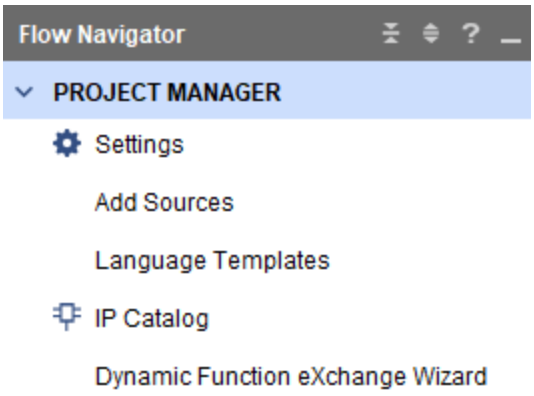
Completing the Dynamic Function eXchange Project Structure

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 may step forward and back within the wizard until everything is completed to your liking, and you may 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 25: **The Dynamic Function eXchange Wizard in the Flow Navigator**

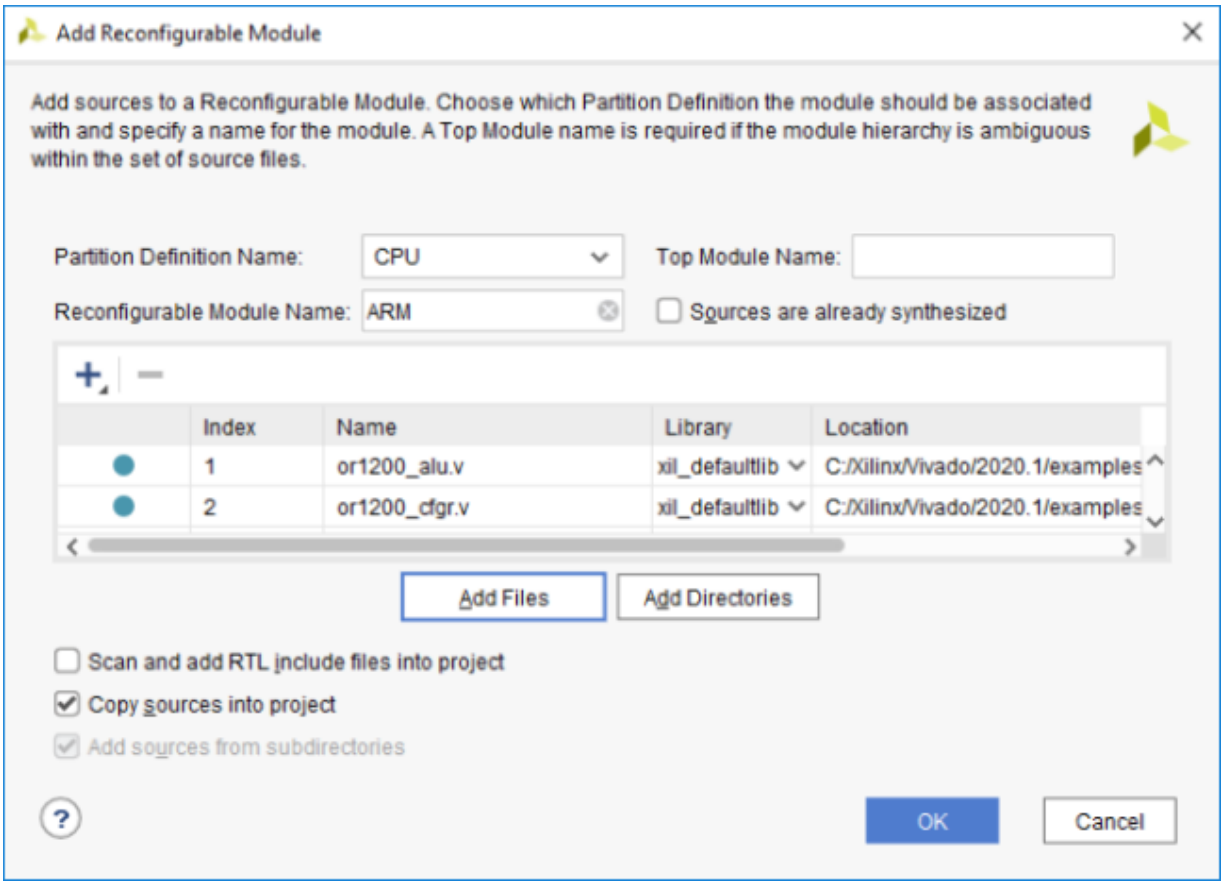


When the DFX Wizard opens, step through each stage of DFX project management.

Editing Reconfigurable Modules

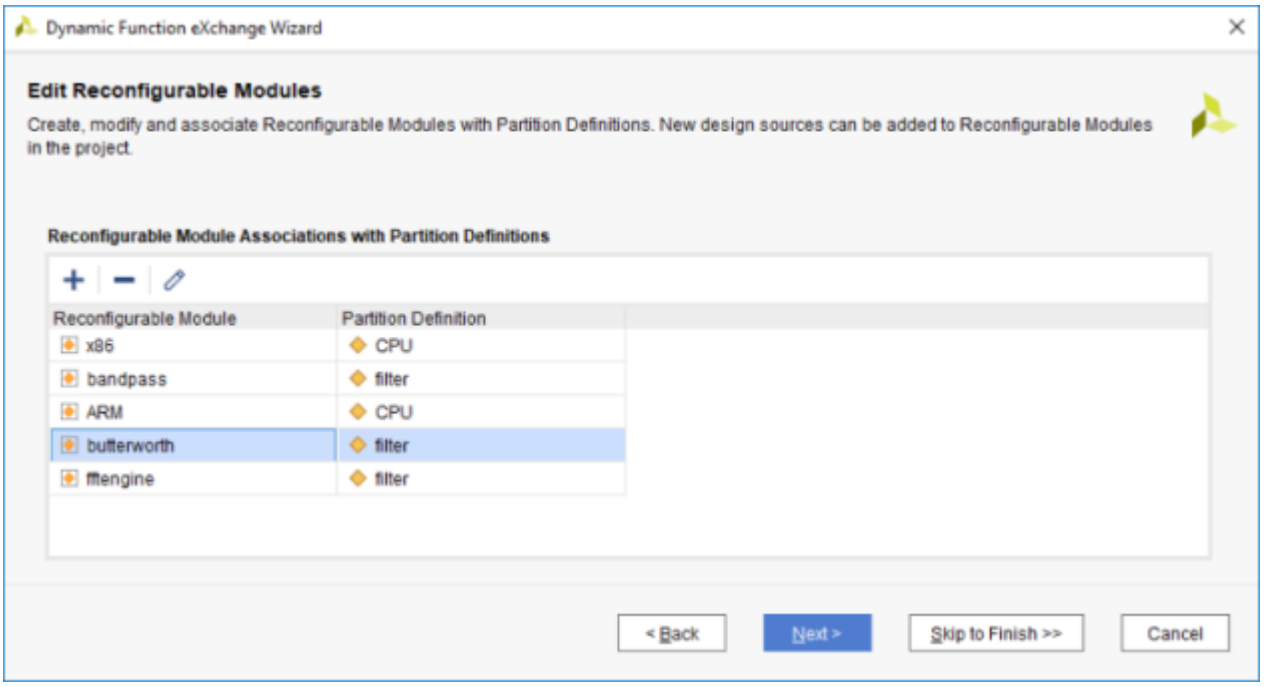
1. After selecting **Next** to progress past the introduction, the first content page allows you to define new RMs for any Partition Definitions (PD) defined. The first RM for each PD has already been included here, if the RTL/netlist source was present when the PD was created. Click on the blue **+** 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.

Figure 26: **Creating a New Reconfigurable Module**



2. Repeat this process for all existing RMs for every Partition Definition. If a gray box module is desired, no action is required. Note that RMs may be edited by clicking on the pencil icon or removed by clicking the red - icon. When all RM are accounted for click **Next**.

Figure 27: **Set of Reconfigurable Modules Defined**

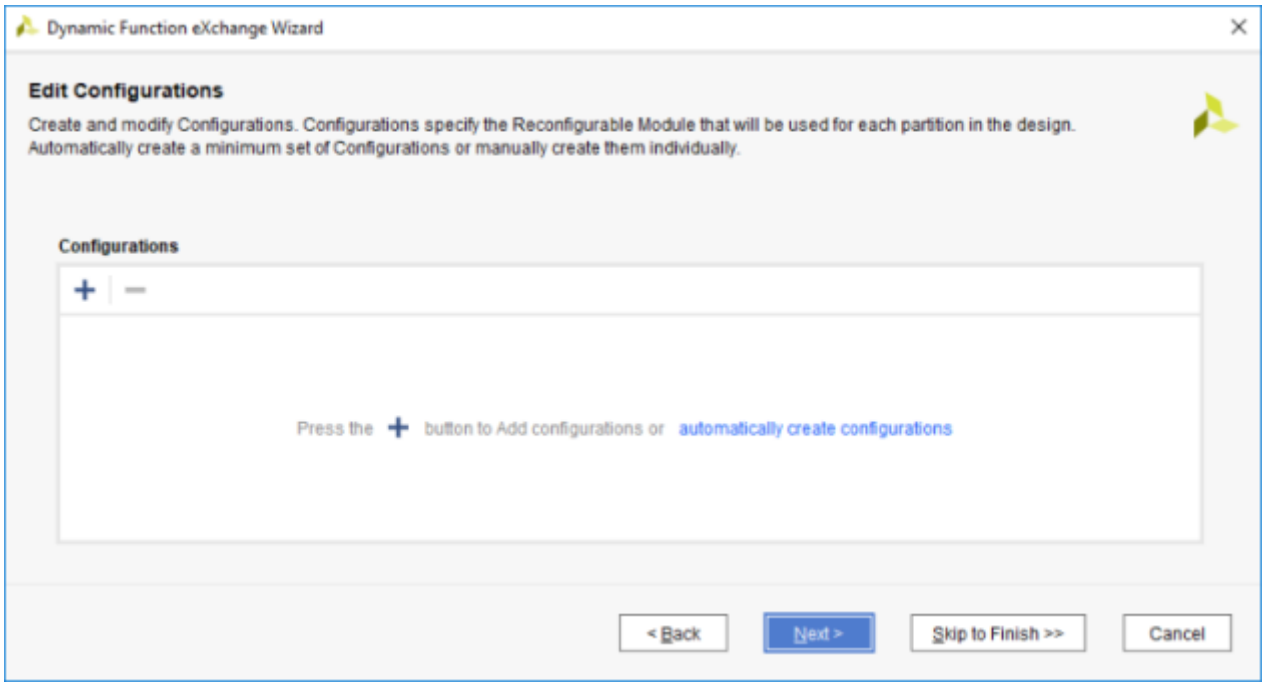


Editing Configurations

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.

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

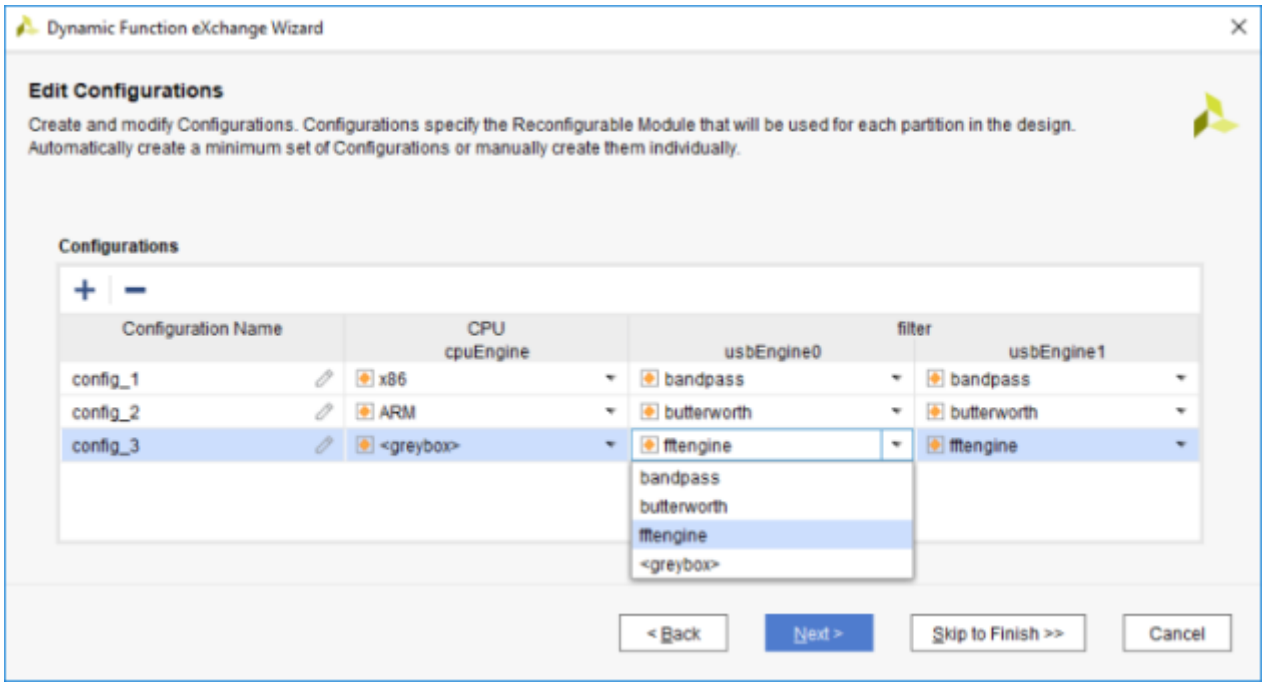
Figure 28: **Edit Configurations before Creating Any Configurations**



If one Partition Definition has more RMs than another, a graybox 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 may be created if desired.

TIP: *Graybox modules are different than black box modules, as they are not truly empty. Graybox 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. Vivado creates these by calling* `update_design -buffer_ports` *on selected modules.*

Figure 29: **Edit Configurations after Automatic Generation of Configurations**

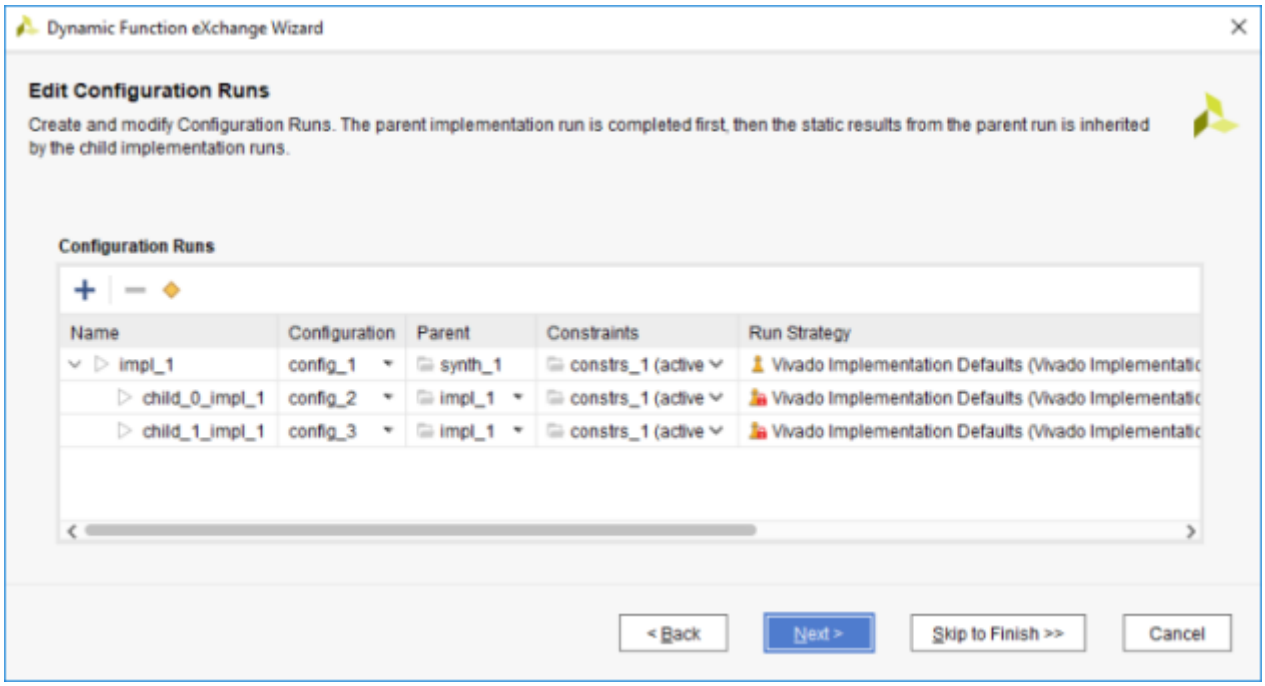


***Note*:** If one RM is used in more than one Configuration, the implementation results may be different, as place and route is performed each time, but only if the RM was initially implemented in a child run. RM implementation results will be reused if they were originally done in the parent configuration. 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. Just like the Configurations themselves, Vivado 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.

Figure 30: **Automatically Generated Configuration Runs**



This structure assumes that the first configuration is the most critical or challenging. Users 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.

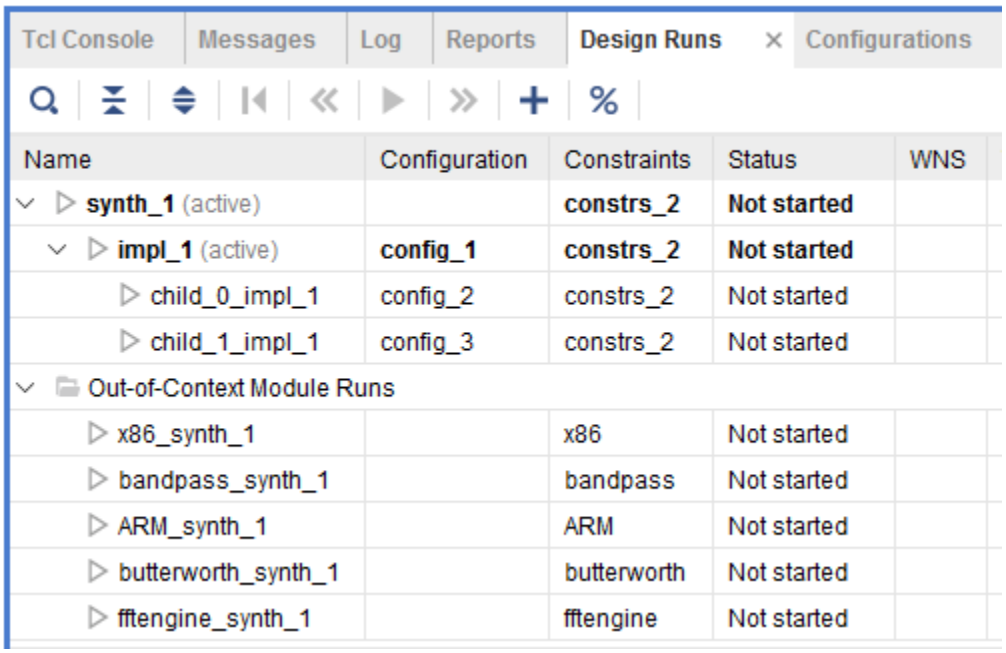
As you explore place and route options, timing closure techniques, and otherwise elaborate on the DFX design, multiple independent parent runs can be used for exploration. Multiple parent runs can be launched in parallel, then child runs can be launched after parent runs complete. Vivado project management 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 will create compatible full and partial bitstreams. This is managed in the Vivado DFX Project flow by establishing a parent-child relationship for related Configurations*

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.

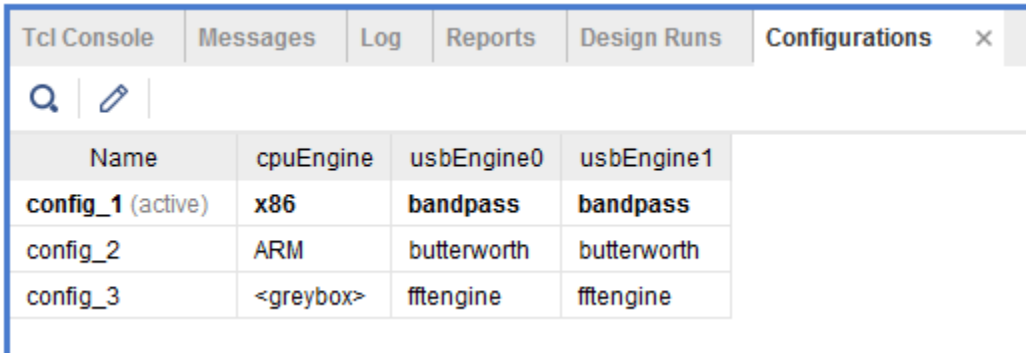
In the Design Runs window, out-of-context synthesis runs are created for each RM, and all Configuration Runs are generated. Relationships between parent and child runs are shown by the levels of indentation.

Figure 31: **Design Runs for Synthesis and Implementation**



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

Figure 32: **Configurations Available in the Project**

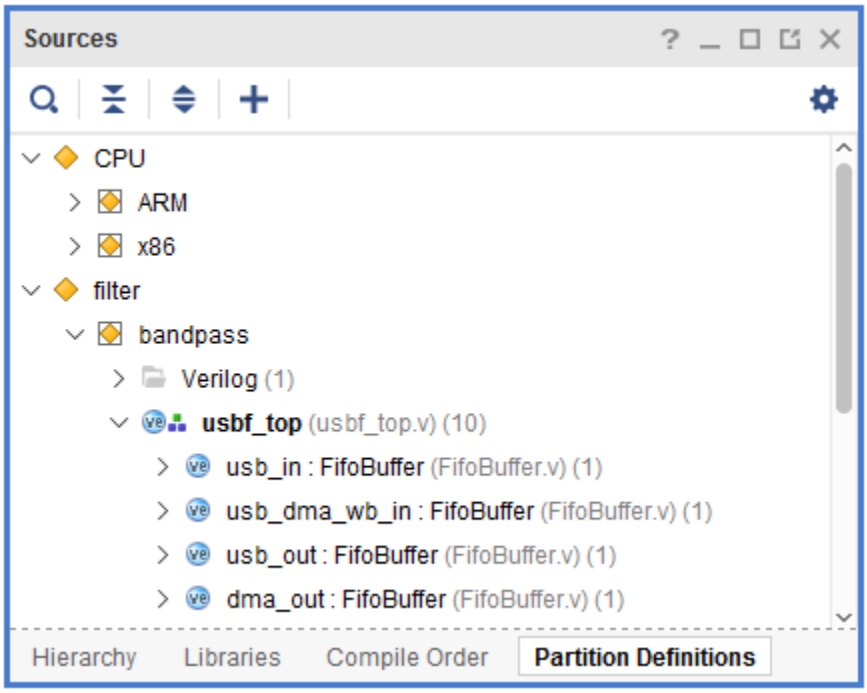


Adding or Modifying Reconfigurable Modules or Configurations

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 33: **Sources Shown in Partition Definitions View**



Adding Reconfigurable Module Constraints

All the constraints in the primary constraint set (`constrs_1` by default) will be applied through synthesis and implementation of the parent run. All constraints for the implemented parent run are contained within that locked static checkpoint so there is no need to reapply these static constraints for child runs.

IMPORTANT! *If new constraints unique to RMs are required, they must be applied in different XDC sources scoped to that module and applied in a new constraint set for the applicable child run(s).*

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 34: **Specifying Global When Generating Output Products**

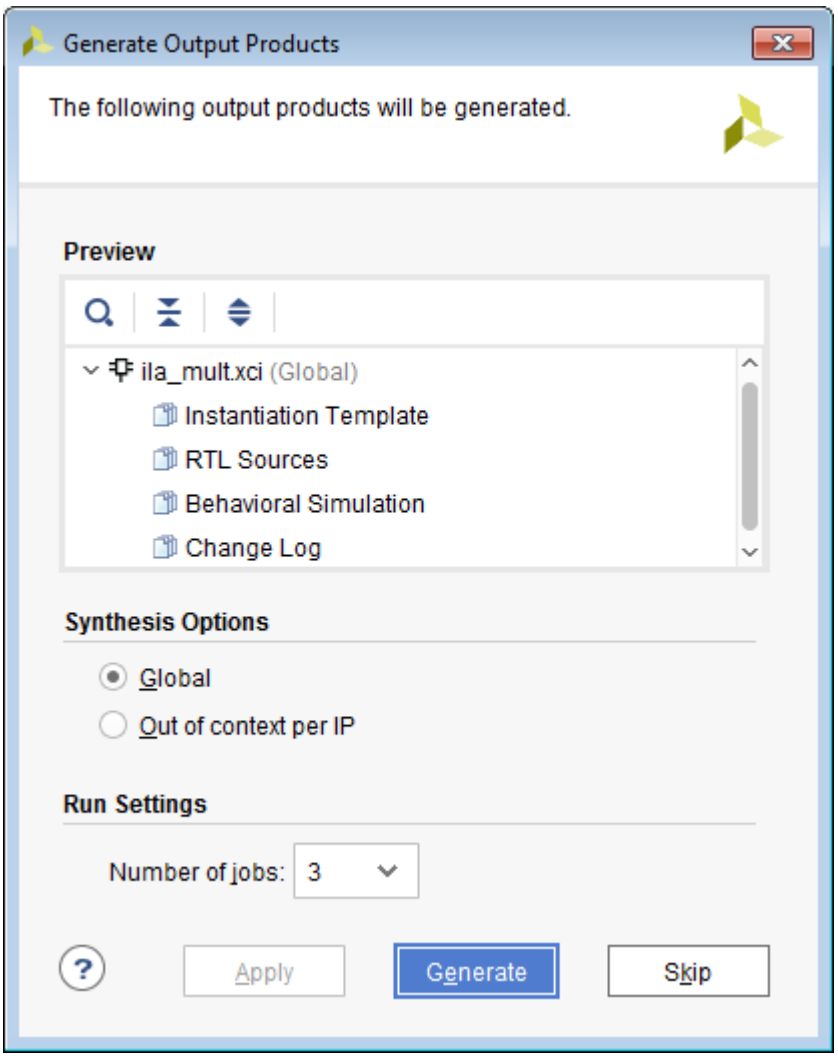
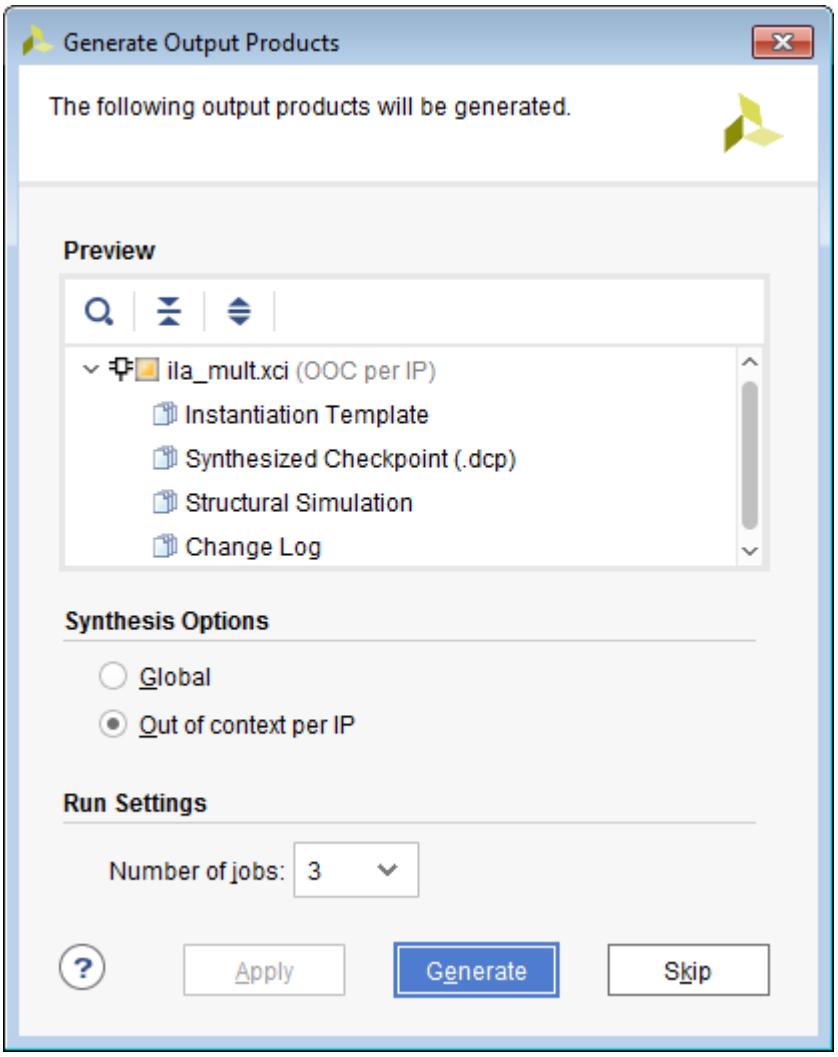
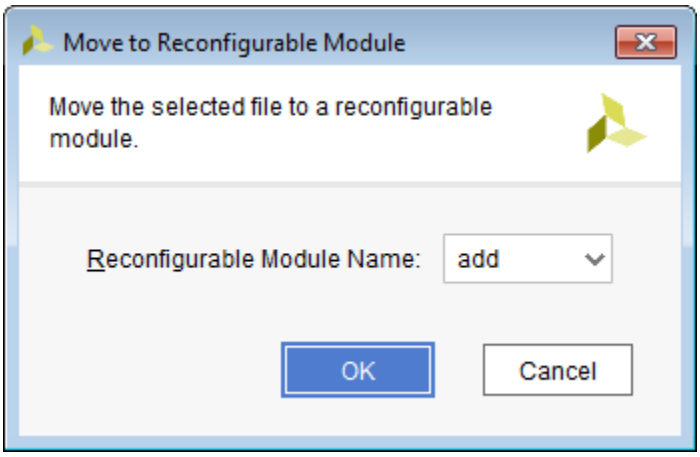


Figure 35: **Specifying Out of Context per IP When Generating Output Products**



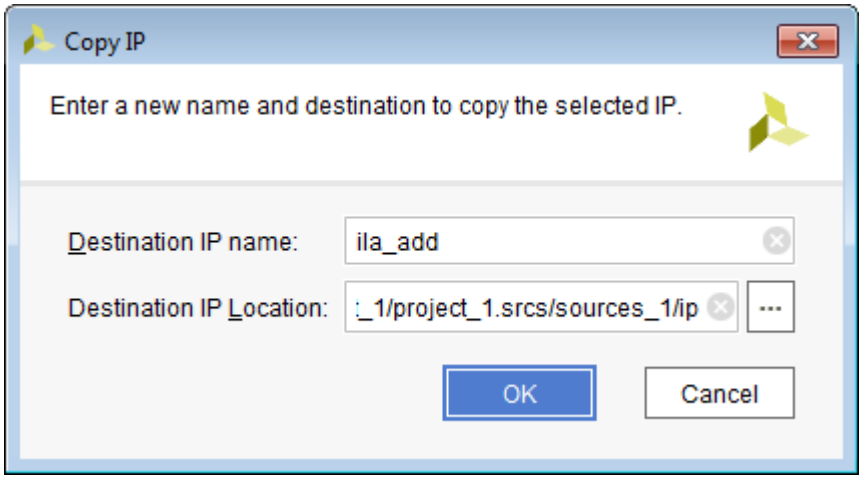
IP can be created from the IP Catalog within a DFX project. After the IP has been created, it is added to the primary blockset of the design, as the IP generation flow does not know which RM the IP is for. To assign a new IP to a specific RM, right-click on the IP in the Sources window and select **Move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

Figure 36: **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 37: **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 will launch all child runs in addition to the active parent.

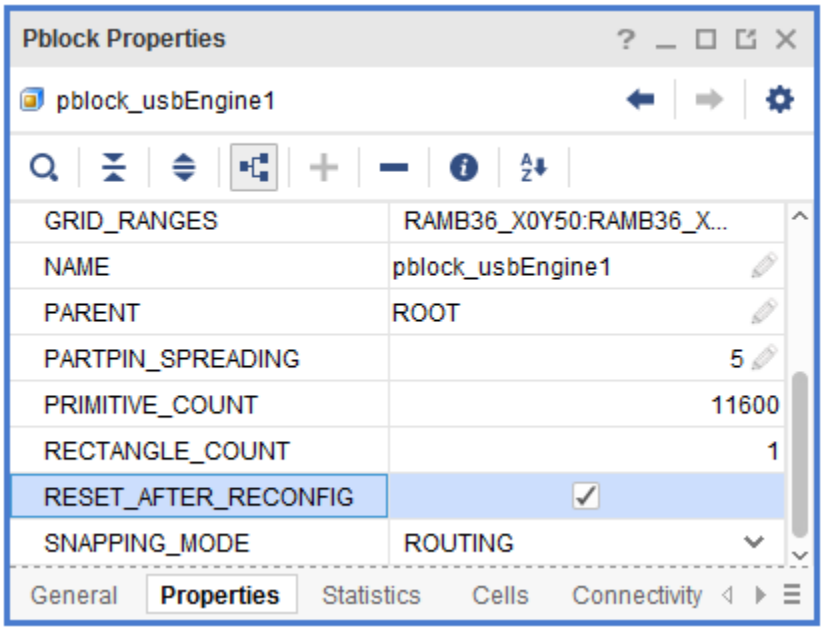
One detail that is required for Dynamic Function eXchange designs is a Pblock for each RP. Without a Pblock defined, the following error is issued in `place_design`:

```
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. In the Netlist hierarchy view, right-click on 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.

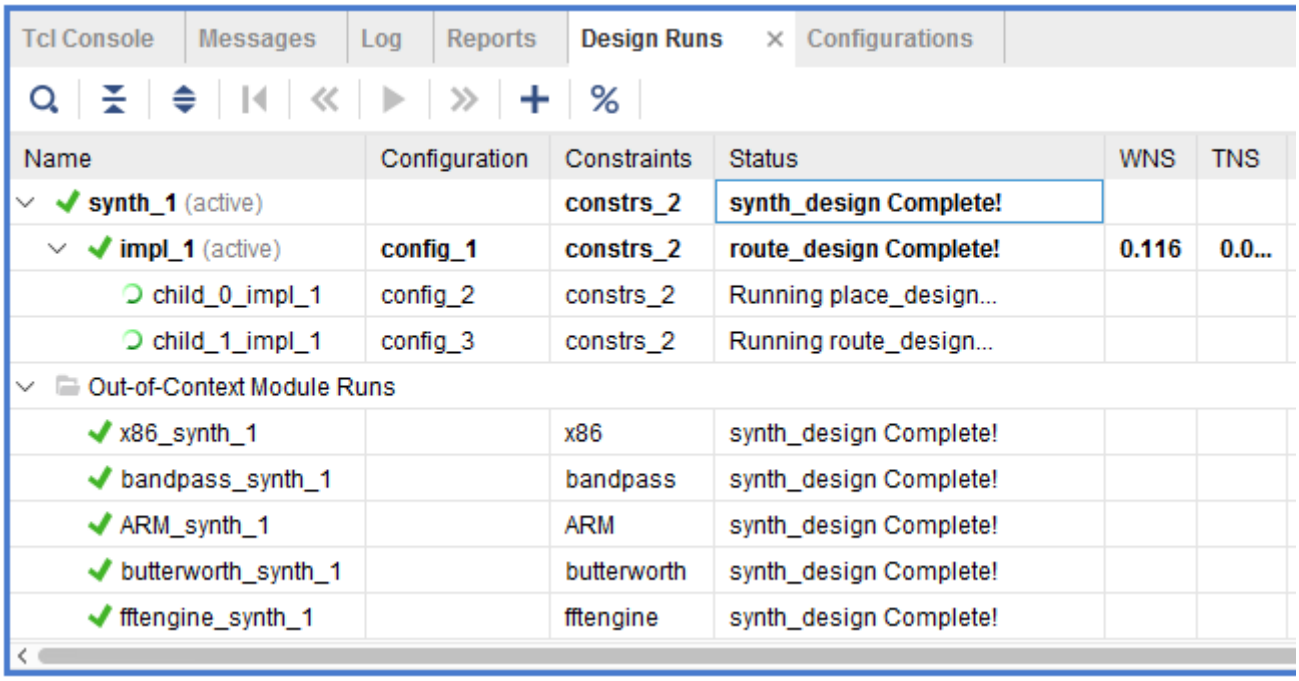
Figure 38: **Dynamic Function eXchange Properties on a Pblock (7 series)**



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.

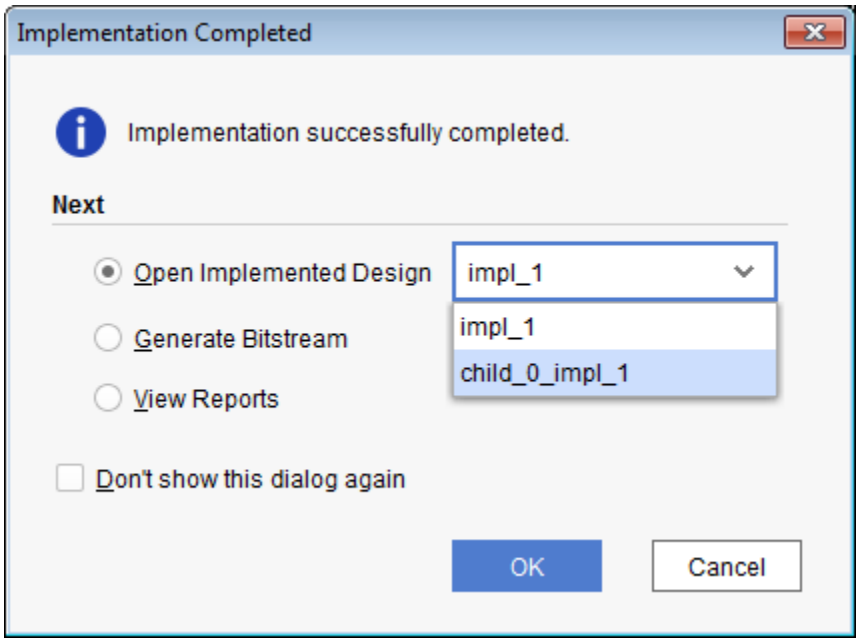
Figure 39: **Implementing Multiple Configurations in Parallel**



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.

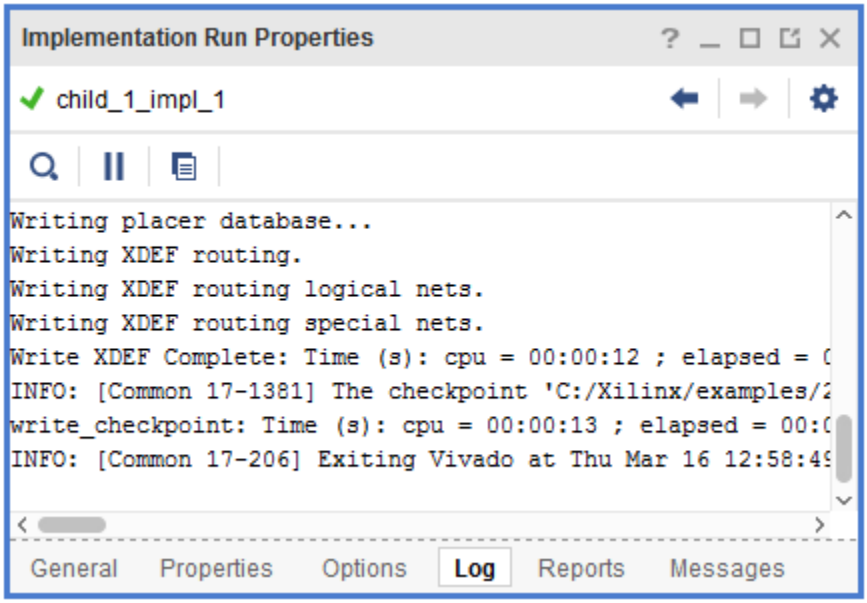
Only parent Configuration Runs can be set as active. The Flow Navigator acts upon the active run, but in the DFX flow, all child runs are also included in whatever action is requested. Pop-up messages (completed run, error, etc.) can relate to multiple runs but default to the parent run. Use the pull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

Figure 40: **Implementation Completed Dialog Box**



Everything shown in the different windows, as shown in Implementing the DFX Design, 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 41: **Implementation Run Properties Window for a Child Run**



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 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, Zynq®, UltraScale and UltraScale+ devices supported by the Dynamic Function eXchange flow.
- Source types for RMs: RTL, DCP, EDIF, XDC, XCI, XCIX.
  - XCI or XCIX (Xilinx® IP) cannot be the top level.
  - EDIF cannot be a sub-module.
- Module-level constraints must be scoped to the hierarchical instance.
- Graybox (black box module with LUT tie-off) 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 prior to 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 or within RM, ensure to use the Block Design Containers flow.*

- Simulation is not supported from within the project.

Known Limitations

- Once Partitions are defined, they cannot be undone. The only way to return to a flat non-DFX project is to create a new one.
- 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 just the parent run, or the parent and all child runs, depending on the action.

IP Integrator Using Block Design Containers

A new feature called Block Design Containers (BDC) allows users to segment designs into multiple block designs, enabling modular and team-based design flows, including DFX. Versal DFX designs are processed through IP integrator. Access to critical Versal features such as the CIPS IP and NoC are much more easily managed in IP integrator. 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

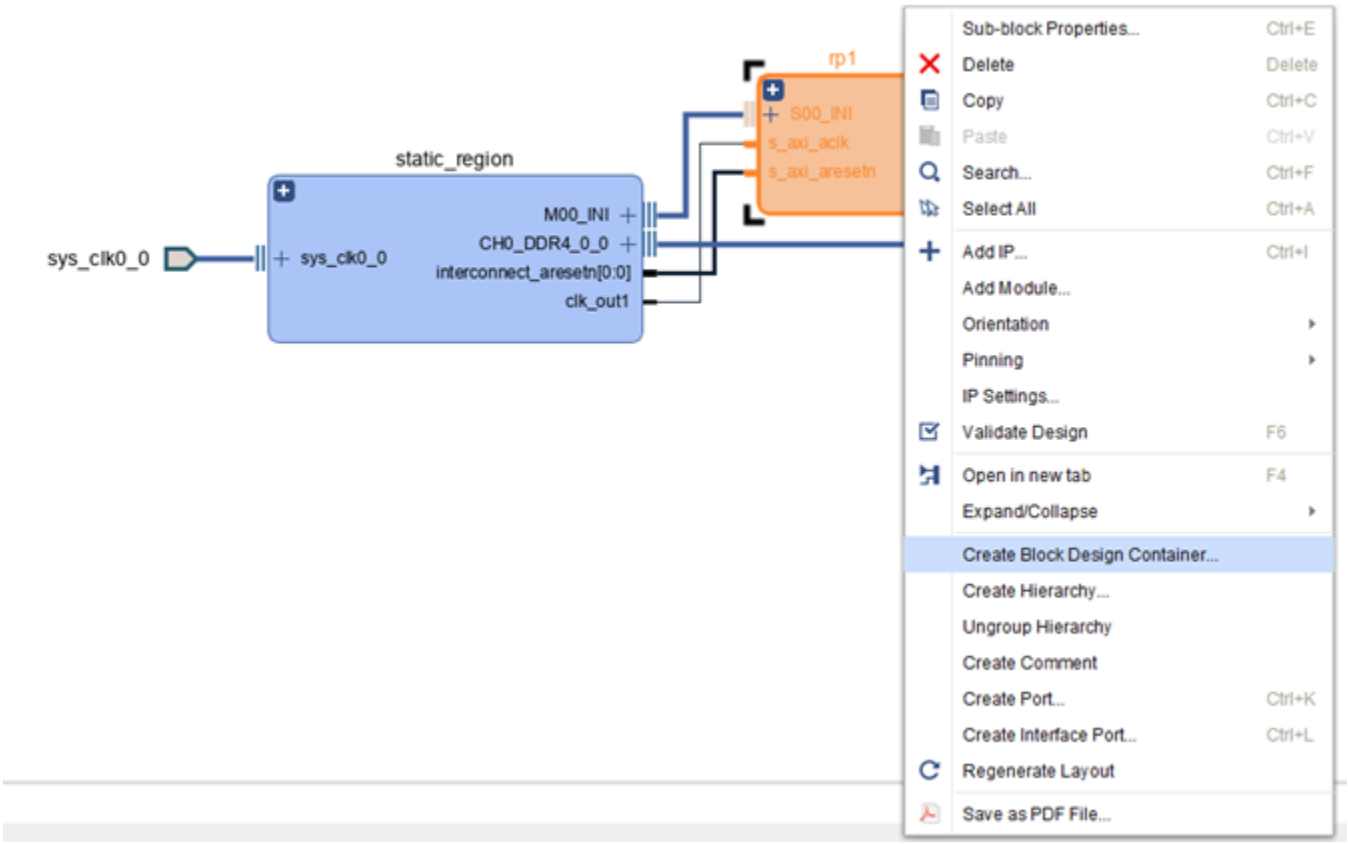
One way BDCs are introduced is by converting a level of hierarchy. To create a level of hierarchy in an existing BD, select one or more elements (using ctrl-click), then right-click to select **Create Hierarchy**. Provide a unique appropriate name to the hierarchy. The name becomes the Reconfigurable Partition name. Once this Hierarchy is turned into a DFX BDC, you can undo this action by right-clicking and selecting **Ungroup Hierarchy**. IP instances can be moved in or out of the hierarchy by dragging them in or out of the hierarchy region on the canvas.

Ensure to validate the BD once the level of hierarchy is created by selecting **Tools → Validate Design**. Alternatively, you can 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.

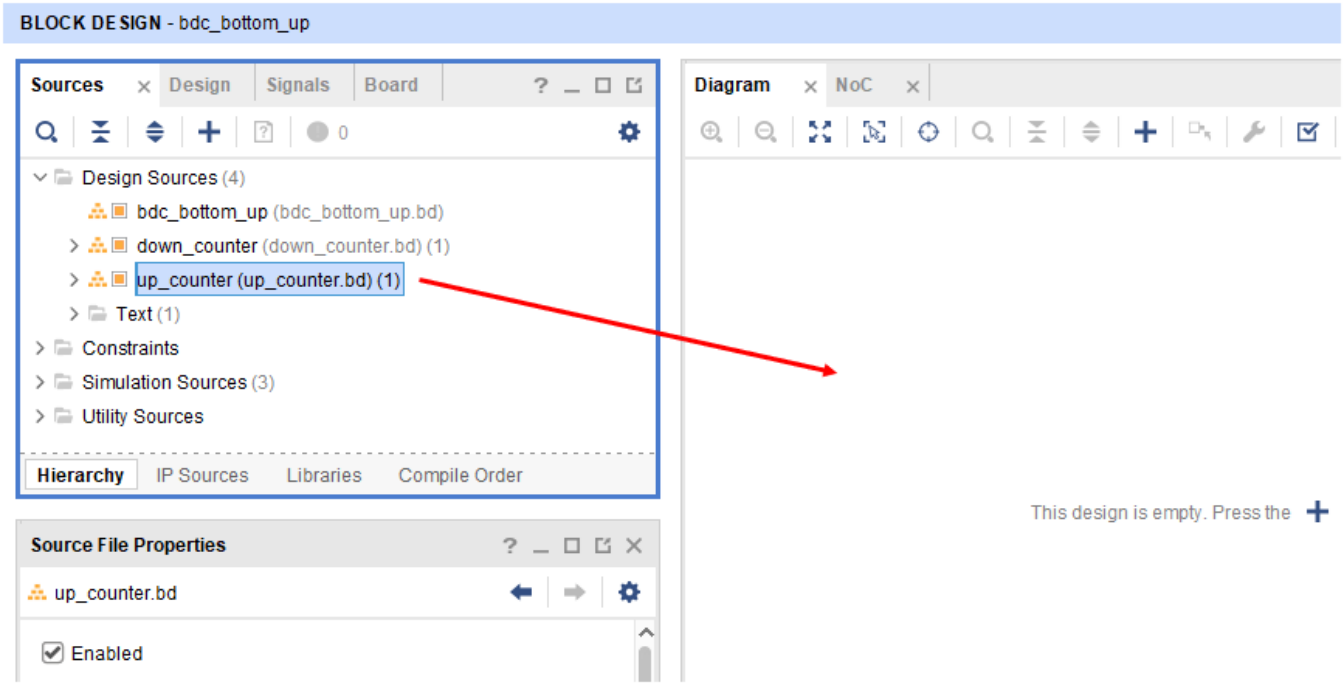
Figure 42: **Create a Block Design Container From a Level of Hierarchy**



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 43: **Bottom-Up Flow in a Block Design**

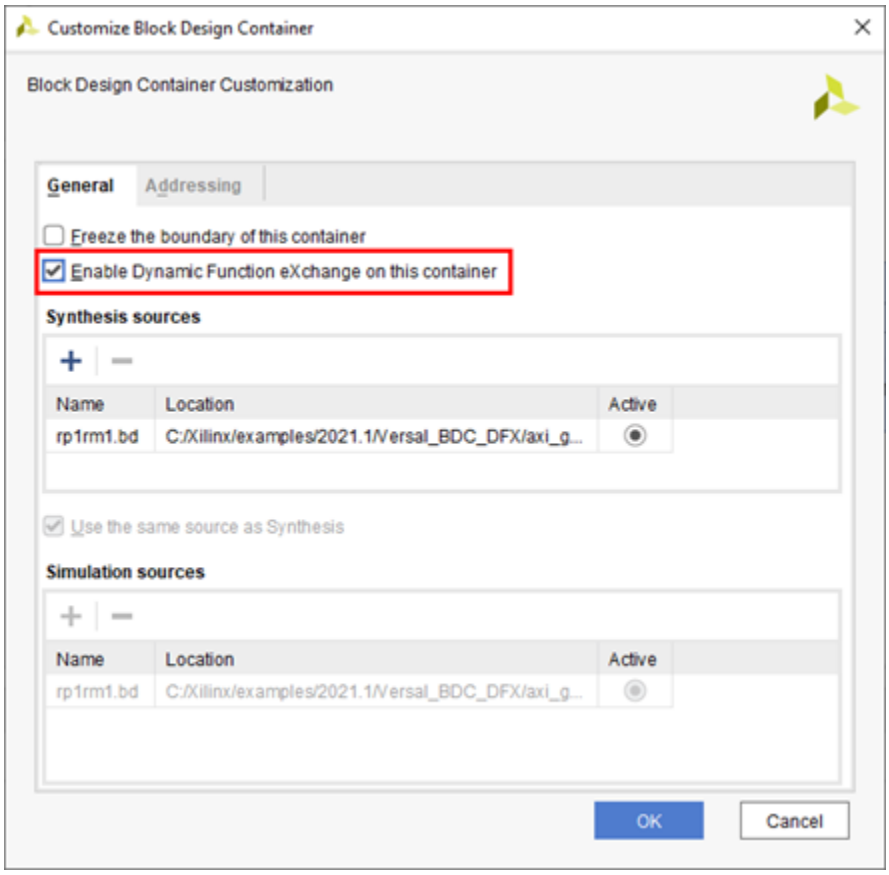


TIP: *This can be done on a blank canvas or one that already contains IP for the static part of the design.*

Turn 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 customization GUI. Check the box labeled **Enable Dynamic Function eXchange (DFX)**. The current block design is now a RM, as are any new design sources added to this BDC.

Figure 44: **Turn a Block Design Container into a Reconfigurable Partition**



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.

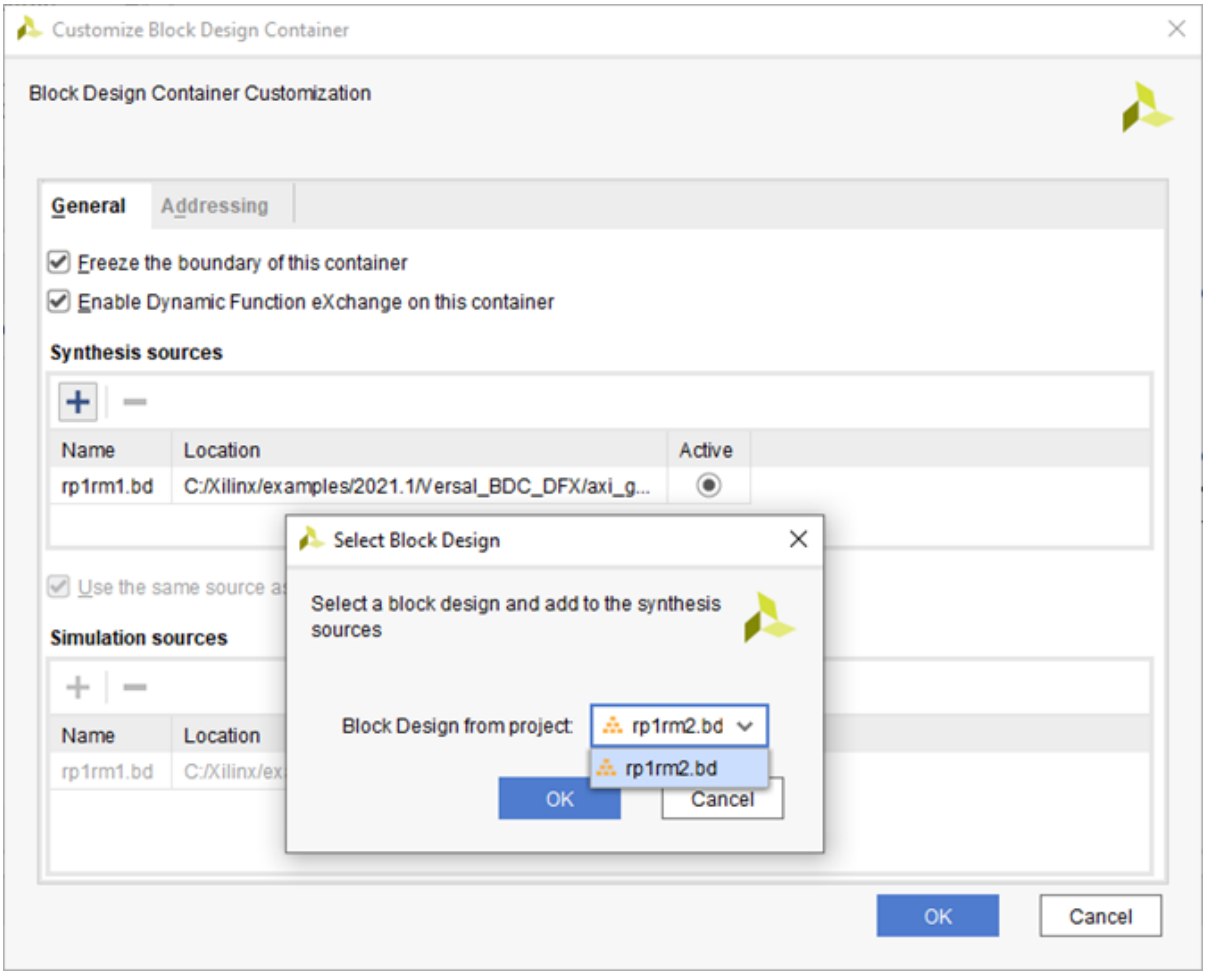
Note: A BDC can be deleted from the top-level block design, but ensure to manually recreate any portion of the design.

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:

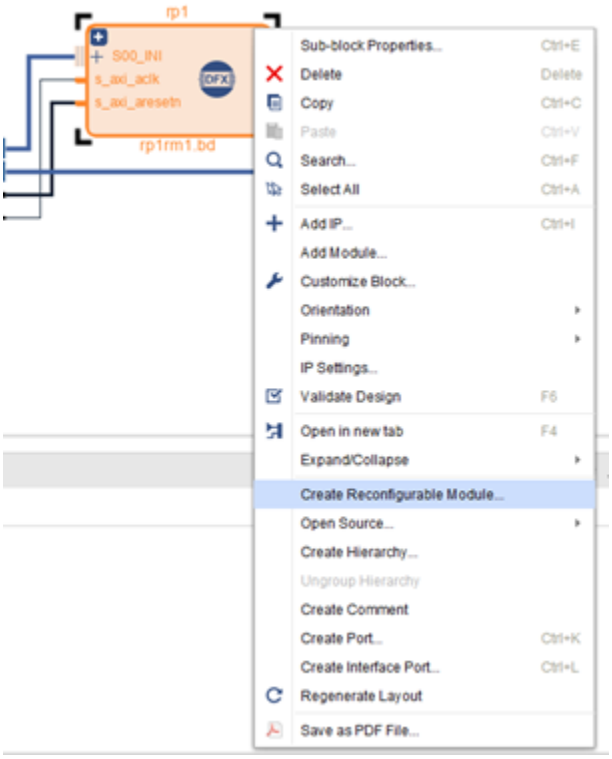
1. If a BDC exists on the static design canvas, a new BD can be added from block designs that are present in the project. Double-click on the BDC to see the list of current BDs already assigned to that BDC. Use the + icon to pick from a list of compatible BDs currently in the project and unassigned to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

Figure 45: **Add a New RM Block Design from the Project**



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.

Figure 46: **Create a New Reconfigurable Module from a BDC**



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.*

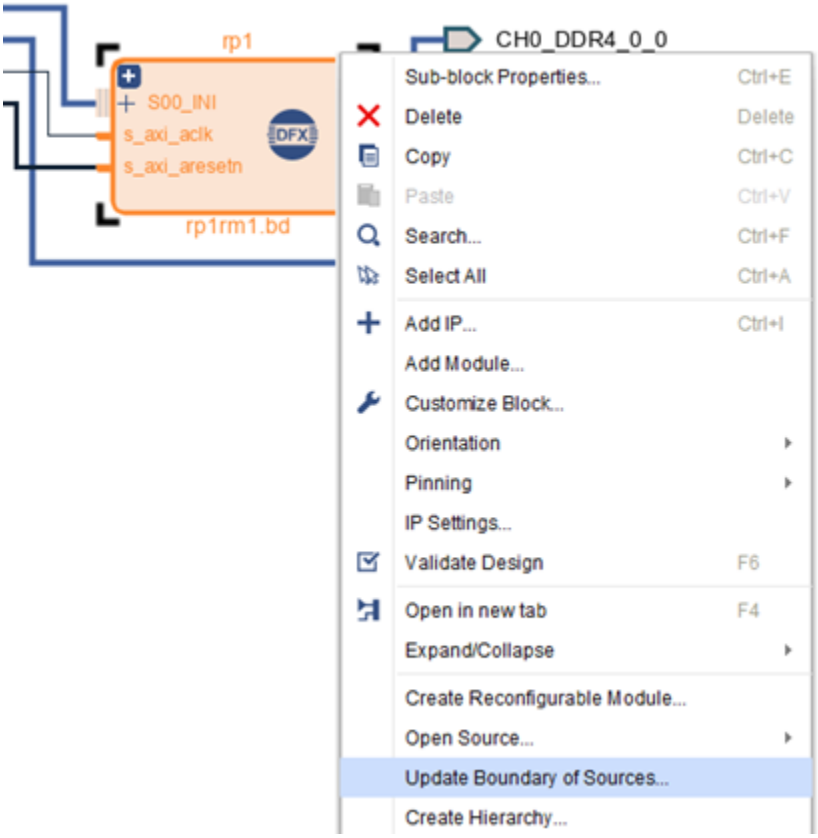
```
[BD 41-2125] There are boundary differences for </RP_inst> between 'rm1.bd'
and 'rm2.bd':
* Port with name 'reset' exists in rm1.bd but not in rm2.bd
[BD 41-2315] Block Design Container </RP_inst> is enabled for Dynamic
Function eXchange,
which means that all the Synthesis sources should settle upon 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 will now show new and modified ports. Make connections to complete the top-level block design. If you validate the BD, you will receive the **Critical Warnings** listed above, as expected.
4. Right-click on the BDC to select **Update Boundary of Sources**. Use the pulldown menu to select the BD to be used as the master instance. All ports from this BD will be echoed in all BDs selected in the main body of the dialog box. Click **OK** to apply the changes.

Figure 47: **Update Port Boundaries within a BDC**



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

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

***Note*:** Ensure to define how to use new ports.

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: *In most cases, the Auto setting is advised.*

- If set as **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.

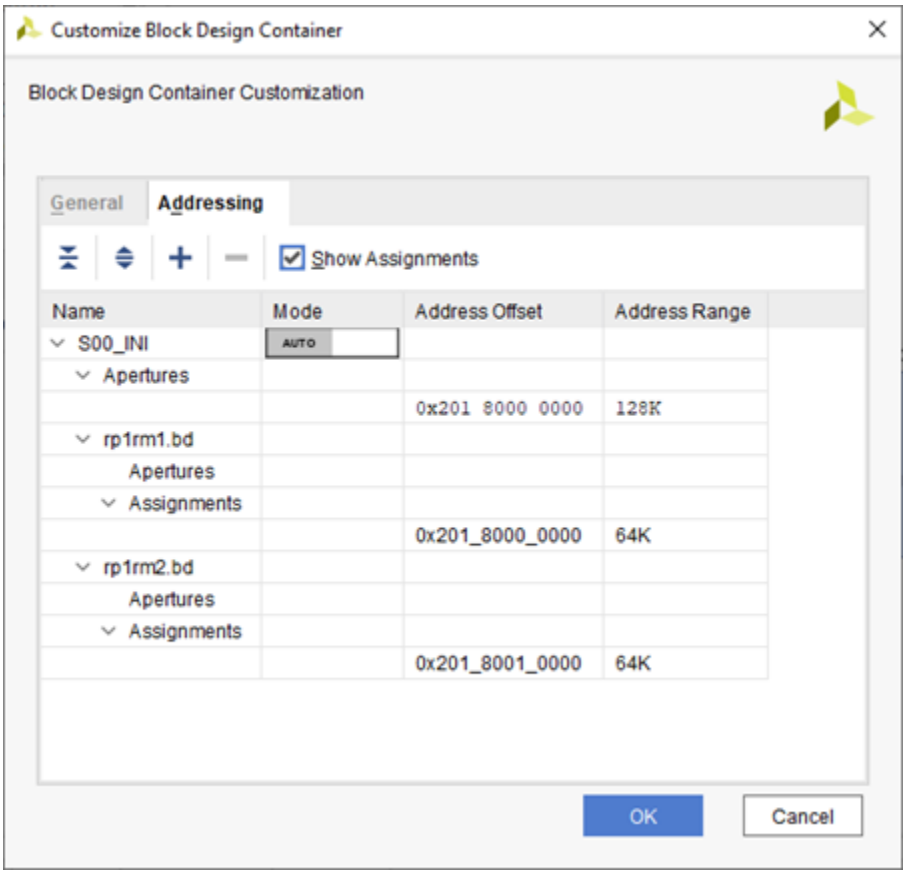
  If your static design has already been implemented with certain auto-computed apertures, adding a new RM source to the BDC will cause IP integrator to re-compute those apertures. If the calculated apertures are different than the previously computed apertures, the static result will be marked out of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

- If set as **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 **Create Reconfigurable Module** command for a BDC will inherit the BDC-specified apertures. This is more relevant to the top-down flow.

Save BD 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**



Prepare 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:

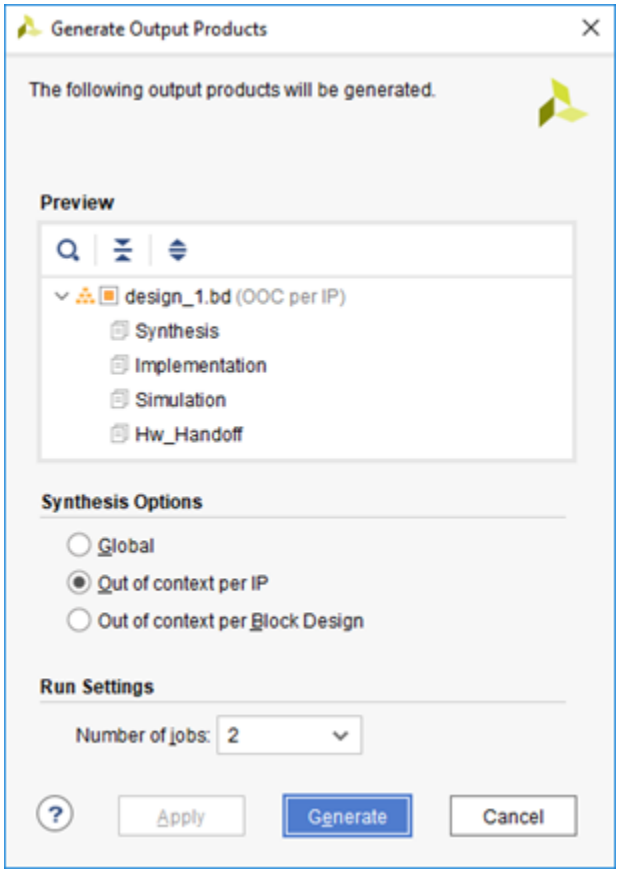
1. Generate a top-level wrapper for the design
2. Generate the RTL and IP for synthesis

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 Vivado 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**

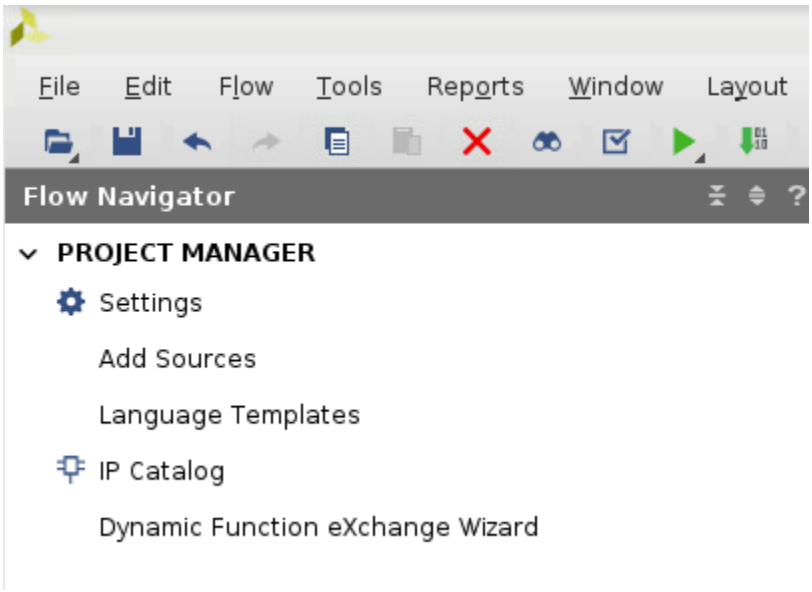


Running the Dynamic Function eXchange Wizard

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.

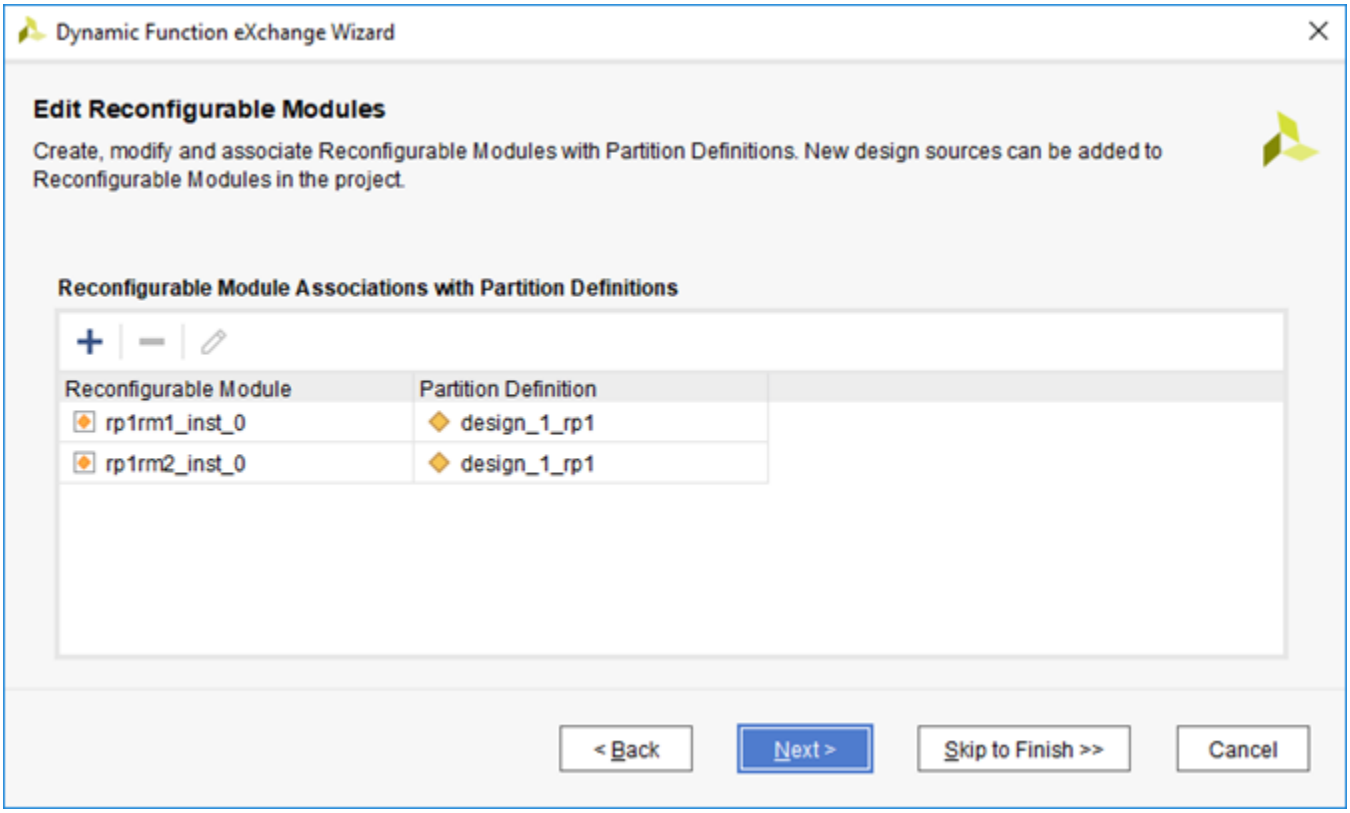
Click on **Dynamic Function eXchange Wizard** in the **Flow Navigator**, or by selecting that option under the **Tools** menu.

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.

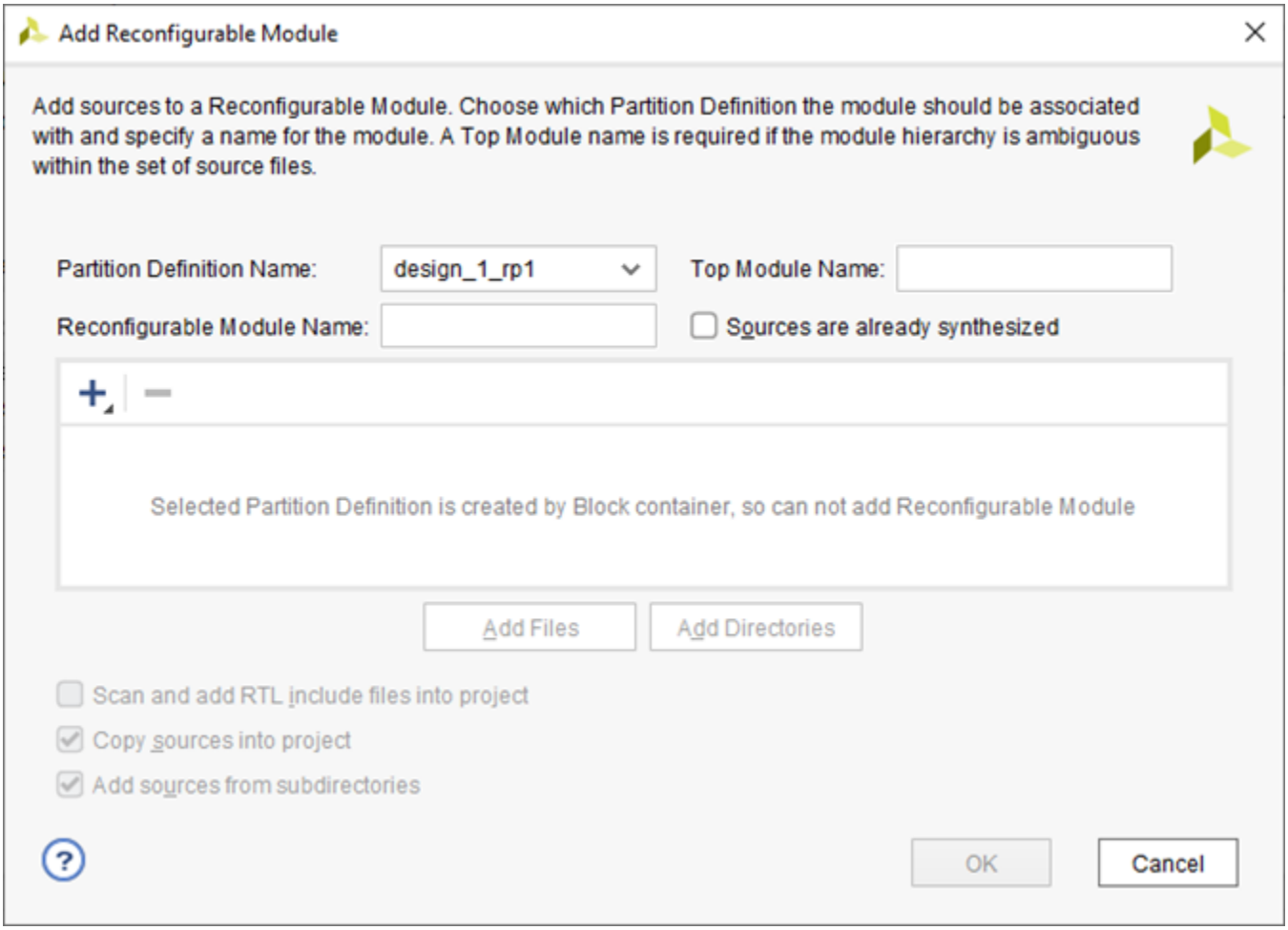
Figure 51: **List of Reconfigurable Modules within the DFX Wizard**



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.

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

Figure 52: **Add Reconfigurable Module**



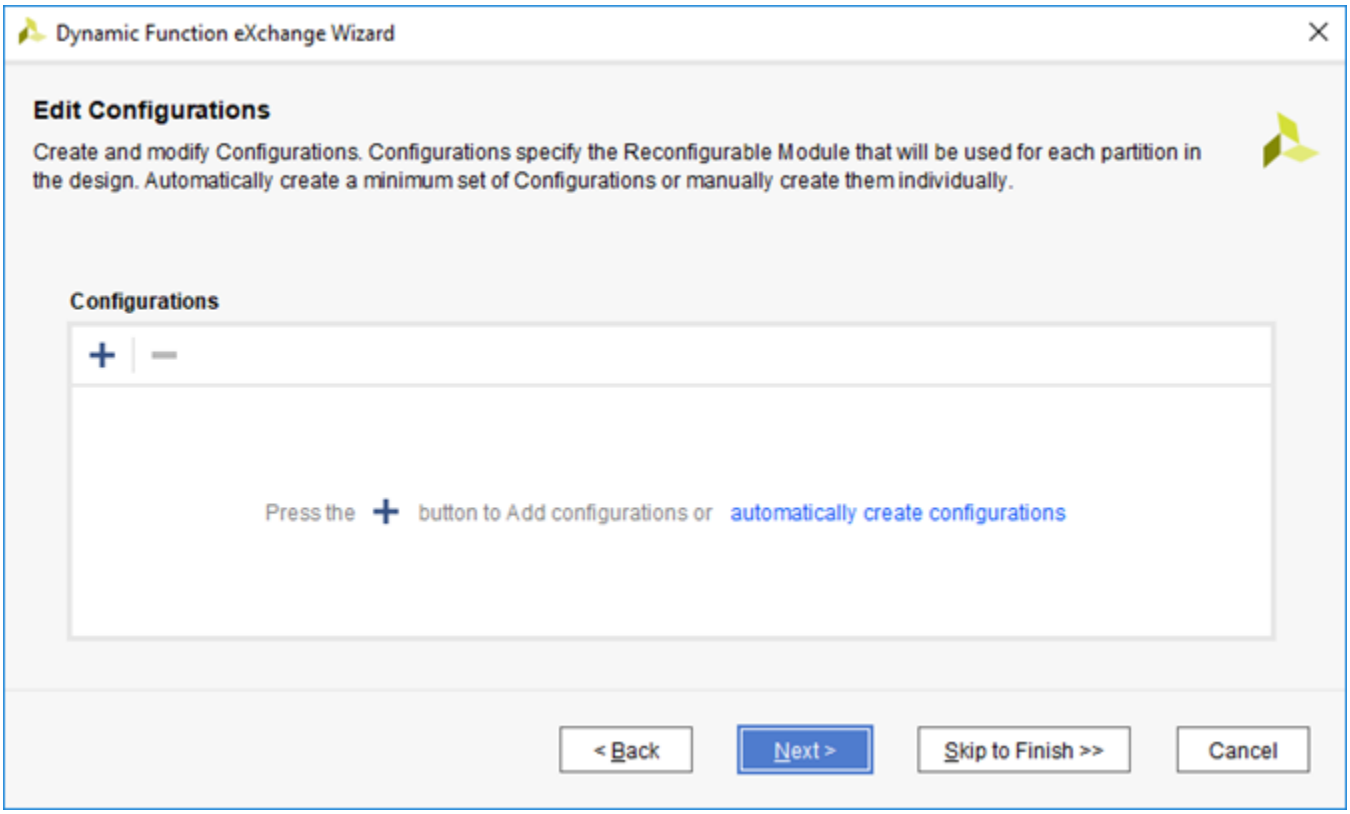
Editing Configurations

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.*

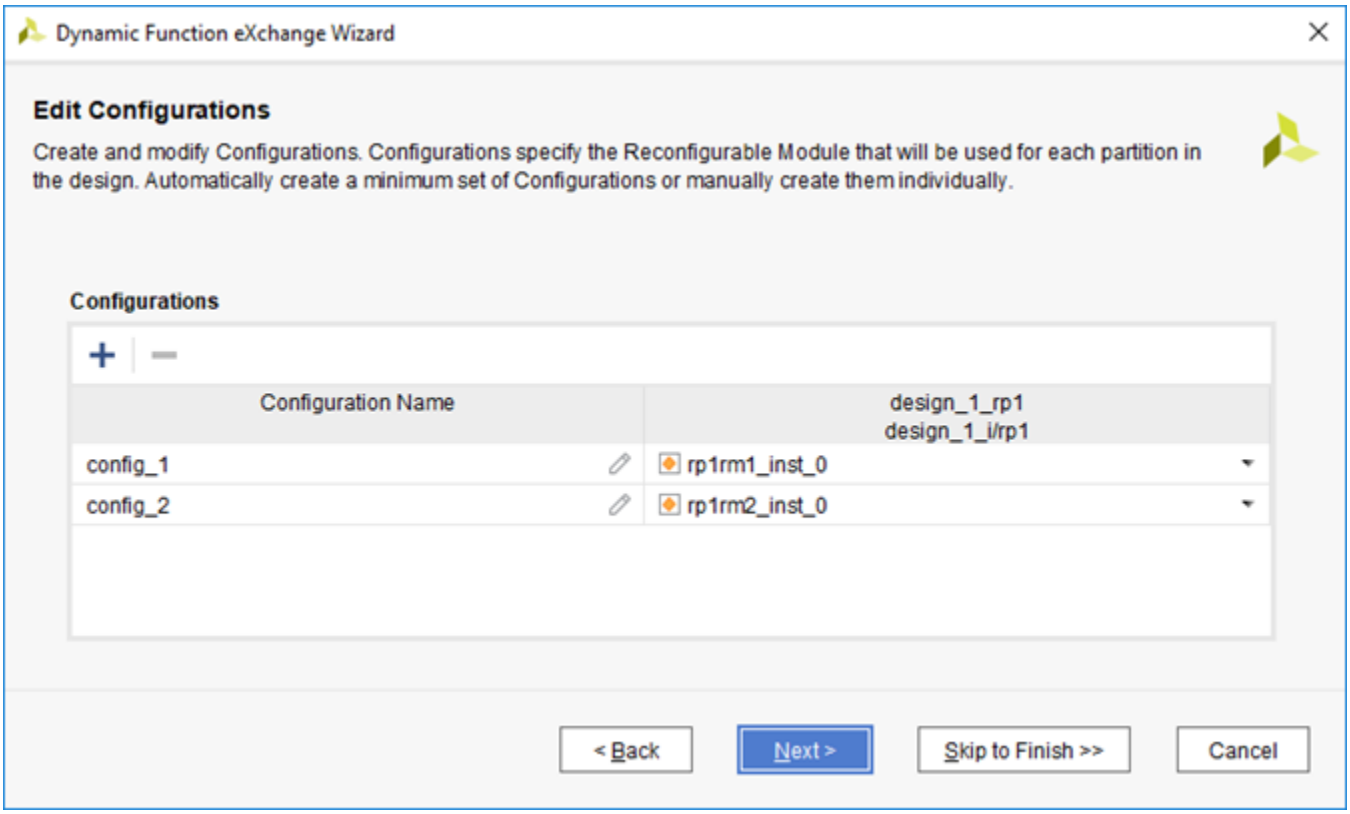
Figure 53: **Edit Configurations before Creating Any Configurations**



If one BDC has more RMs than another, a graybox 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 may be created if desired.

TIP: *Graybox modules are different than black box modules, as they are not truly empty. Graybox 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. Vivado creates these by calling* `update_design -buffer_ports` *on selected modules.*

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

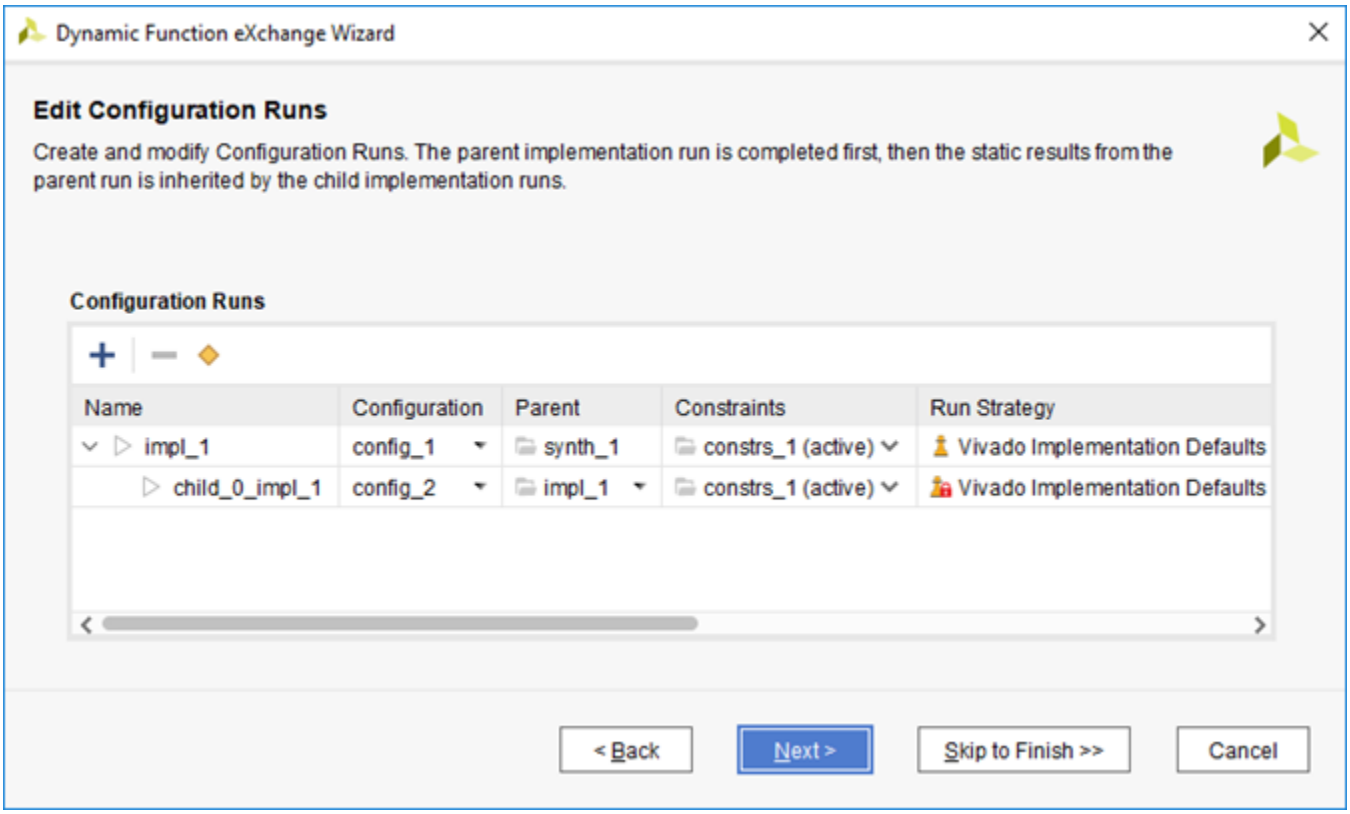


***Note*:** If one RM is used in more than one Configuration, the implementation results may be different, as place and route is performed each time, but only if the RM was initially implemented in a child run. RM implementation results will be reused if they were originally done in the parent configuration. 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. Just like the Configurations themselves, Vivado 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.

Figure 55: **Automatically Generated Configuration Runs**



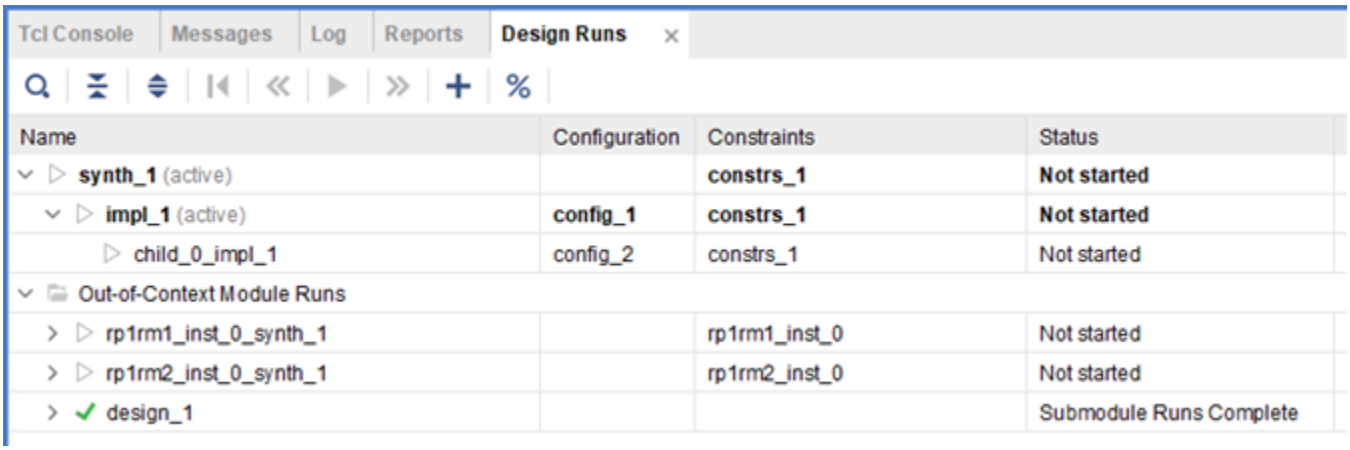
This structure assumes that the first configuration is the most critical or challenging. Users are free to change the parent-child relationship by setting that value in the Parent column. A Parent of a synthesis run (**synth_1** here) indicates the Configuration (most notably the static part) will be 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 will create compatible full and partial bitstreams. This is managed in the Vivado DFX Project flow by establishing a parent-child relationship for related Configurations.*

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 56: **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 the **Finish** button is clicked, so you can move forward or back through the screens, making adjustments as needed.

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.
- Abstract Shell for any architecture in IP integrator or RTL project mode.

Known Issues and Limitations

- When using the DFX Decoupler in IP integrator, when the Interface is created and the VLNV is subsequently changed, the VLNV property of the port pins does not changed. Consequently, IP integrator does not allow connections to the newly changed ports, since 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

XILINX

Chapter 5

Design Considerations and Guidelines for All Xilinx Devices

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

To take advantage of the dynamic reconfiguration capability of Xilinx FPGAs, you must analyze the design specification thoroughly, and consider the requirements, characteristics, and limitations associated with PR 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 Xilinx 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

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

These four IP for Dynamic Function eXchange are currently available in Vivado and can be used for any Xilinx® device that supports Dynamic Function eXchange in Vivado. As of Vivado 2020.1, these IP now have the DFX terminology within the name and throughout the IP, but are functionally equivalent to their Partial Reconfiguration named predecessors. You should use the IP upgrade feature to transition any existing PR IP to DFX IP. See the product guides for each IP for more information.

- **Dynamic Function eXchange 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 run time, 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 *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947) show examples of the DFX Controller IP in a sample design.

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

- **Dynamic Function eXchange 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.

- **Dynamic Function eXchange 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.

- **Dynamic Function eXchange 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 Chapter 12: 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 (v1.8)* (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. Since the static side of the interface is locked after the initial configuration, the tools are unable to adjust for these changes in later configurations. Xilinx 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 Xilinx 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.

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 Xilinx 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).

A Partial Reconfiguration Decoupler IP is available, allowing users to insert MUXes to efficiently decouple AXI4-Lite, AXI4-Stream, and custom interfaces. More information about the DFX Decoupler IP is available on the Xilinx website.

Black Boxes

You can implement an RP as a *pseudo* black box, referred to in Vivado as a graybox. 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 logic 0 (ground). If it is necessary to drive a logic 1 (V_{CC}) 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 a route-thru (output is 0). Setting this property to '`1`' configures the LUT as an inverter (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>]
```

The graybox has no user logic (just the tool-inserted LUT1 buffers). The graybox 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 grayboxes 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 will be 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 this link in *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906). Note that 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 will be missing from these checkpoints. The most complete information is not available until after `opt_design` when all the linking and expansion has been done. We advise 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 may 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 this link in *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, Xilinx 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.

> **IMPORTANT!** *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 caving of the RM if the associated I/O buffer is part of an RP.*

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 Chapter 9: 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.

Xilinx 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 (v1.0)* (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 Application Note (v1.0)* (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.*

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.

XILINX

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 Zynq®-7000 SoC devices.

To take advantage of the Dynamic Function eXchange capability of Xilinx® 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.)

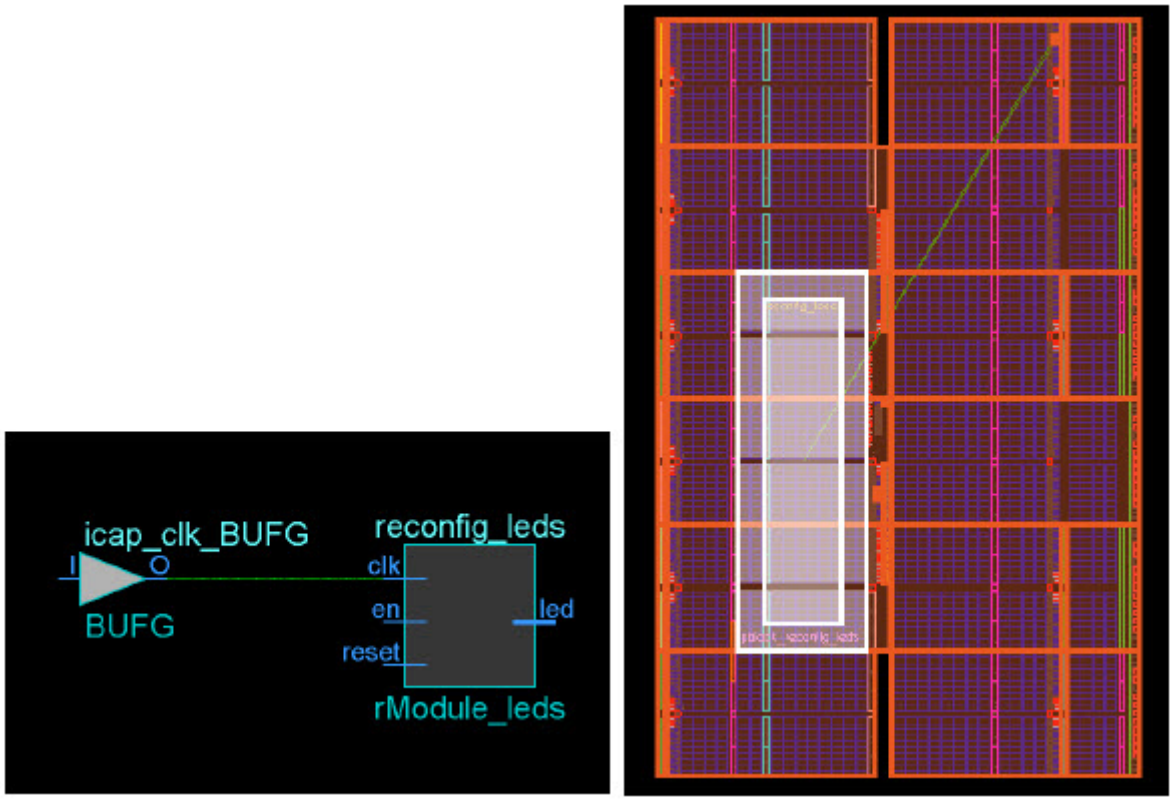
Global Clocking Rules

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 Global Clocking Rules, `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 57: **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.

Creating Pblocks 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.

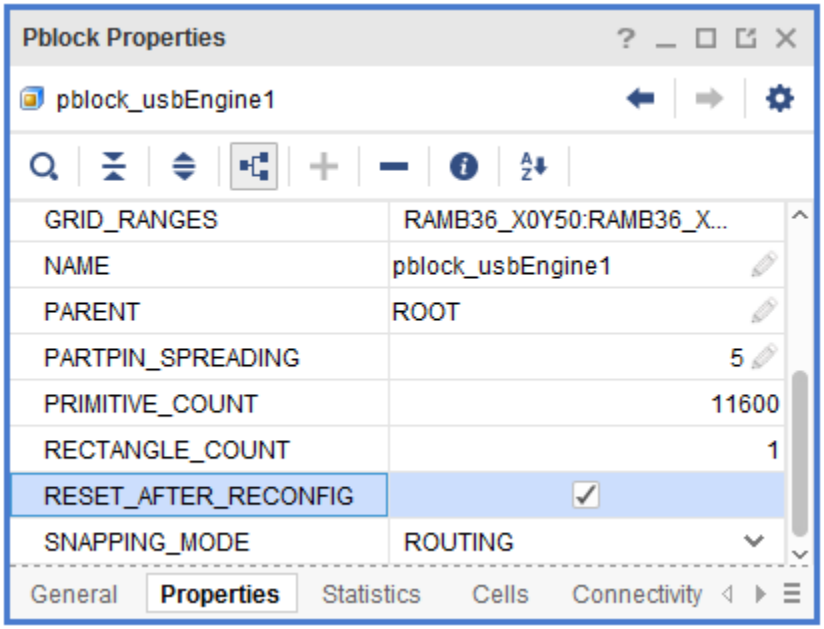
Automatic Adjustments for Reconfigurable Partition Pblocks

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 Automatic Adjustments for Reconfigurable Partition Pblocks.

Figure 58: **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.

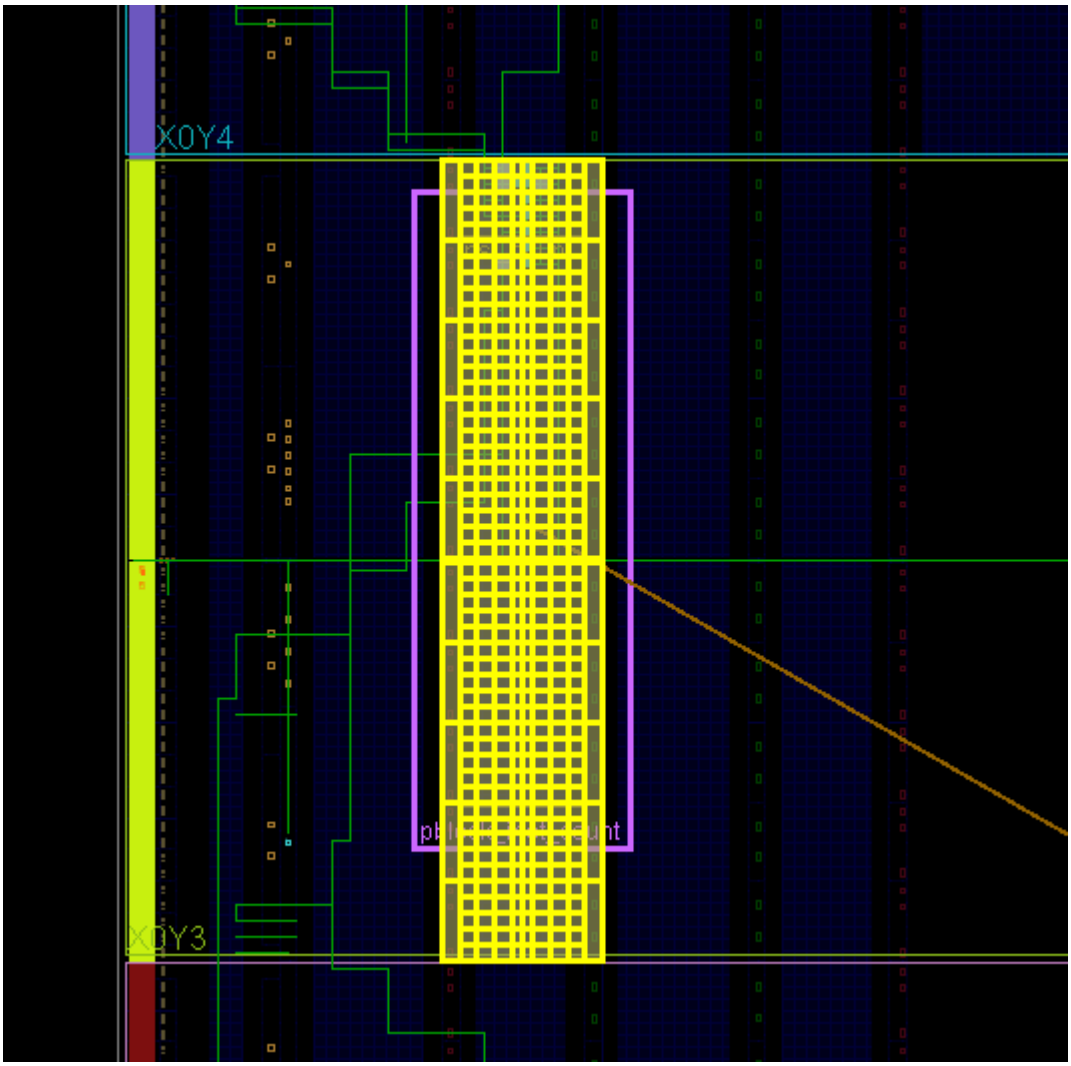
Table 11: **SNAPPING_MODE Property Values for 7 Series Devices**

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.

Automatic Adjustments for Reconfigurable Partition Pblocks 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 59: **Original and Derived Pblocks Using SNAPPING_MODE**

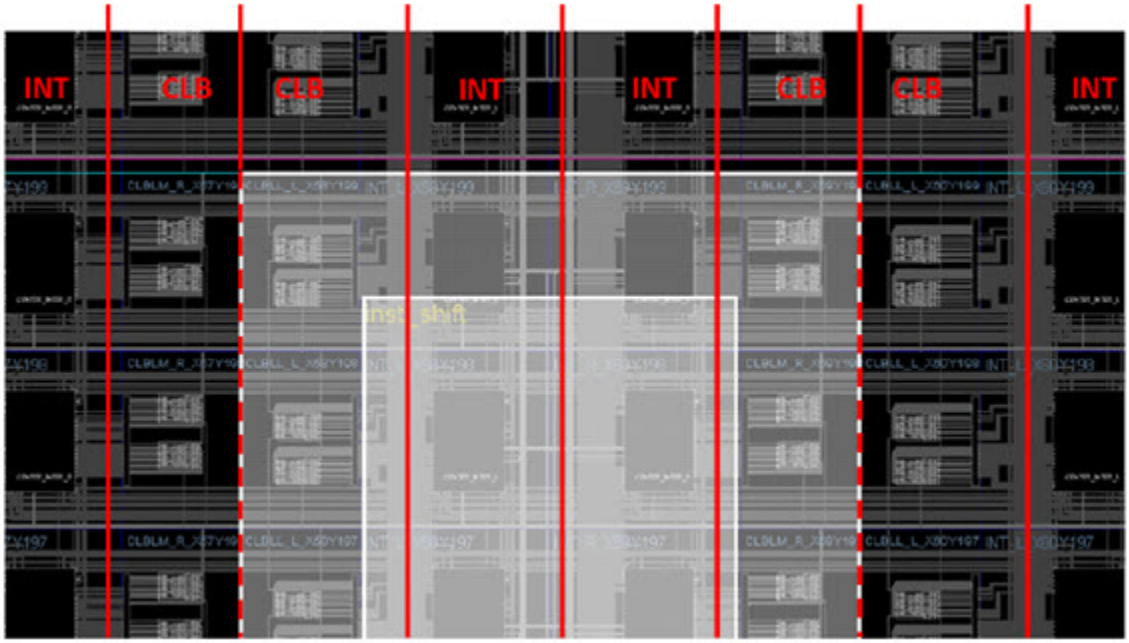


Creating Reconfigurable Partition Pblocks Manually

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 Creating Reconfigurable Partition Pblocks Manually, note that 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 60: **Optimal - Reconfigurable Partition Pblock Splitting CLB-CLB on Both Left and Right Edges**



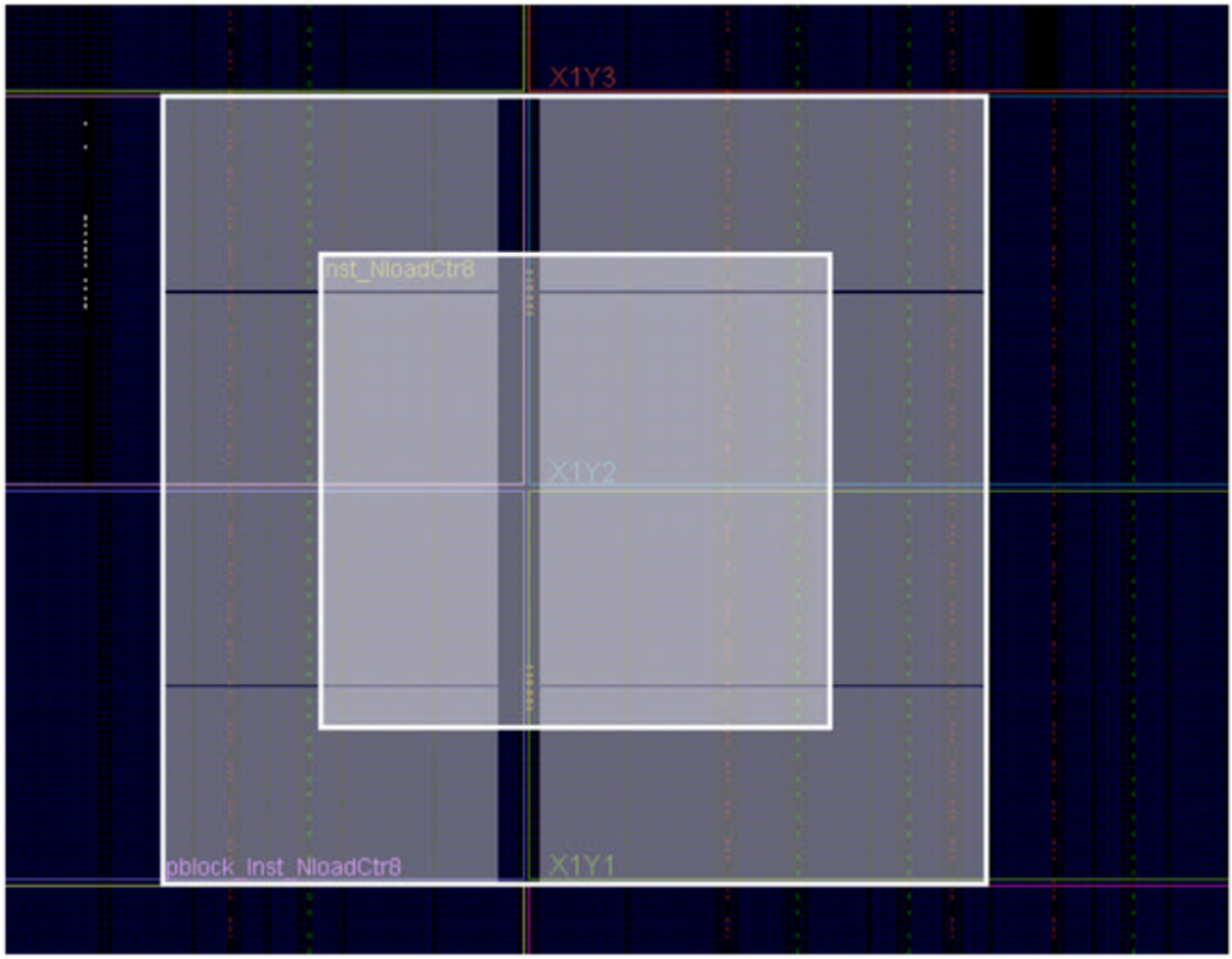
The RP Pblock must include all reconfigurable element types within the shape drawn. In other words, if the rectangle selected encompasses CLB (Slice), block RAM, and DSP elements, all three types must be included in the Pblock constraints. If one of these is omitted, a DRC is triggered with an alert 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.

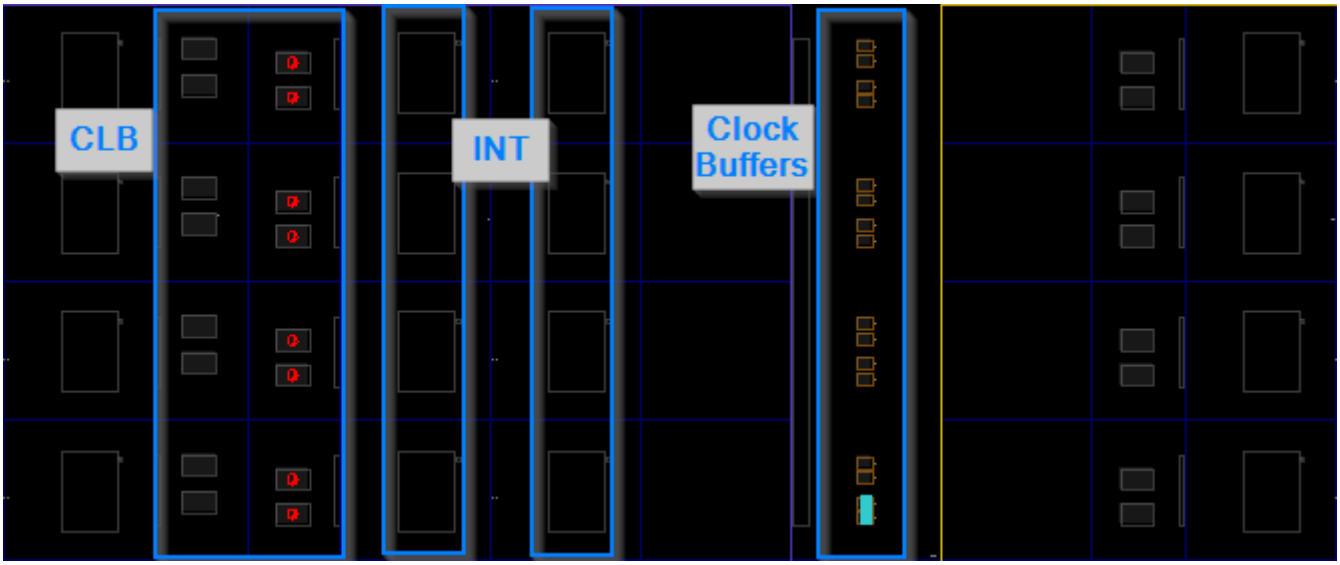
The one exception to this behavior is around the clock column. If a violation occurs at the clock column boundary, `PROHIBIT` constraints are generated for the RM side of the violation (typically SLICE prohibits), but the clocking resources do not get prohibit constraints and are still available to the 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 Creating Reconfigurable Partition Pblocks Manually 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 Creating Reconfigurable Partition Pblocks Manually. There is violation caused by spanning this clock column but the resources can still be used by the static logic.

Figure 61: **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 Creating Reconfigurable Partition Pblocks Manually. 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 Creating Reconfigurable Partition Pblocks Manually, 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 62: **Prohibited Sites in a Checkpoint**



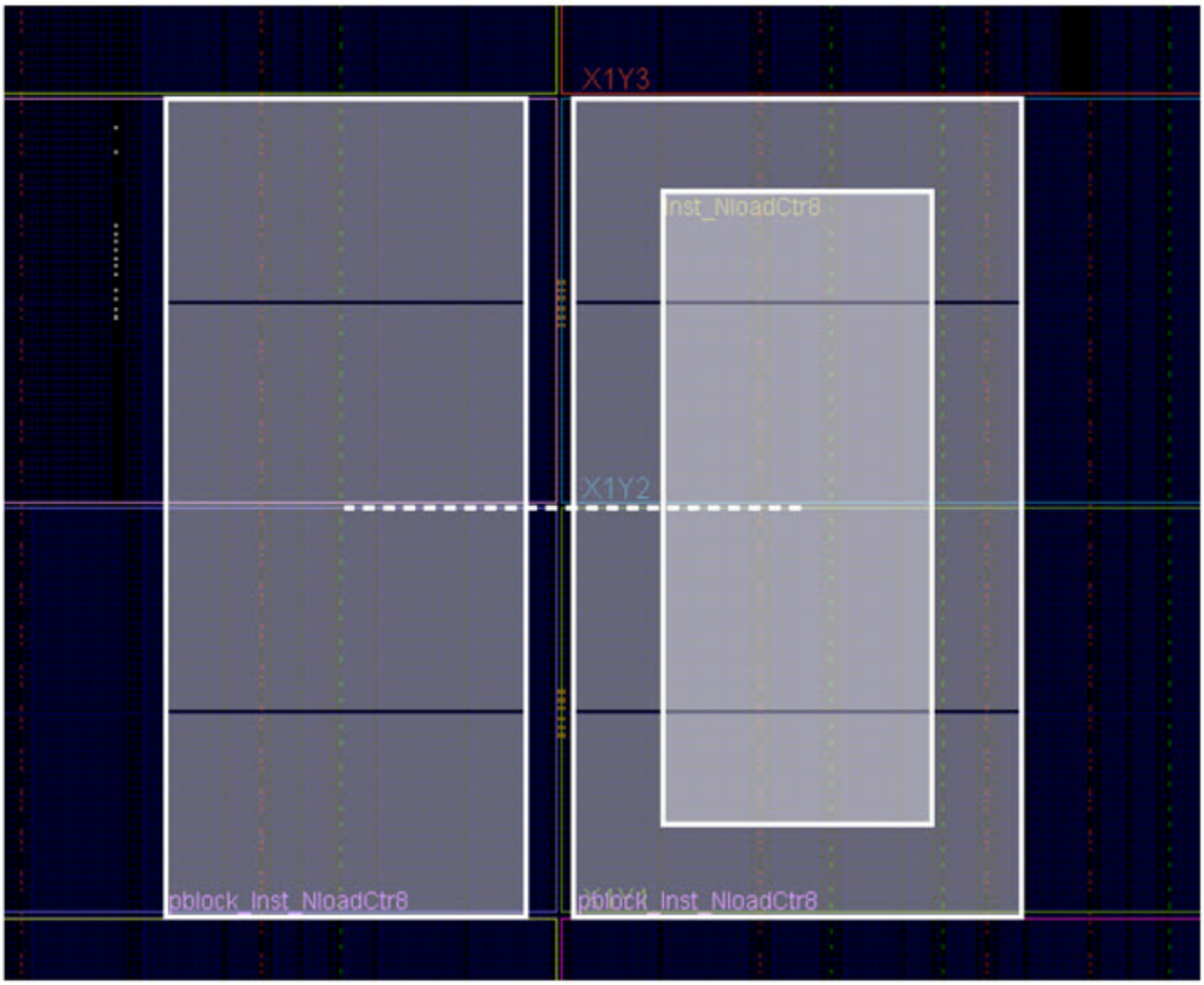
If a back-to-back violation prohibits sites that are needed for the design (that is, ICAP or BSCAN sites), a placement error is issued, stating that not enough sites are available in the device.

```
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 Creating Reconfigurable Partition Pblocks Manually, or use the Pblock `SNAPPING_MODE` property.

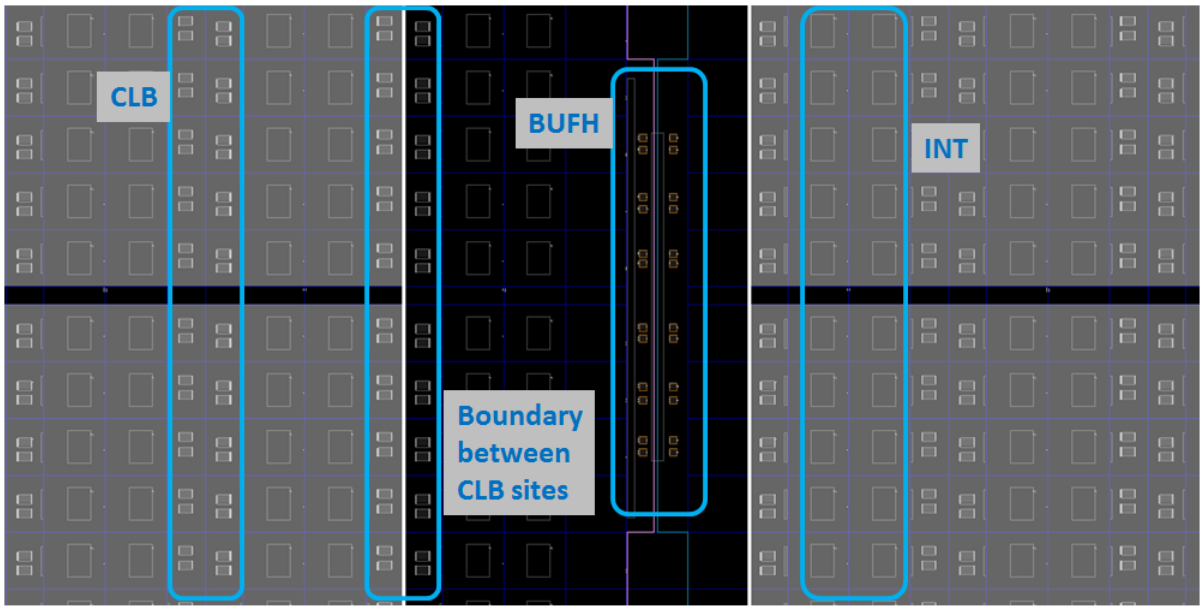
In general, spanning non-reconfigurable site types (such as IOB, configuration, or clocking columns) should be avoided whenever possible. If the Pblock must span one of these, the clocking column is the least risky choice, owing to its special nature (described previously). Use `SNAPPING_MODE ROUTING` to cross this boundary as efficiently as possible.

Figure 63: **Multiple Pblock Rectangles that Avoid Non-Reconfigurable Resources**



Creating Reconfigurable Partition Pblocks Manually 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 64: **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

Xilinx 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)

Xilinx 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 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 Chapter 9: 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.

XILINX

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 UltraScale and UltraScale+ devices.

To take advantage of the Dynamic Function eXchange capability of Xilinx® 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:

- BSCAN
- CFG_IO_ACCESS
- DCIRESET
- EFUSE_USR
- FRAME_ECC
- ICAP
- MASTER_JTAG
- STARTUP
- USR_ACCESS

Creating Pblocks for UltraScale and UltraScale + Devices

As part of improvements to the UltraScale 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). Because adjacent sites share a routing resource (or Interconnect Tile) in UltraScale, a PU is defined in terms of pairs.

Examples of some of the minimum PU that can be reconfigured based on the site types include:

- CLB PU: 2 adjacent CLBs and the shared interconnect
- Block RAM PU: 1 block RAM/FIFO, the 5 adjacent CLBs and the shared interconnect
- DSP PU: 1 DSP, the 5 adjacent CLBs, and the shared interconnect
- IOB PU: The IO 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
- GT PU: A full GT quad (4 GT_CHANNEL and 1 GT_COMMON), the adjacent 60 CLBs and the shared interconnect

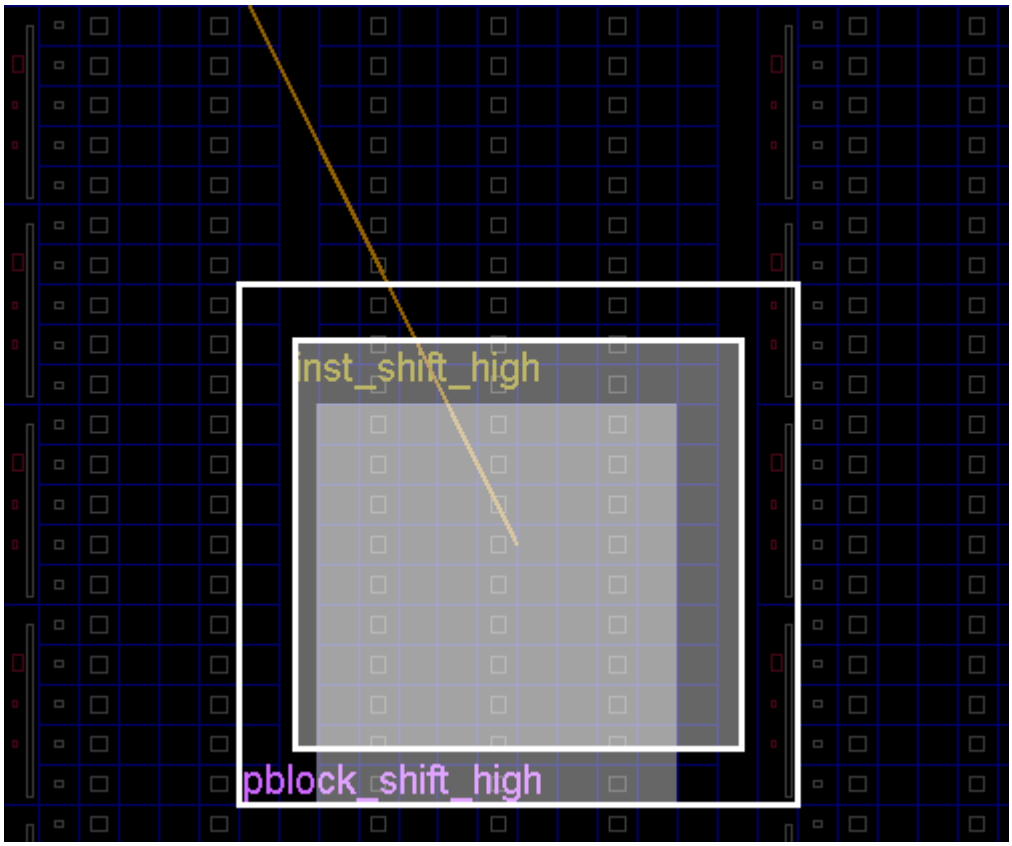
- PCIe PU: 1 PCIE40E4 or PCIE4CE4, 120 adjacent CLBs (60 on each side) and the shared interconnect
- CMAC PU: 1 CMACE4, 120 adjacent CLBs (60 on each side) and the shared interconnect
- Interlaken PU: 1 ILKNE4, 120 adjacent CLBs (60 on each side) and the shared interconnect
- CONFIG PU: 1 CONFIG_SITE, 120 adjacent CLBs (60 on each side) and the shared interconnect
  - Note: 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.
- HBM BLI PU: 1 HBM_PLI, 15 adjacent CLBs and the shared interconnect.

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 PR.

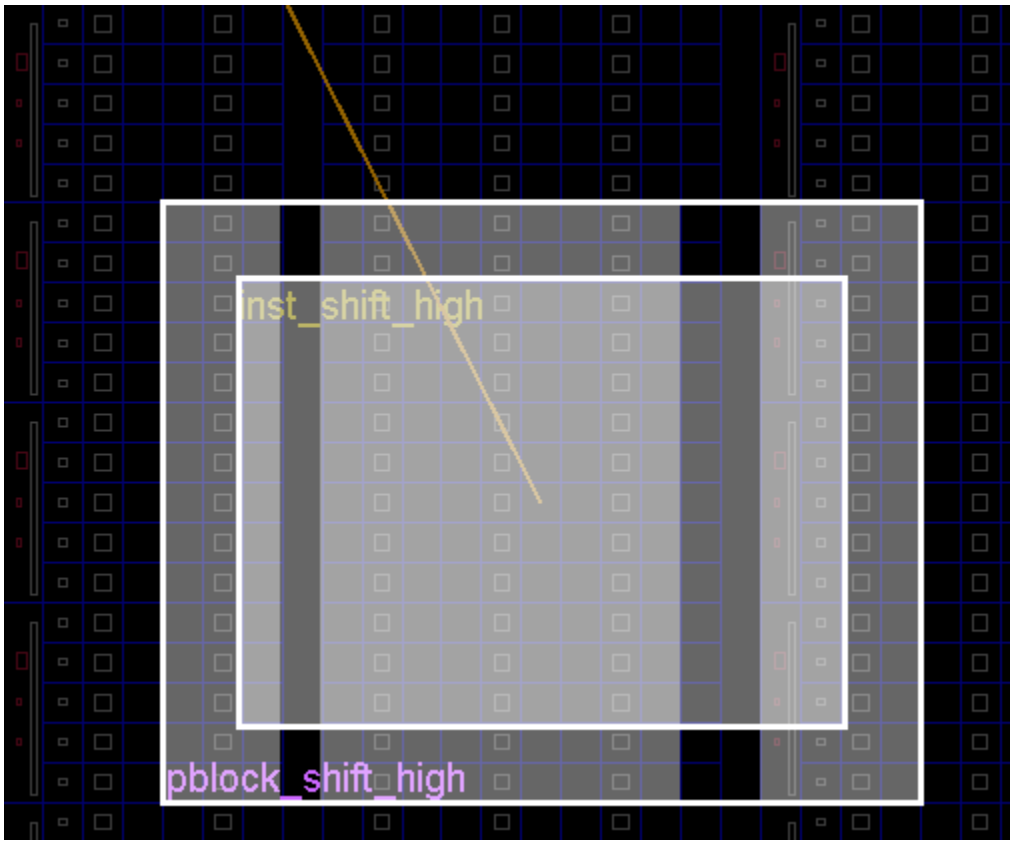
Automatic Adjustments for PU on Pblocks and Automatic Adjustments for PU on Pblocks below give an example of how `SNAPPING_MODE` adjusts the Pblock for PU alignment. In Automatic Adjustments for PU on Pblocks, despite the larger outer rectangle, only the selected tiles belong to the RP Pblock. The upper 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 65: **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. In Automatic Adjustments for PU on Pblocks 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 66: **PU Aligned Pblock**



While shading shows what is included in a reconfigurable partition (RP), you can best visualize the sites owned by a RP by using highlighting scripts that the Vivado Design Suite tools create automatically for Pblocks of RPs. 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. Source the highlighting script that was generated by the Vivado tools.

```
source ./hd_visual/<pblock_name>_AllTiles.tcl
```

Note: The scripts in the `hd_visual` directory are updated any time the Pblock constraints are processed. This includes opening a design that contains Pblocks and creating or modifying Pblocks in an open design.

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.

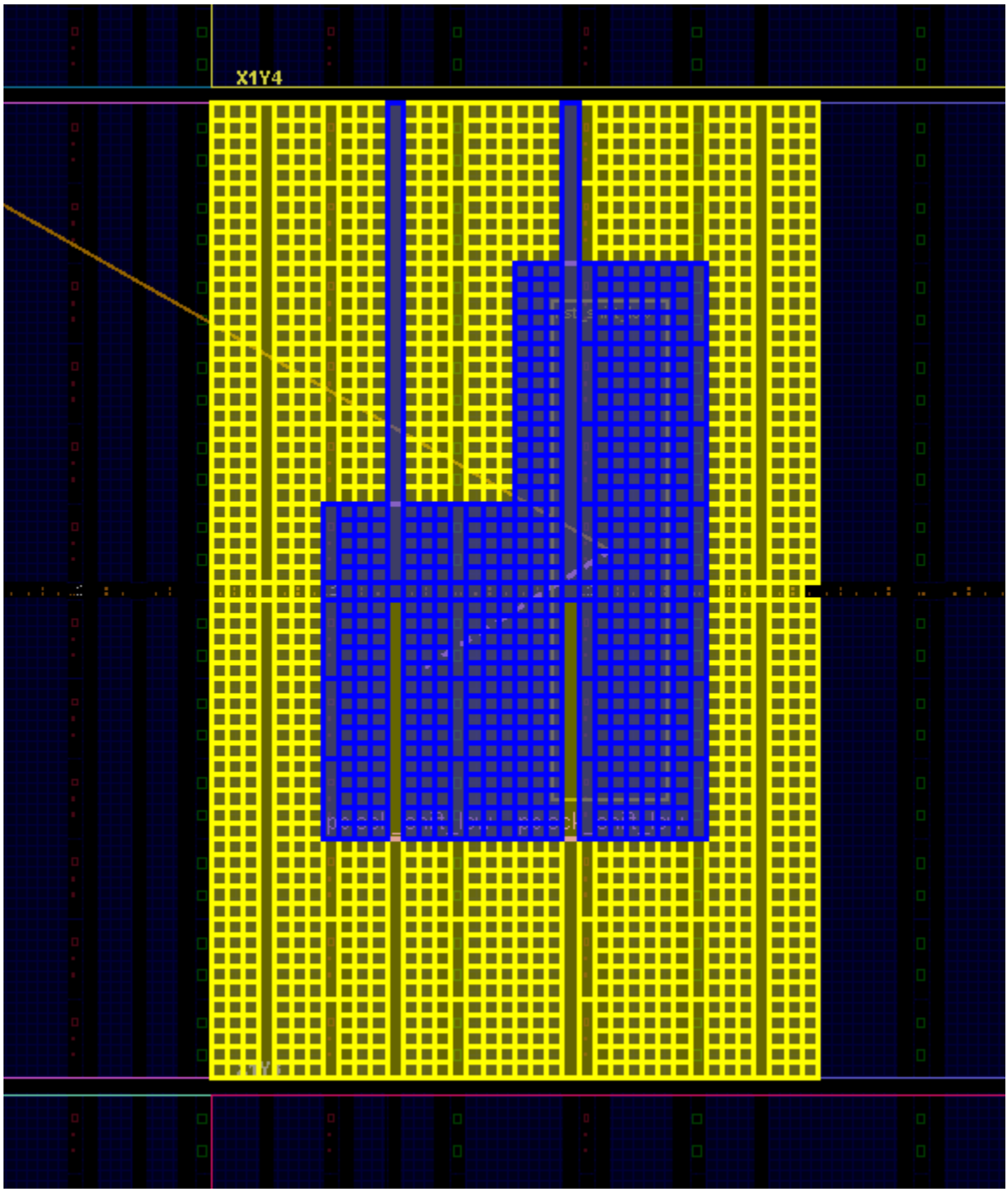
Expansion of CONTAIN_ROUTING Area

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 sourcing one of the hd_visual Tcl scripts. These are scripts that are generated automatically during the Dynamic Function eXchange flow, and can be found in the `./hd_visual` subdirectory within the current working directory. The visualization script that shows the expanded routing footprint is named `./hd_visual/<pblock_name>_Routing_AllTiles.tcl`. The expanded routing footprint is actually determined during routing, so this file is not be available until route_design completes. To see the expanded routing footprint, source this file from the Tcl Console after opening a routed design. This selects all tiles available to the router, and then selected tiles can be highlighted or marked as desired. In the figure below, the user-defined Pblock that bounds placement is shown in blue, and the expanded routing zone is shown in yellow.

```
source ./hd_visual/pblock_inst_shift_low_Routing_AllTiles.tcl
```

Figure 67: **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. Xilinx 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.*

In Vivado 2020.2, the algorithms that determine the device and design resources included in the expanded routing region were updated. The expansion region is now a bit smaller than in prior tool versions, producing smaller partial bitstreams with minimal impact on design routability. The changes made were required to support the Abstract Shell feature for UltraScale+ targets.

For DFX designs that are brought into Vivado 2020.2 for the purpose of generating only new partial bitstreams, the original routing expansion will be maintained, to preserve bitstream compatibility. In other words, if the parent configuration (the run that establishes the static design results) was implemented in Vivado 2020.1 or older and will not be reimplemented in Vivado 2020.2, the partial bitstreams for all new RMs will continue to use the older style routing expansion so these new partial bitstreams will be compatible with existing deployed static platforms.

All designs that are new in Vivado 2020.2 or whose static design will be reimplemented in 2020.2 will use the new routing expansion solution. No user intervention is required beyond simply implementing the static design through place and route. Per DFX methodology rules, all child configurations to create results for all remaining RMs must also be implemented. Therefore, all partial bitstreams will remain compatible.

IMPORTANT! *The use of Abstract Shell requires the enhanced expanded routing footprint. All DFX designs that intend to use this feature must be implemented in Vivado 2020.2, the first production release for Abstract Shell, to ensure the abstract shells are generated correctly. You cannot generate abstract shells from pre-2020.2 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.

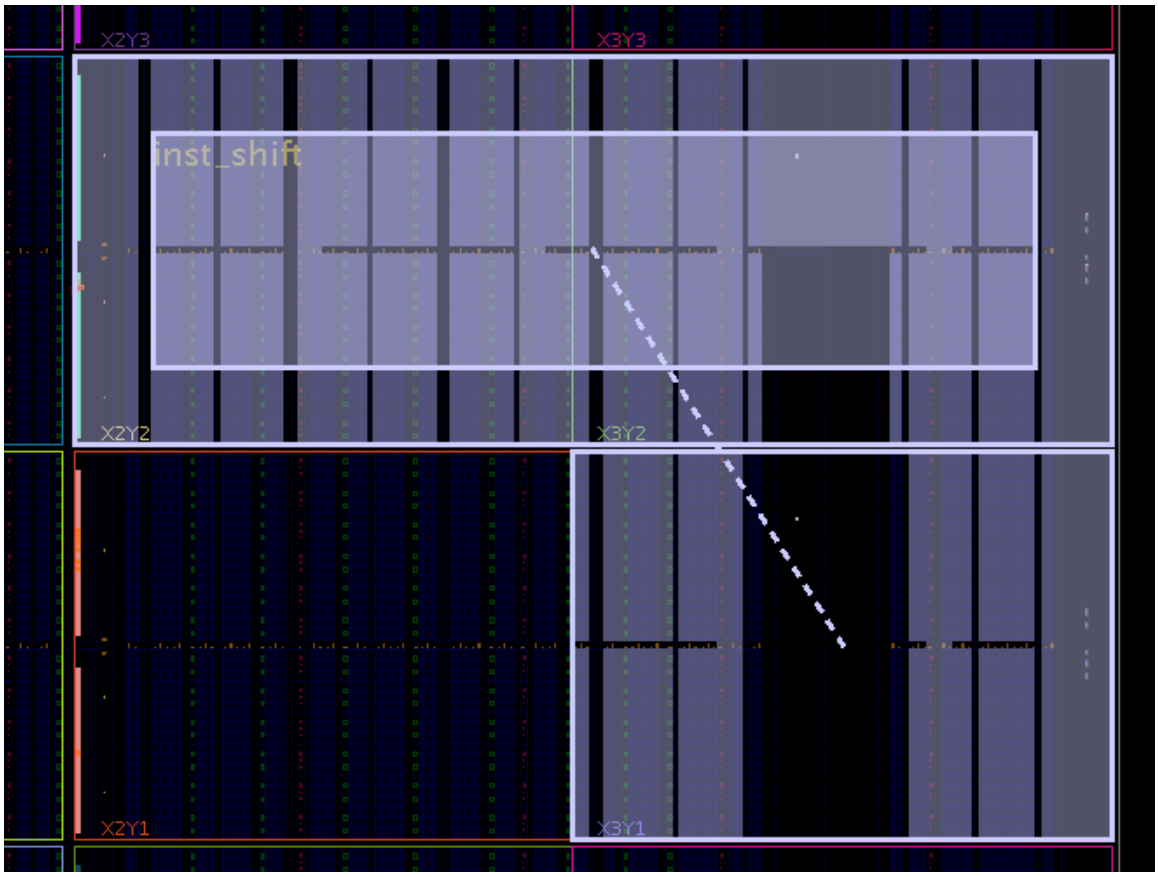
Floorplanning Rules for Clocks Inside an RP

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.

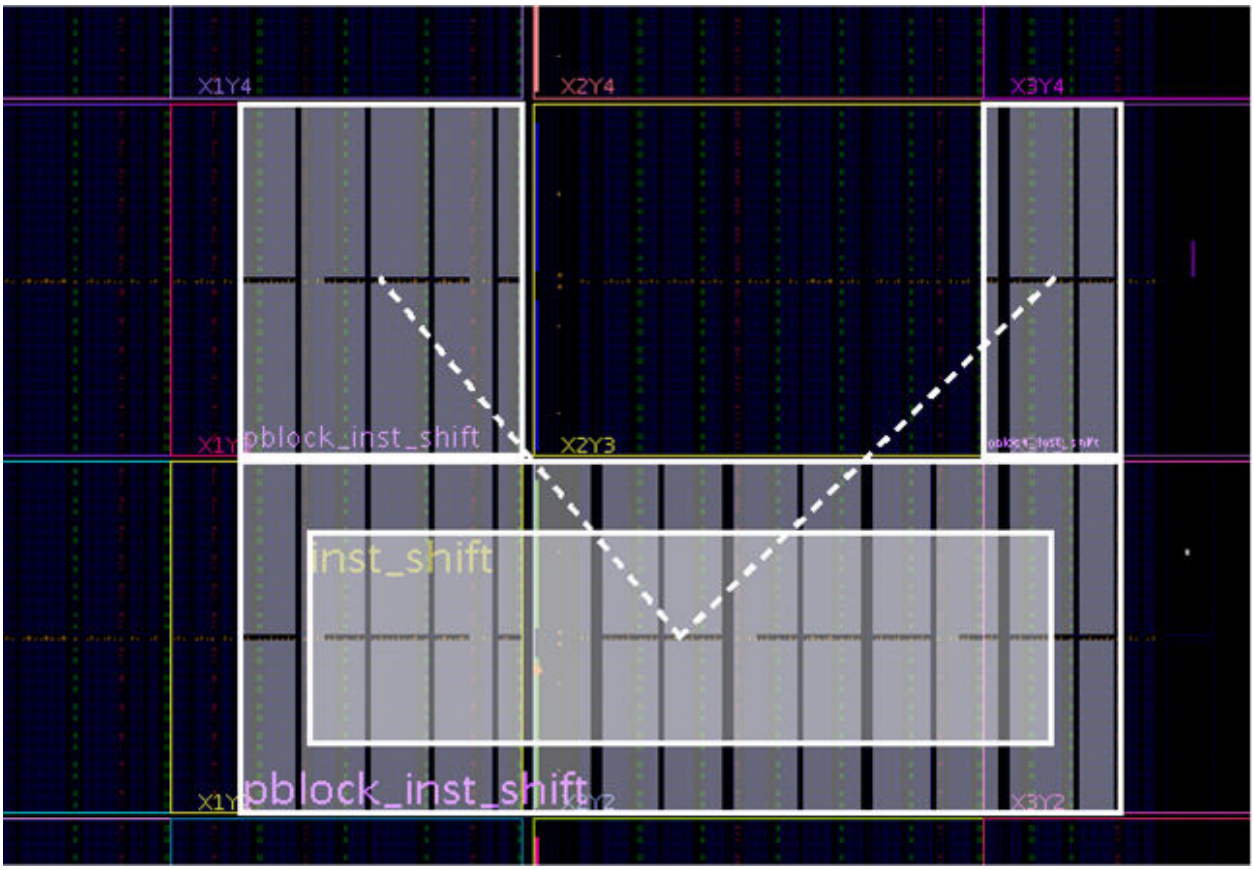
      As shown in Floorplanning Rules for Clocks Inside an RP, a CLOCK_ROOT defined in region `X2Y2` (top-left of the Pblock) would prevent routing to any loads in region `X3Y1` (bottom-right of the Pblock), because the region `X2Y1` is not available to the clock. Conversely, if the CLOCK_ROOT were defined in either `X3Y2` or `X3Y1`, no clock routing restrictions would apply.

Figure 68: **CLOCK_ROOT Restrictions on L-Shaped Pblocks**



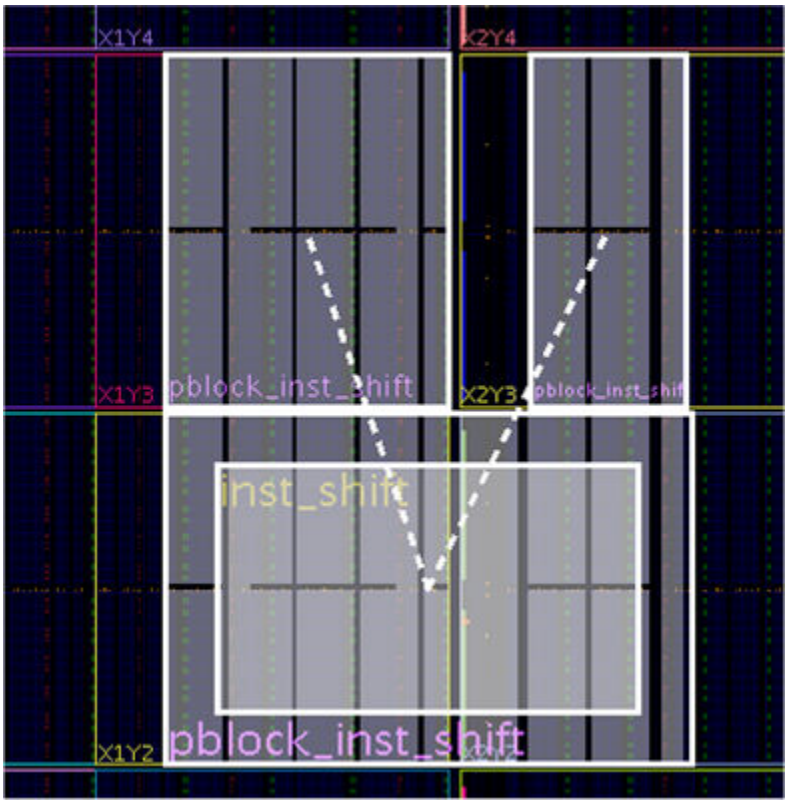
3. If a CLOCK_ROOT cannot be set to the tallest column, the loads of the clock can be contained to regions accessible by the clock using nested Pblocks within the RP region. The nested Pblock will prevent the placer from putting a load in a region that is not accessible by the clock due to irregularly shaped Pblocks.
4. Do not create U or H shaped Pblocks with large gaps that span an entire `clock_region`, as shown in Floorplanning Rules for Clocks Inside an RP below.

Figure 69: **Unsupported Pblock with clock_region Gap**



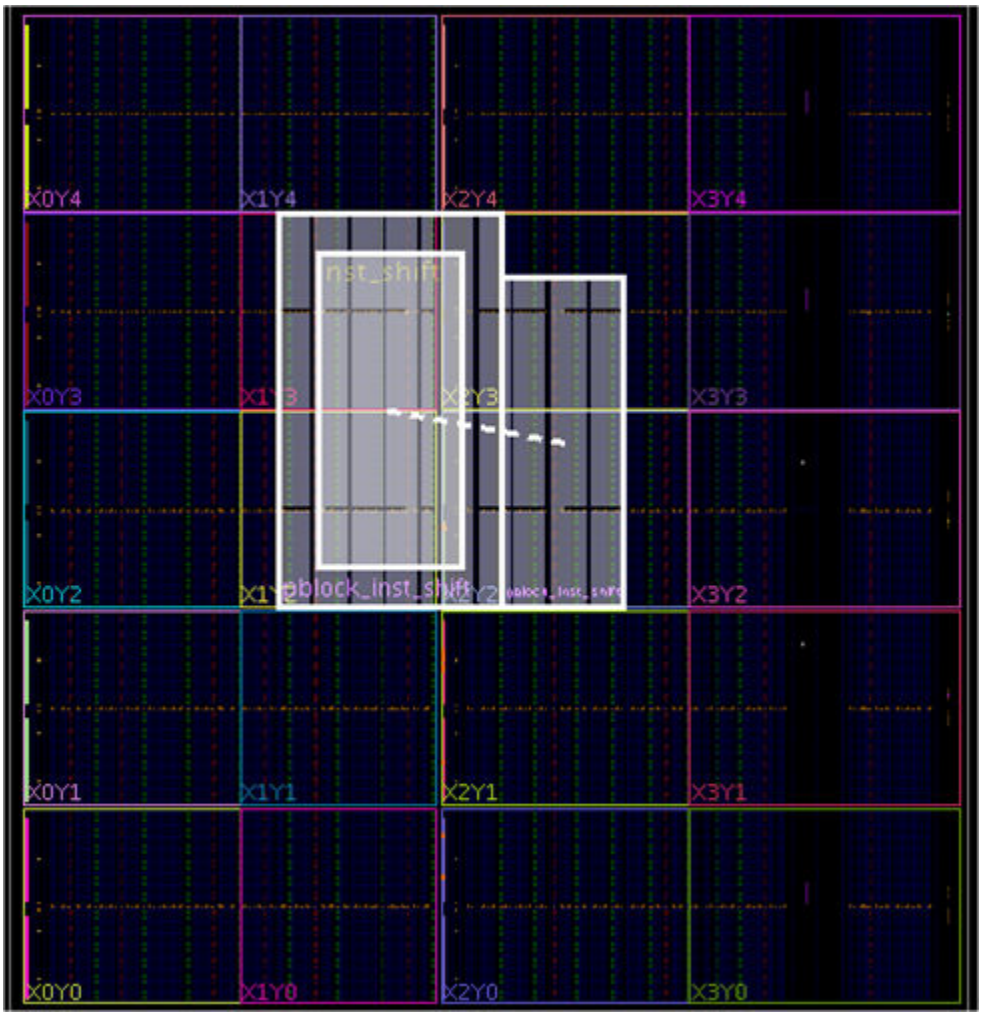
As shown in Floorplanning Rules for Clocks Inside an RP below, small static gaps, such as an IOB column, are permitted in the row of an RP Pblock. However, Xilinx recommends avoiding these gaps when possible, as they are a potential source of routing congestion because RM routes need to route over these gaps.

Figure 70: **Supported Pblock with Small Static Gap**



Small stair-step shaped Pblocks, as shown in Floorplanning Rules for Clocks Inside an RP, are sometimes necessary. While they are supported, they can also lead to routing congestion around the corners.

Figure 71: **Supported Stair-Step Shaped Pblock**



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.

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 this link in *Vivado Design Suite Properties Reference Guide* (UG912).

- The I/O direction and I/O standard must be the same between all RMs whenever the I/O is used.
- 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 may 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 (40, 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 will be at its smallest when the lock cycle of these components are set to "no wait". Similarly, IO with DCI matching requirements will 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

Xilinx high speed transceivers (GTH, GTY) are supported within a 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 during reconfiguration or when the memory is not need; memory contents will be maintained in this case as well. 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 here: https://www.xilinx.com/products/intellectual-property/hbm.html.

Dynamic Function eXchange Checklist for UltraScale and UltraScale+ Device Designs

Xilinx 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)?

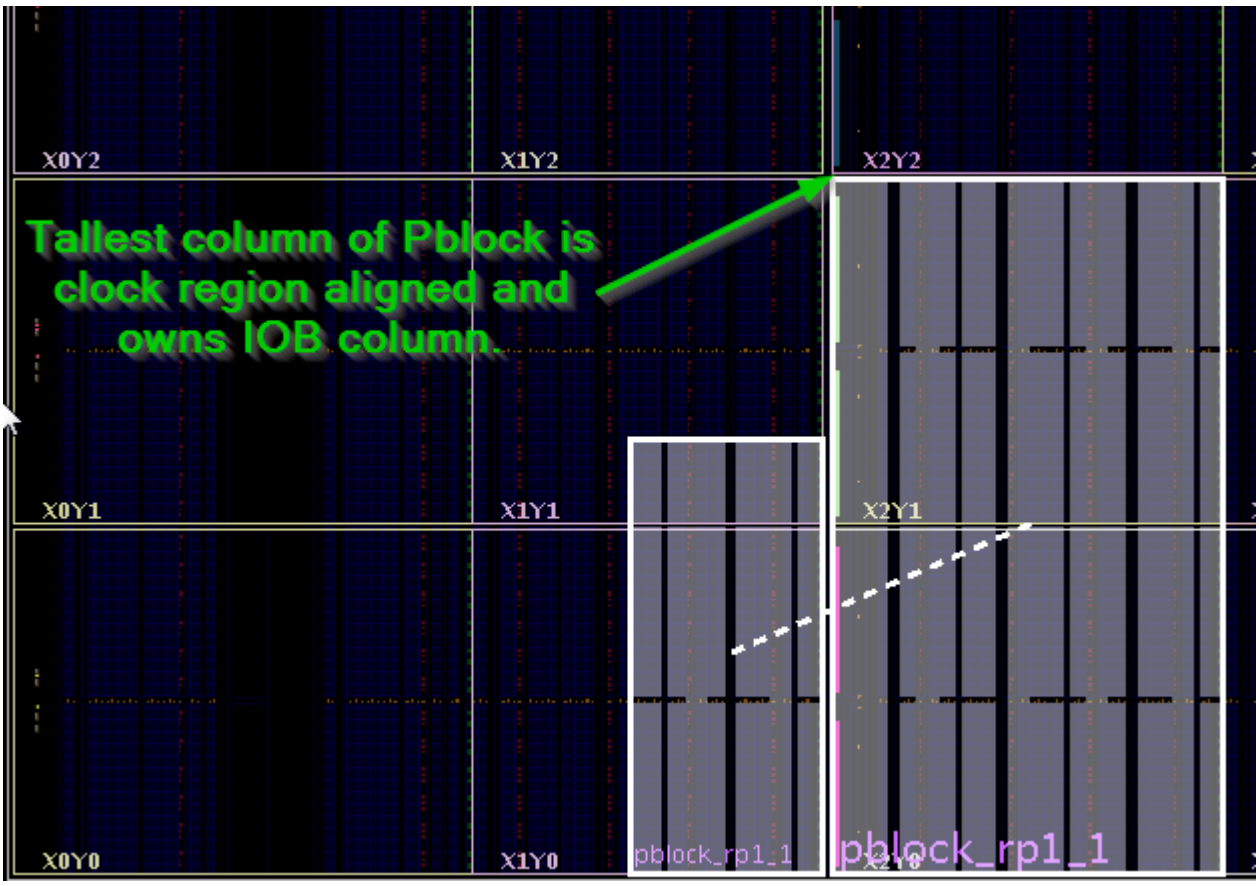
These blocks can be reconfigured, but all elements in this frame type must be reconfigured. This includes an entire I/O bank and all clocking elements in that shared region, plus one column of CLBs that share the interconnect.

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:

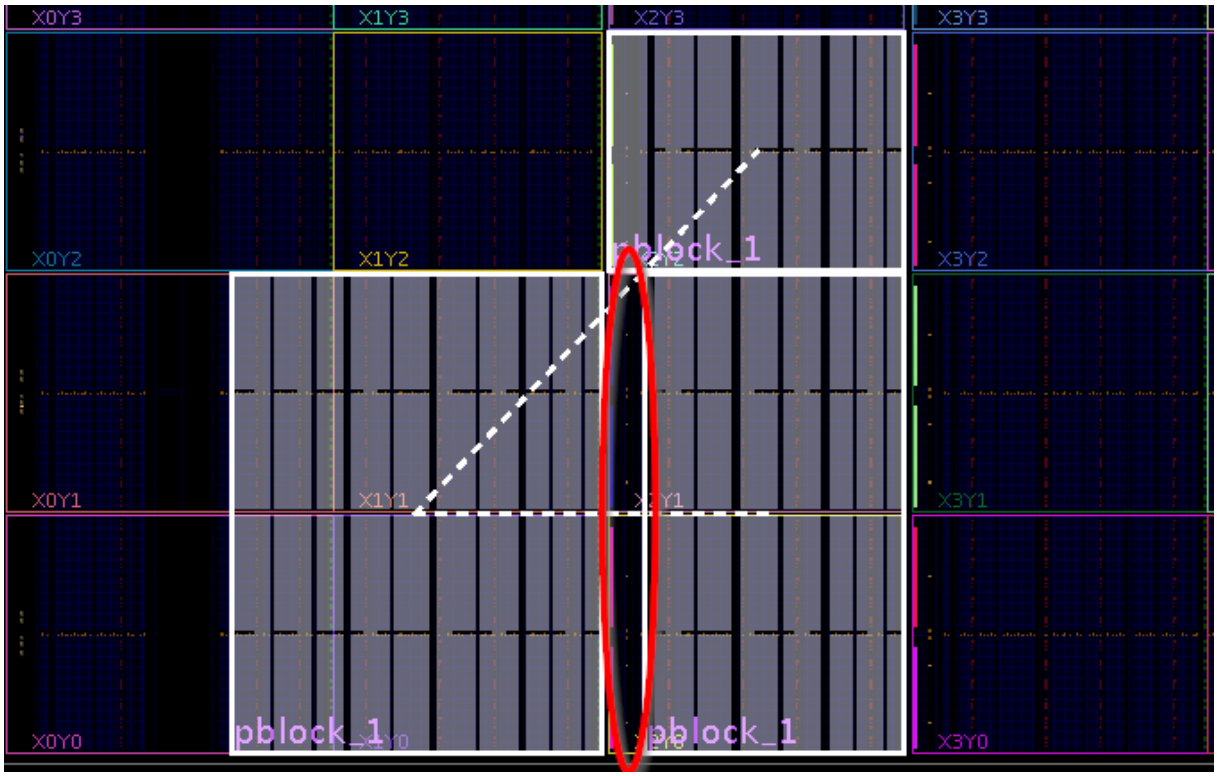
- Xilinx recommends using rectangular Pblock shapes. Non-rectangular shapes are also supported for RPs with clocking logic, as long as the tallest column of the Pblock is aligned vertically and horizontally with the clock region. The tallest column of the RP Pblock must also range the IOB, and this range must cover the full height of all the rectangles that define the RP Pblock, as shown in Dynamic Function eXchange Checklist for UltraScale and UltraScale+ Device Designs. 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 72: **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 Dynamic Function eXchange Checklist for UltraScale and UltraScale+ Device Designs below.

Figure 73: **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 Design Elements Inside Reconfigurable Modules 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 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 Chapter 9: Configuring the Device for more information.

XILINX

Chapter 8

Design Considerations and Guidelines for Versal Devices

This chapter describes the design requirements that are unique to DFX and are specific to Versal devices.

To take advantage of the DFX capability of Xilinx 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

Versal devices support partial reconfiguration for almost all component types. Logic that can be placed in a RM includes:

- NoC Master Units (NMUs) and NoC Slave Units (NSUs)
- Boundary Logic Interface (BLI) flipflops
- XPIO and HDIO banks:
  - Includes XPHY, ISERDES, OSERDES, and IDELAYCTRL
- Memory Controllers:
  - DDRMC and DDRMC_RIU
- Serial transceivers (MGTs) and related components:
  - GTYE5_QUAD, MRMAC, and PCIE40E5
- All logic components that are mapped to a CLB slice:
  - LUTs (look-up tables), LUTRAMs, FFs (flip-flops), SRLs (shift registers), MUXFs, and LOOKAHEAD.

- Block RAM:
  - RAMB18E5 and RAMB36E5
- DSP blocks: DSP48E2
- PCIe® (PCI Express), CMAC (100G MAC), and ILKN (Interlaken MAC) blocks
- UltraRAM blocks: URAM288E5 and URAM288E5_BASE
- Clocks and Clock Modifying Logic:
  - Includes BUFG_FABRIC, BUFGCE, BUFG_GT, BUFG_GT_SYNC, BUFGMUX, MMCM, DPLL, XPLL and MBUFG
- AI Engines
  - Versal AI Engine inclusion in RM is supported through Vitis platform flows only.

Creating Pblocks for Versal Devices

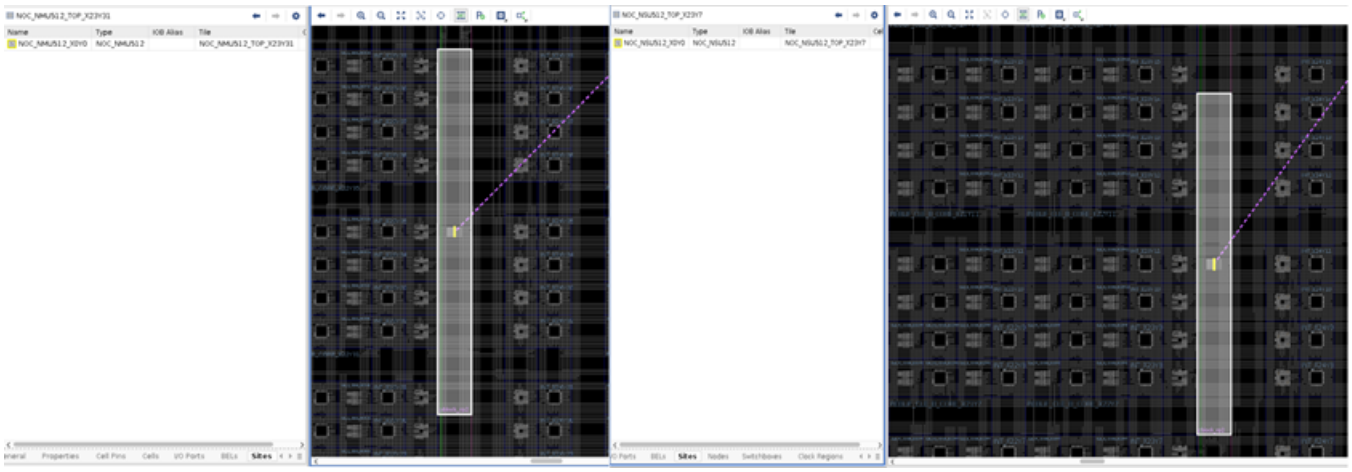
As part of improvements to the Versal architecture, the smallest unit that can be reconfigured is much smaller than in previous architectures. The minimum required resources for reconfiguration varies based on the resource type, and are referred to as a Programmable Unit (PU). Many site types have improved PU requirement making granularity of reconfigurable Pblocks significantly improved compared to previous architecture.

TIP: *While the fundamental building blocks are shown in the following images, in real design scenarios they will be part of a larger collection of resources, creating a comprehensive floorplan for each dynamic region.*

The following are the details provided for each site type:

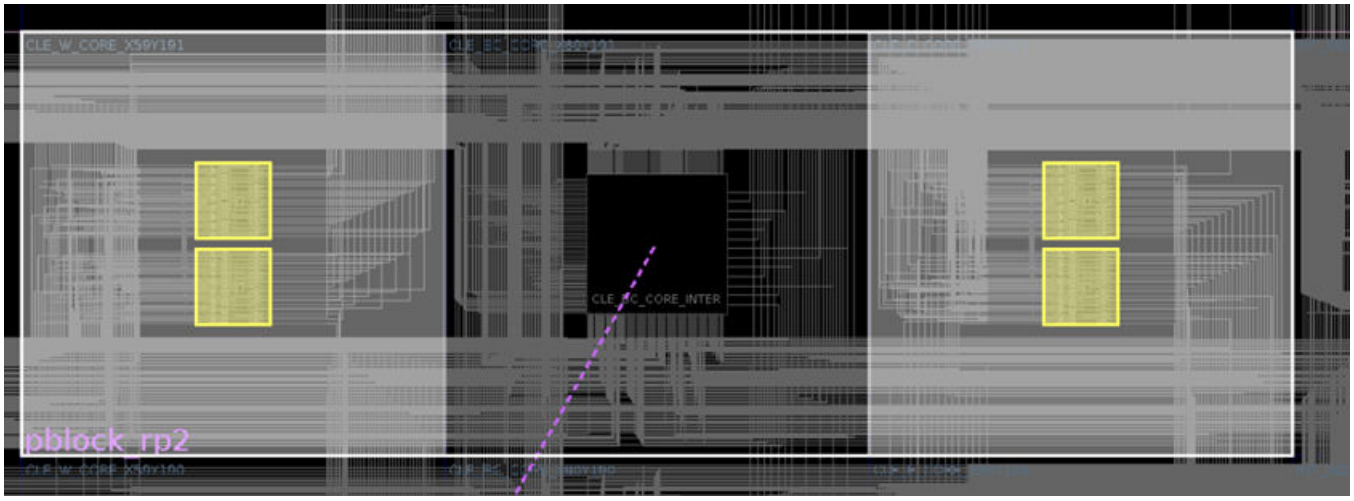
Programmable Logic (PL) NoC NMU and NSU: PU is the corresponding NOC_NMU or NOC_NSU tile.

Figure 74: **PL NoC NMU and NSU**



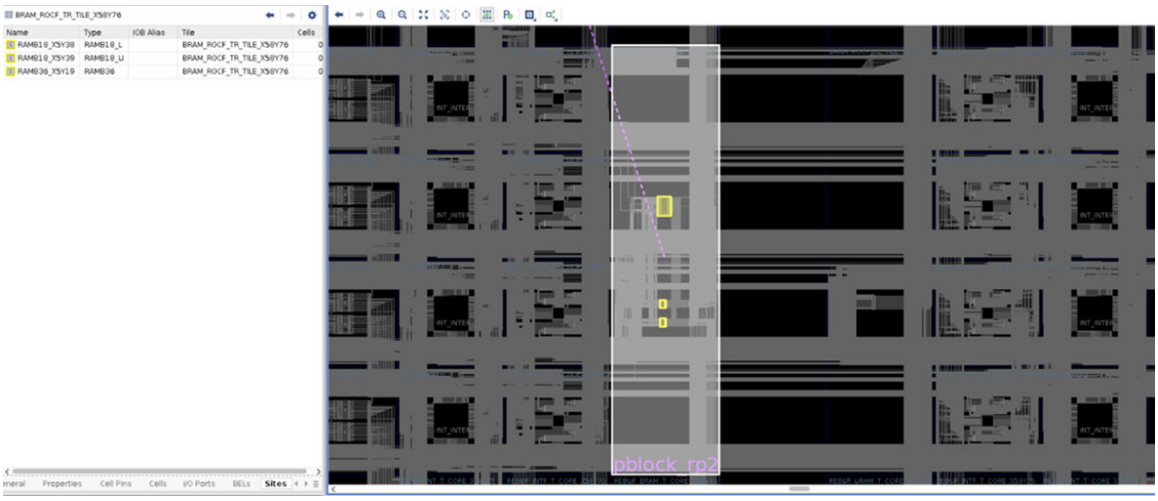
CLE: Two adjacent CLE tiles share a routing resource (interconnect tile). PU is 2 CLE tiles (4 SLICE sites) with shared interconnect.

Figure 75: **CLE PU**



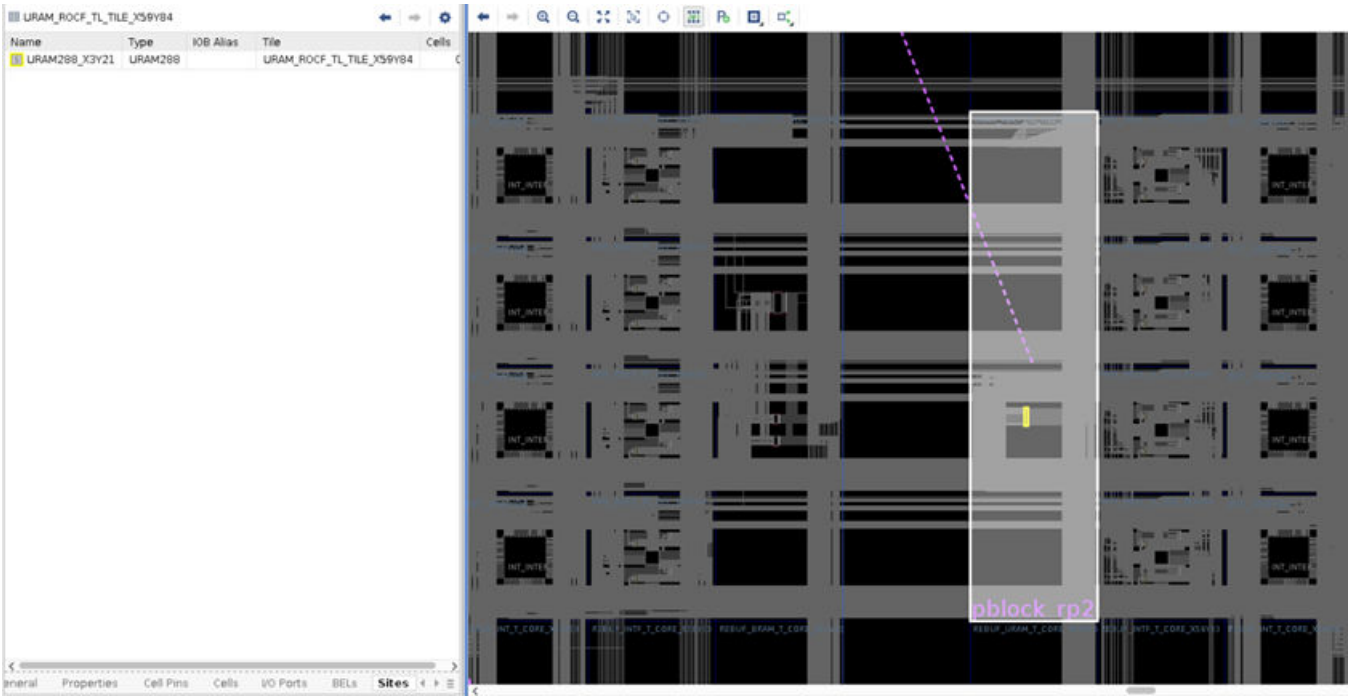
BRAM: PU is the corresponding BRAM tile. One BRAM tile includes two RAMB18s and one RAMB36. Adjacent INTF and INT tiles are automatically pulled into the routing footprint if it is not covered by the Pblock. Unlike previous architecture, adjacent CLE sites are not part of BRAM PU.

Figure 76: **BRAM PU: RAMB18s and RAMB36 of 1 BRAM Tile**



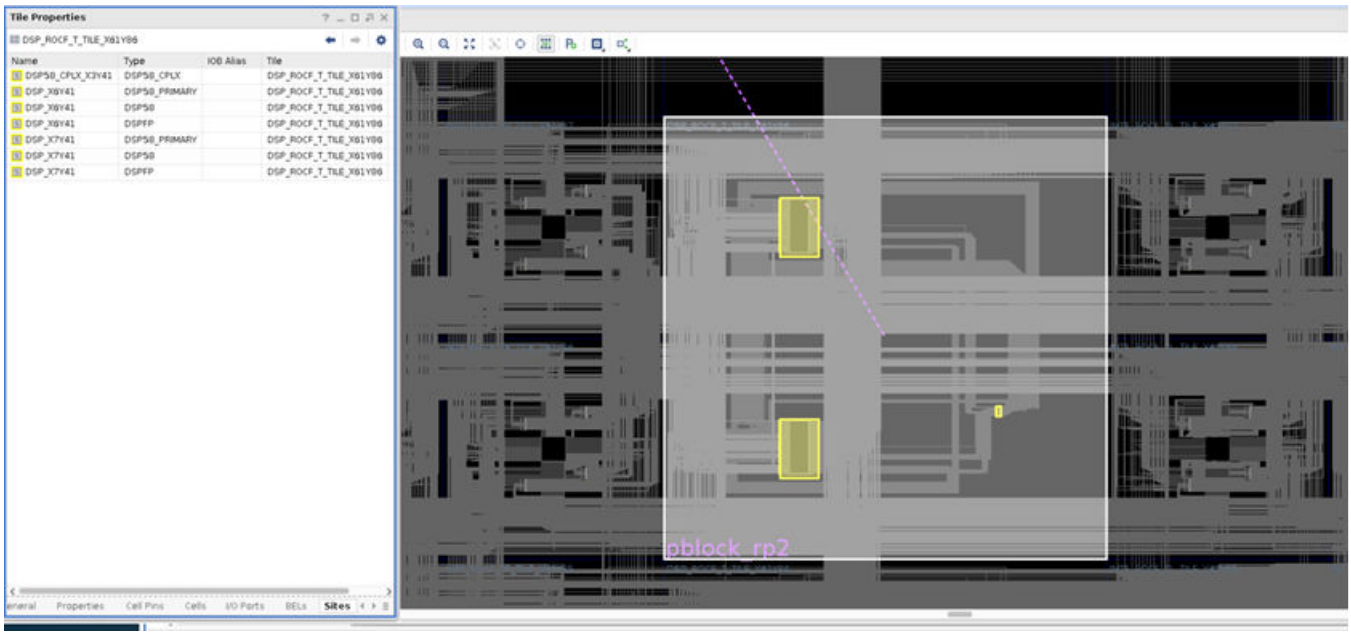
URAM: PU is the corresponding URAM tile. One URAM tile includes only 1 URAM site. Adjacent INTF and INT tiles are automatically pulled into the routing footprint if it is not covered by the Pblock.

Figure 77: **URAM PU:URAM Tile**



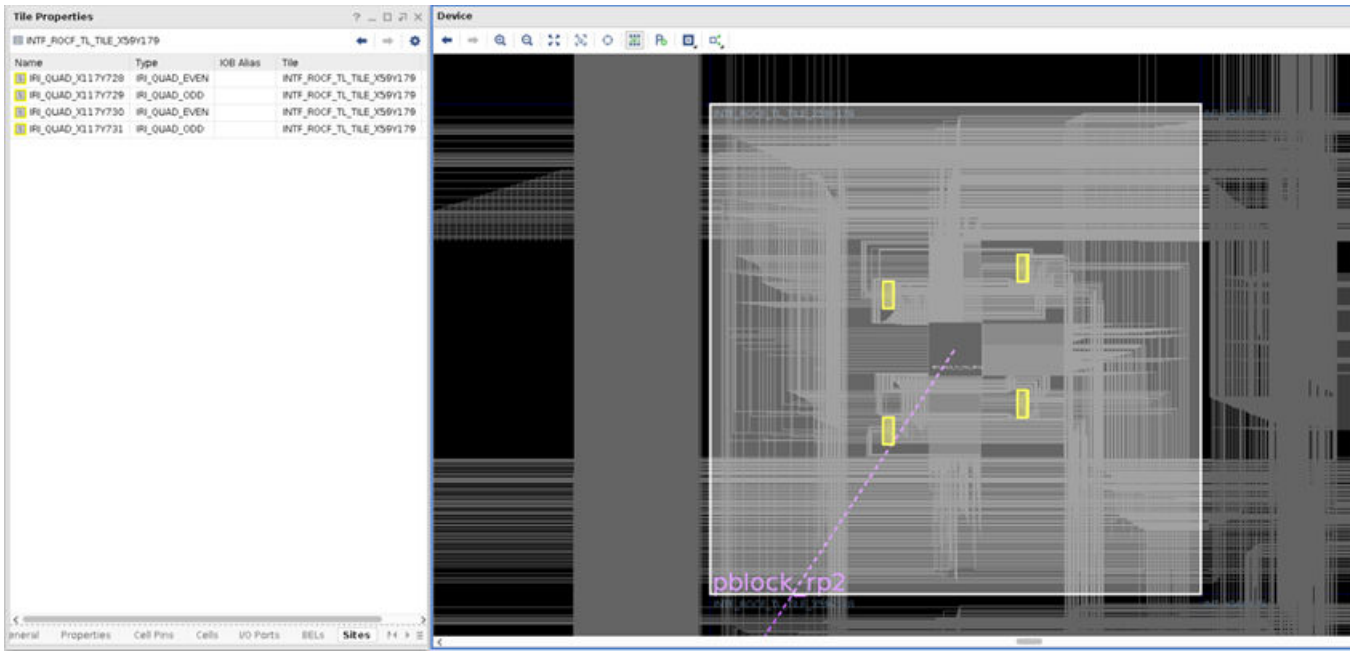
DSP: PU is the corresponding DSP tile. One DSP tile includes 2 DSP sites.

Figure 78: **DSP PU: DSP Tile**



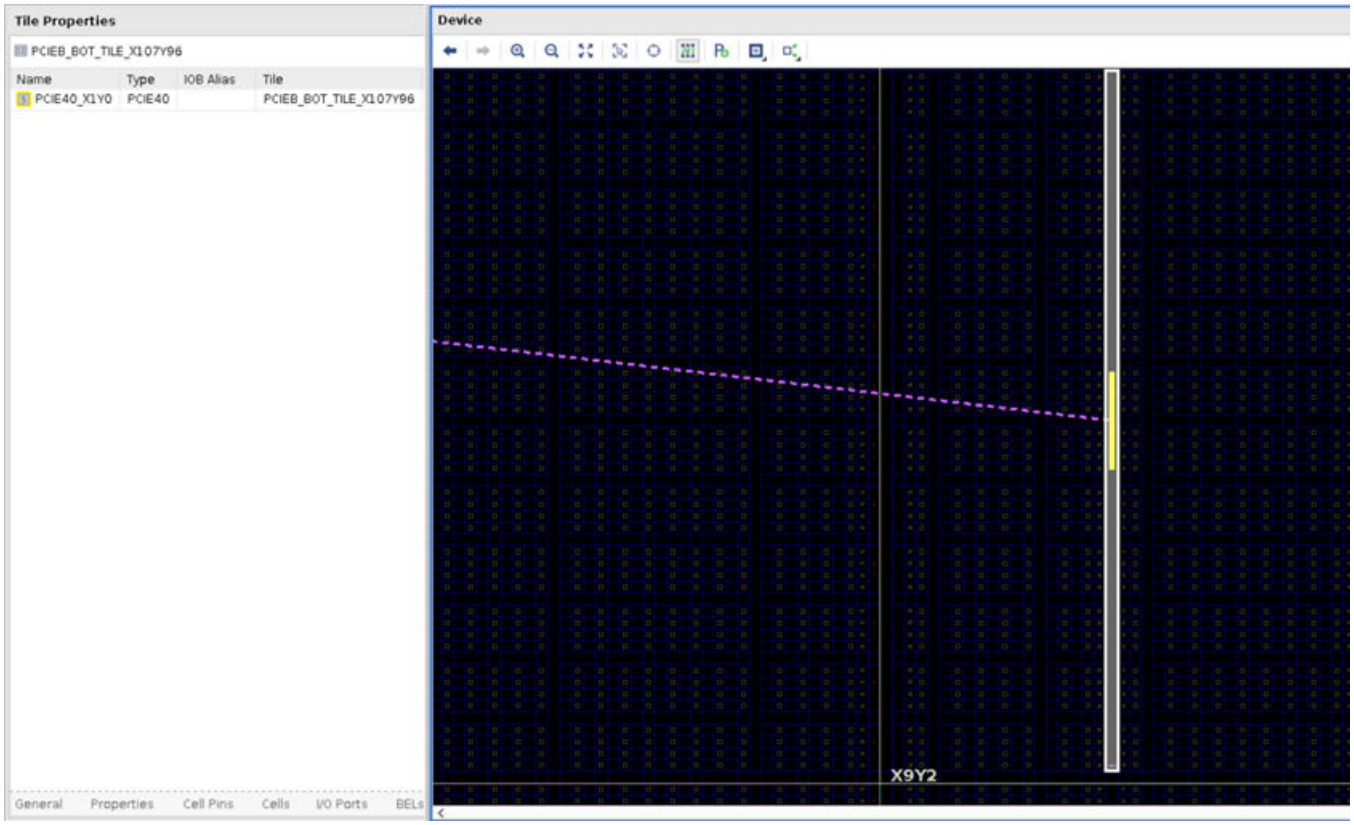
IRI_QUAD (ODD/EVEN): PU is the corresponding INTF_ROCF_TL_TILE. One tile includes 4 IRI Quads. INTF at the center of IRI quads is automatically pulled into routing footprint. Though IRI_QUADs are user range-able, adjacent IRI_QUADs of RP Pblock are automatically pulled in to the routing footprint since expanded routing footprint is always 2 INT tile expansion.

Figure 79: **Imux Register Interface Quad: PU is INT_ROCF_TL Tile**



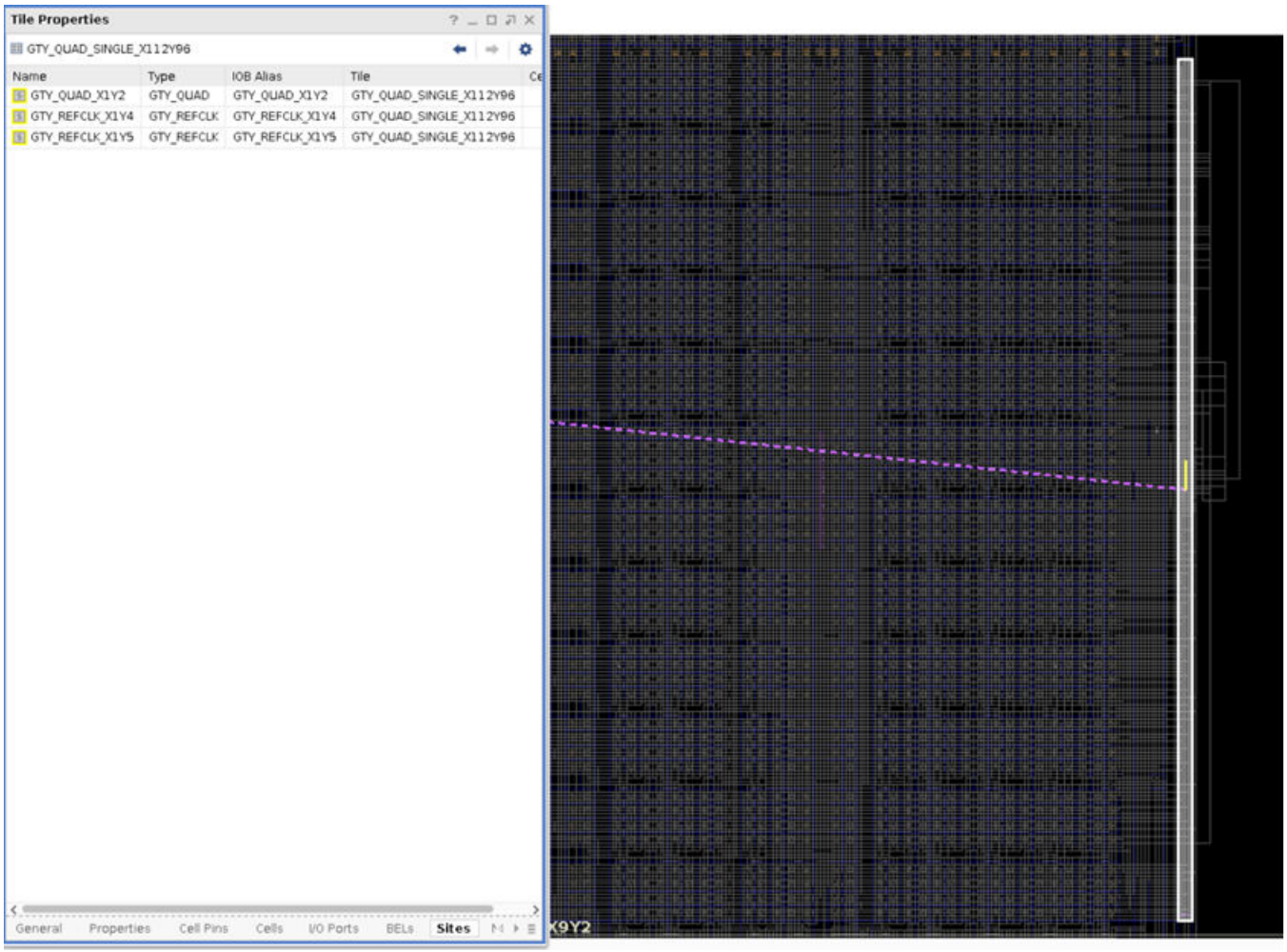
PCIe 40: PU is the corresponding `PCIEB_BOT_TILE`. Adjacent INTF tiles are automatically included in the routing footprint of reconfigurable Pblock.

Figure 80: **PCIe 40 PU is PCIe Tile**



GTY_QUAD: PU is the corresponding GTY_QUAD_SINGLE tile. Sites included in the tile are GTY_QUAD and GTY_REFCLK. Adjacent INTF_GT tiles are automatically pulled into the routing footprint of reconfigurable Pblock.

Figure 81: **GTY_QUAD PU is GTY_QUAD Tile**



DDRMC and DDRMC_RUI : PUs are corresponding DDRMC_DMC_CORE and DDRMC_RIU_CORE tiles respectively.

Figure 82: **DDMRC PU**

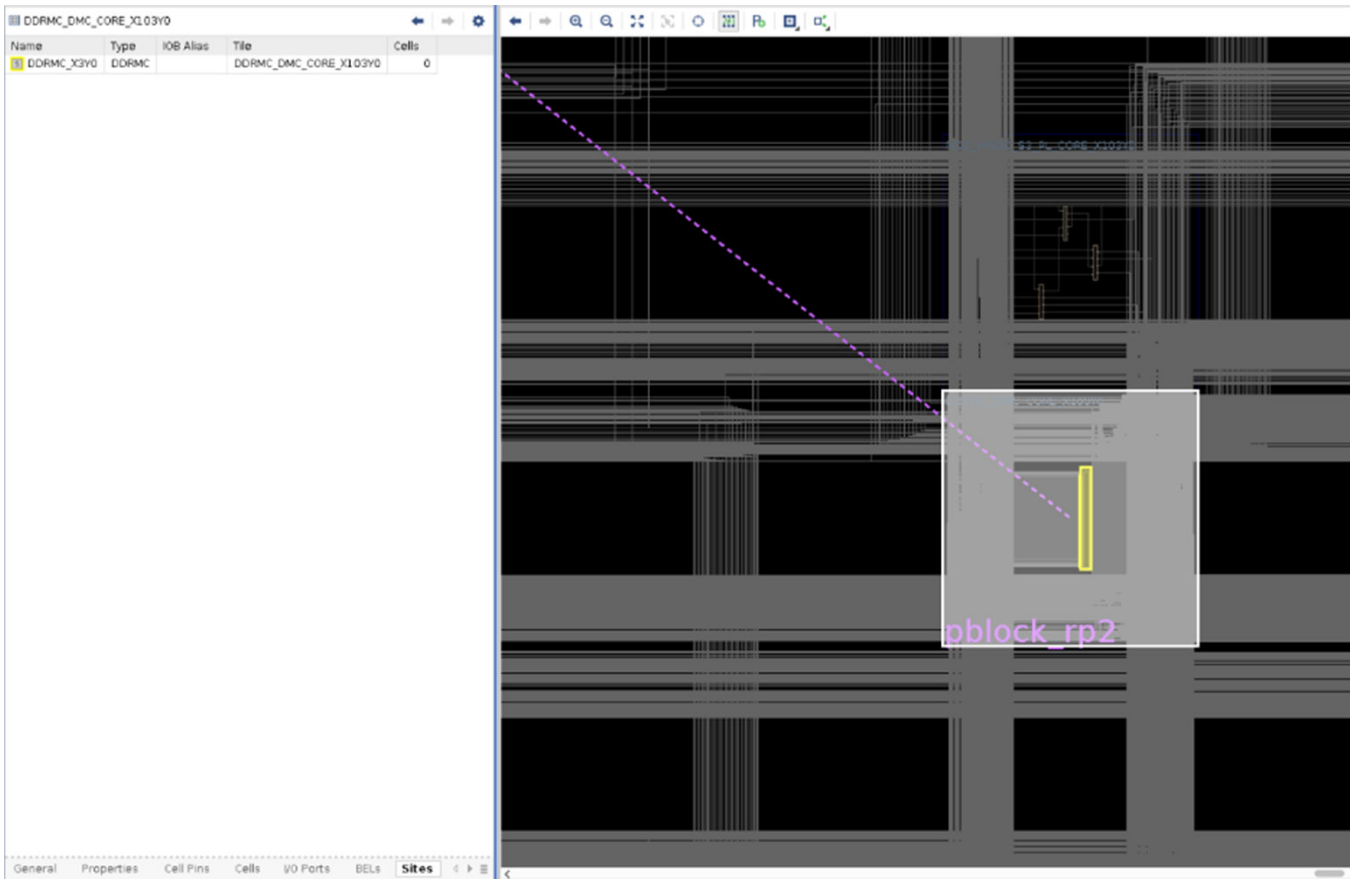
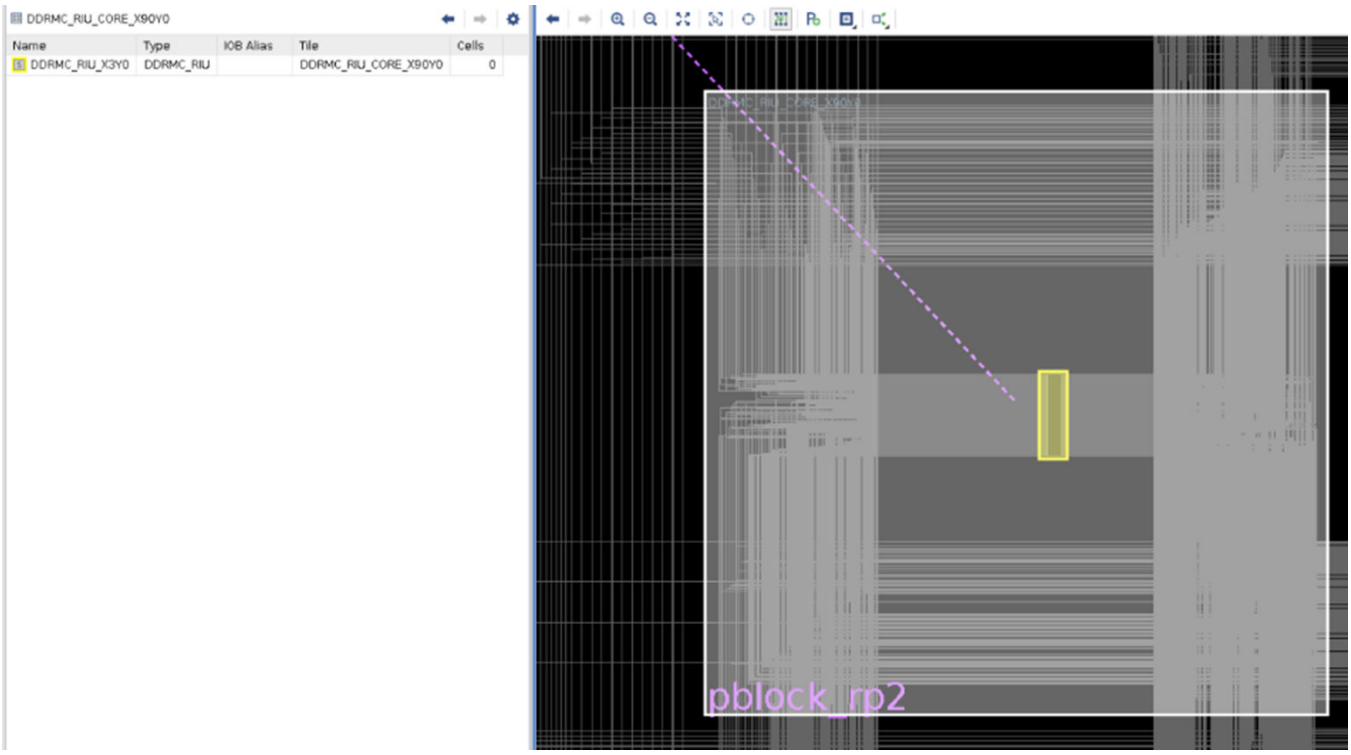
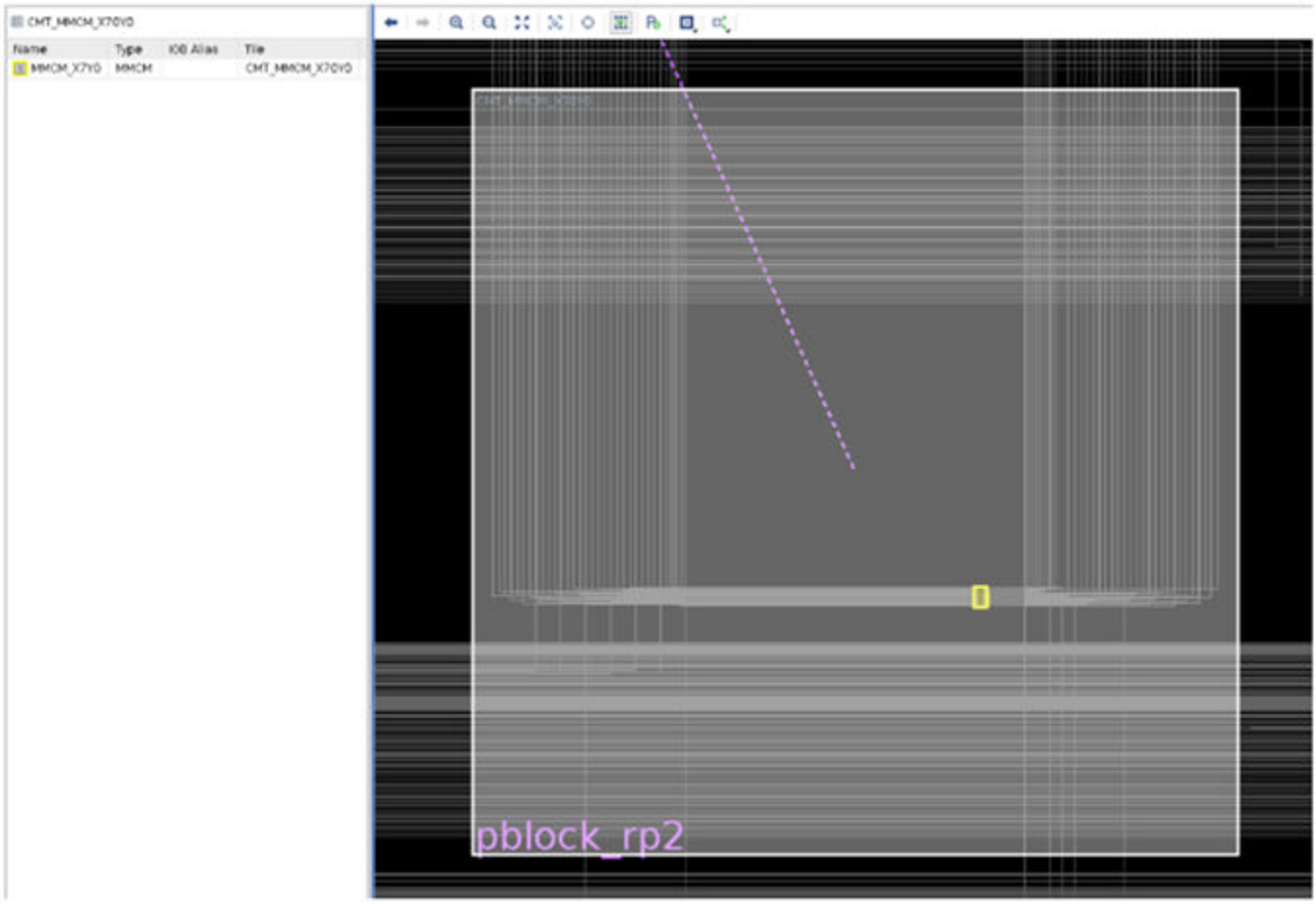


Figure 83: **DDMRC_RUI PU**



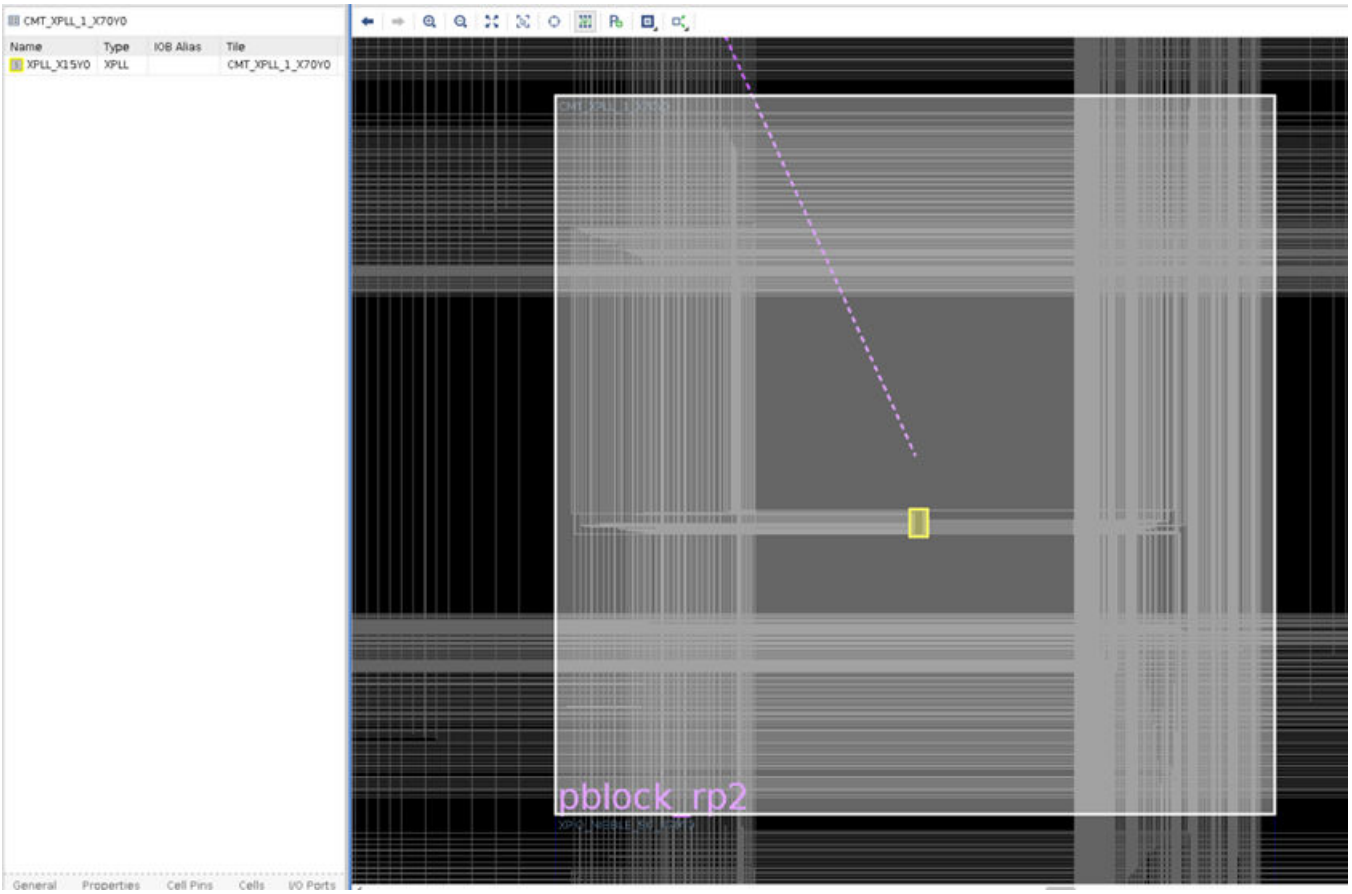
MMCM: PU is corresponding CMT_MMCM tile.

Figure 84: **MMCM PU is CMT_MMCM Tile**



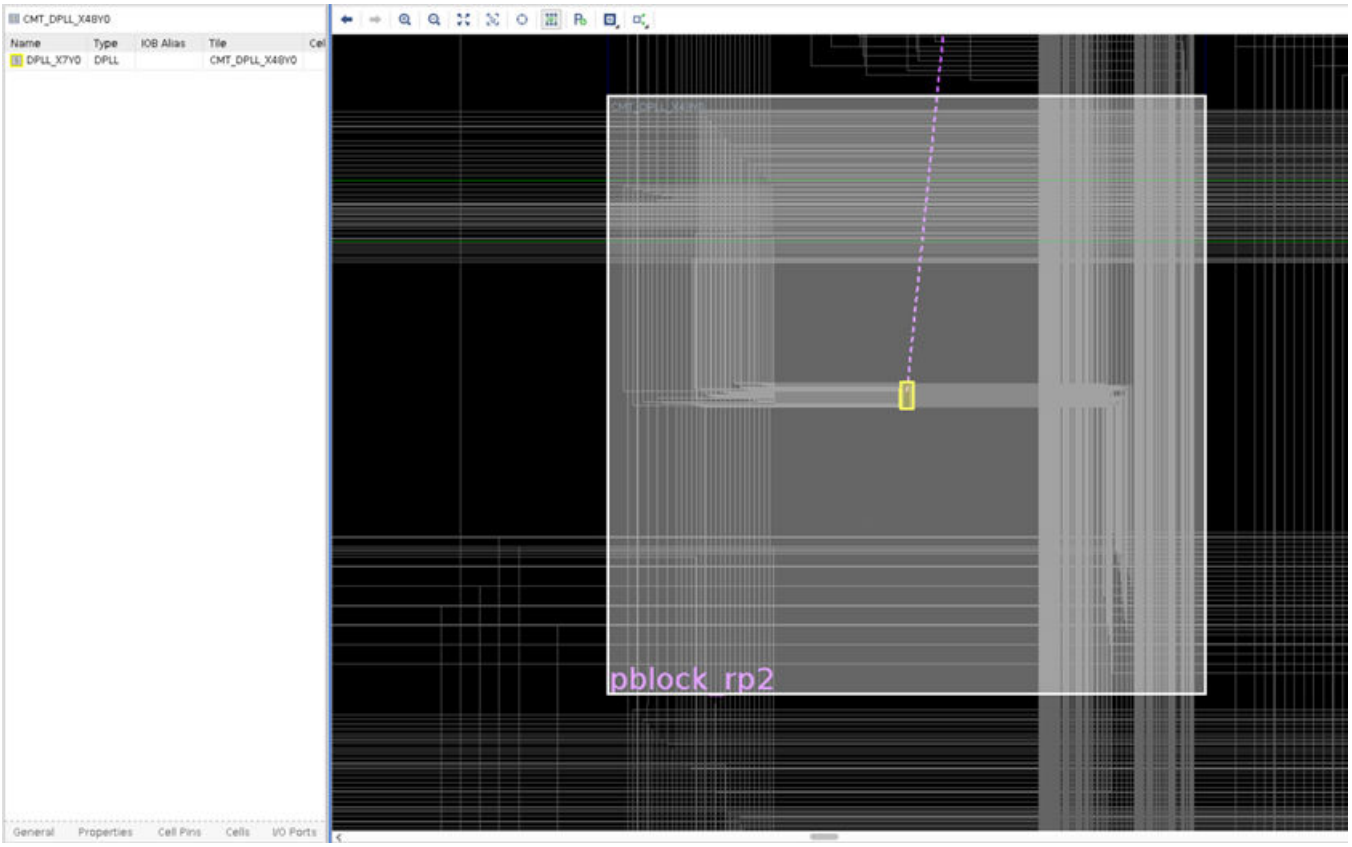
XPLL: PU is corresponding CMT_XPLL tile.

Figure 85: **XPLL PU is CMT_XPLL Tile**



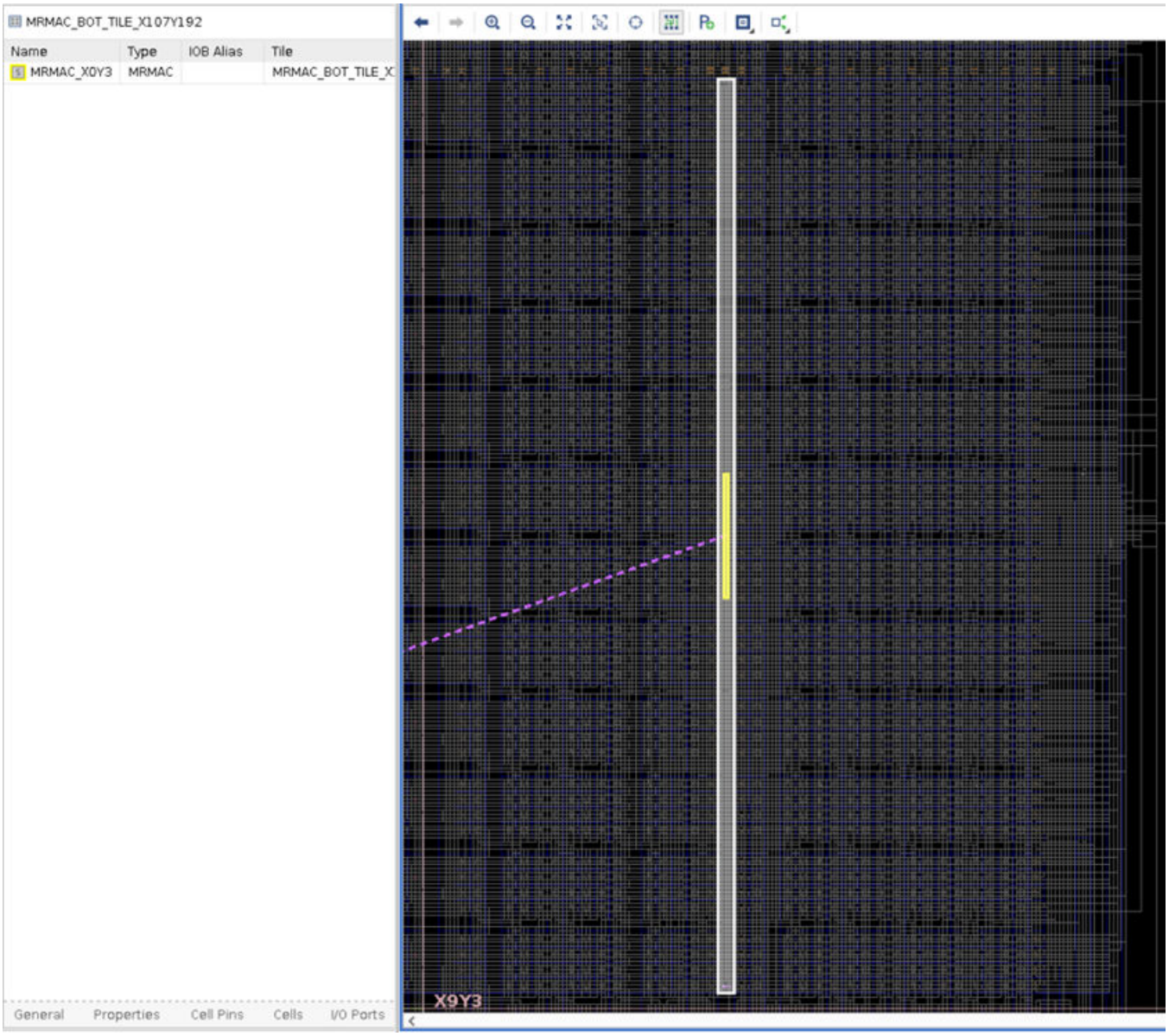
DPLL: PU is corresponding CMT_DPLL tile.

Figure 86: **DPLL PU is CMT_DPLL Tile**



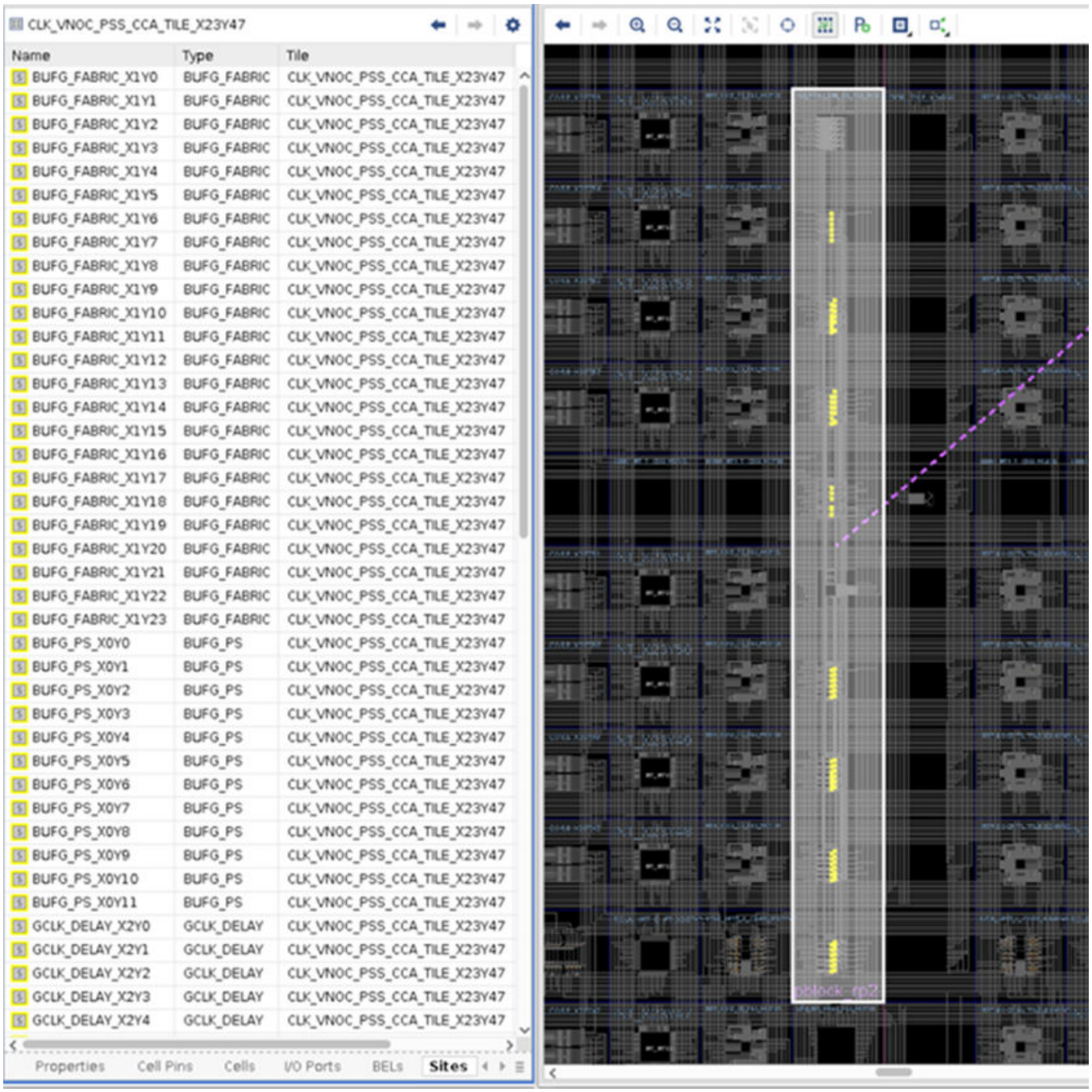
MRMAC: PU is corresponding MRMAC_BOT tile.

Figure 87: **MRMAC PU is MRMAC_BOT Tile**



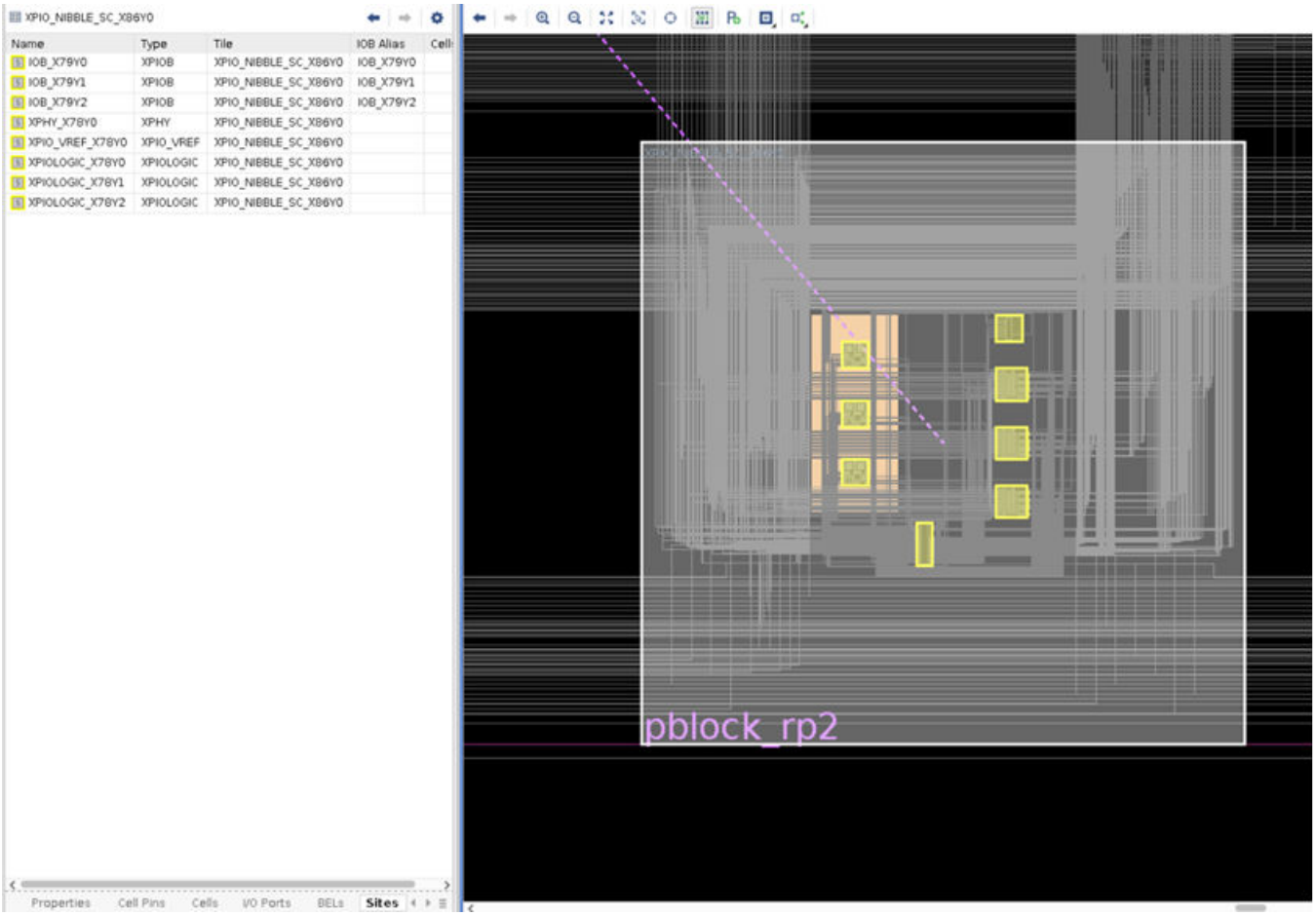
BUFG_FABRIC, BUFG_PS and GCLK_DELAY: These three site types are included in same tile: CLK_VNOC tile. BUFG_PS will be present only in the VNOC column adjacent to CIPS. Other VNOC tiles include only BUFG_FABRIC and GCLK_DELAY. PU requirement is the CLK_VNOC tile.

Figure 88: **BUFG_FABRIC, BUFG_PS, and GCLK_DELAY Shares Same Tile As PU**



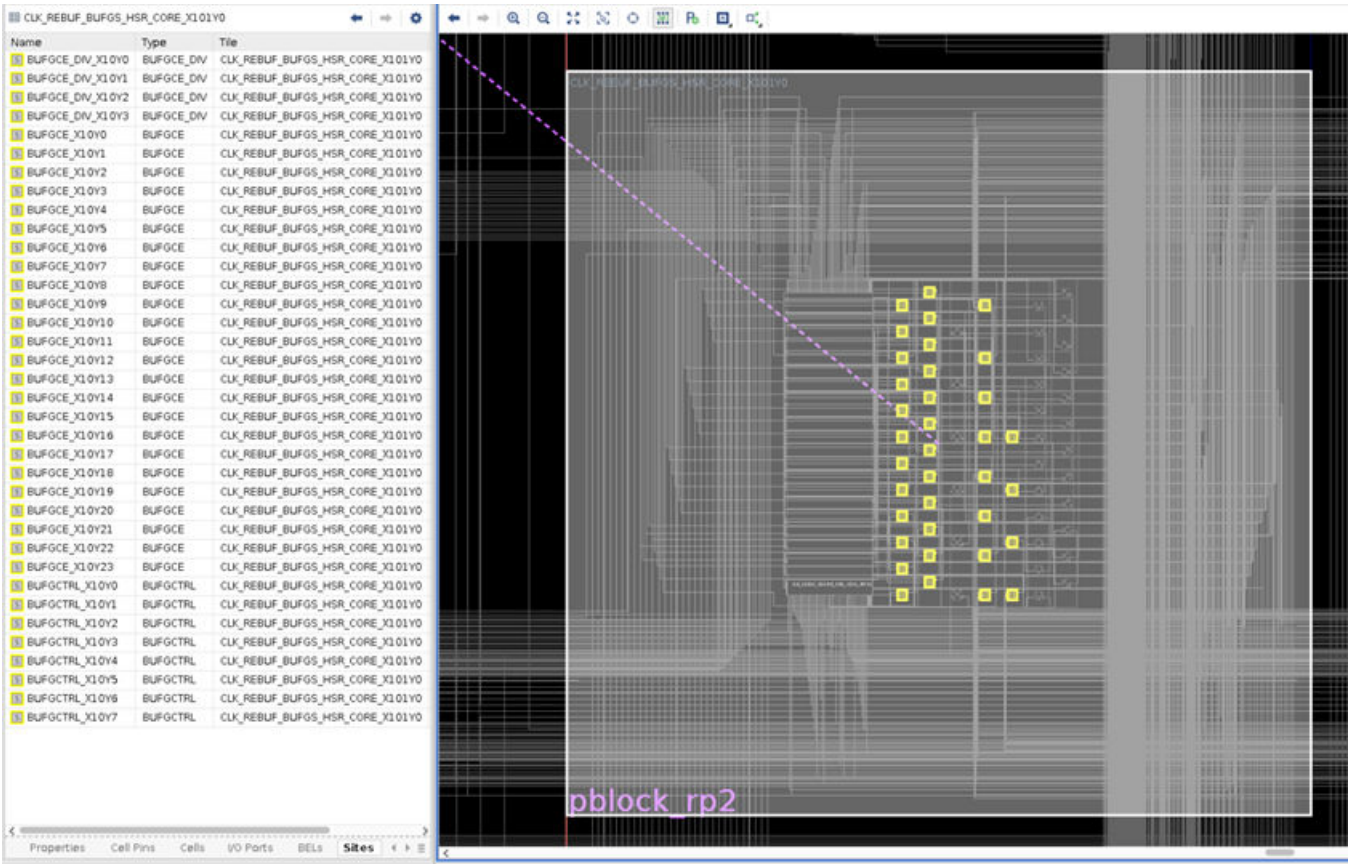
XPHY, XPIO and IOB: XPIO_NIBBLE tile is the PU

Figure 89: **XPIO, XPHY, and IOB share same PU: XPIO_NIBBLE Tile**



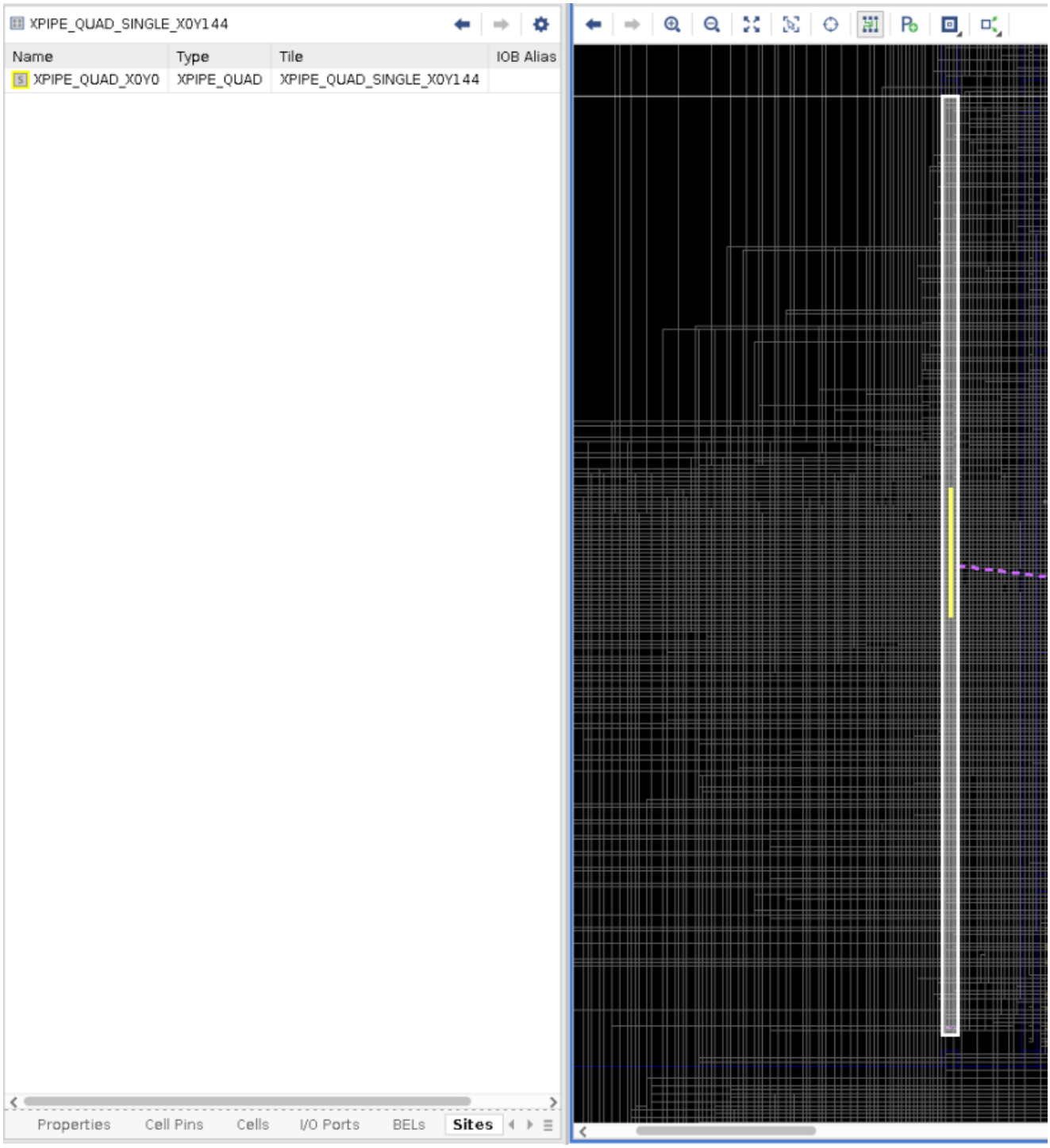
BUFG_GT, BUFG_GT_SYNC and GCLK_DELAY: CLK_GT tile is the PU.

Figure 90: **BUFG_GT, BUFG_GT_SYNC Share Same PU: CLK_GT Tile**



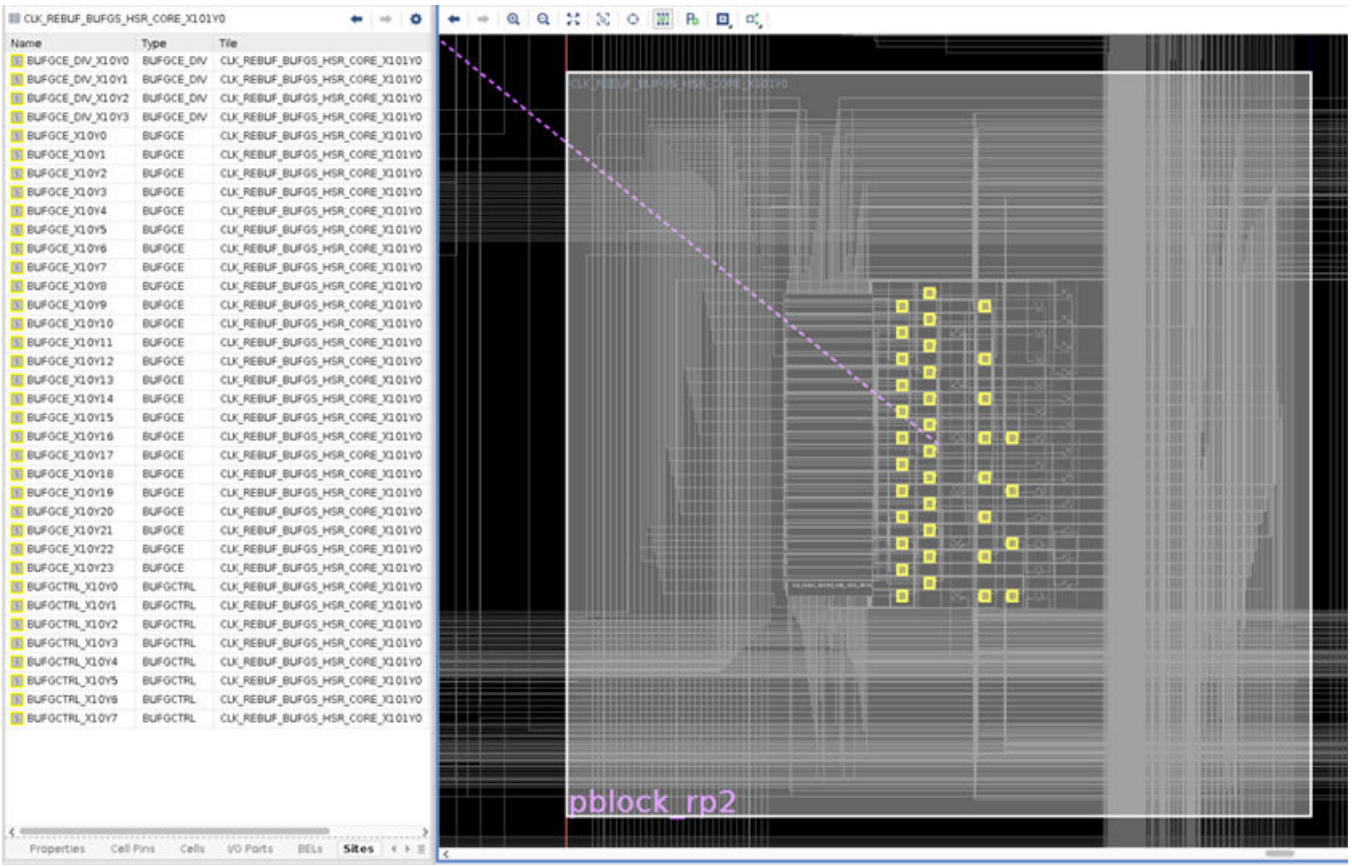
XPIPE_QUAD: XPIPE_QUAD tile is the PU.

Figure 91: **XPIPE_QUAD PU**



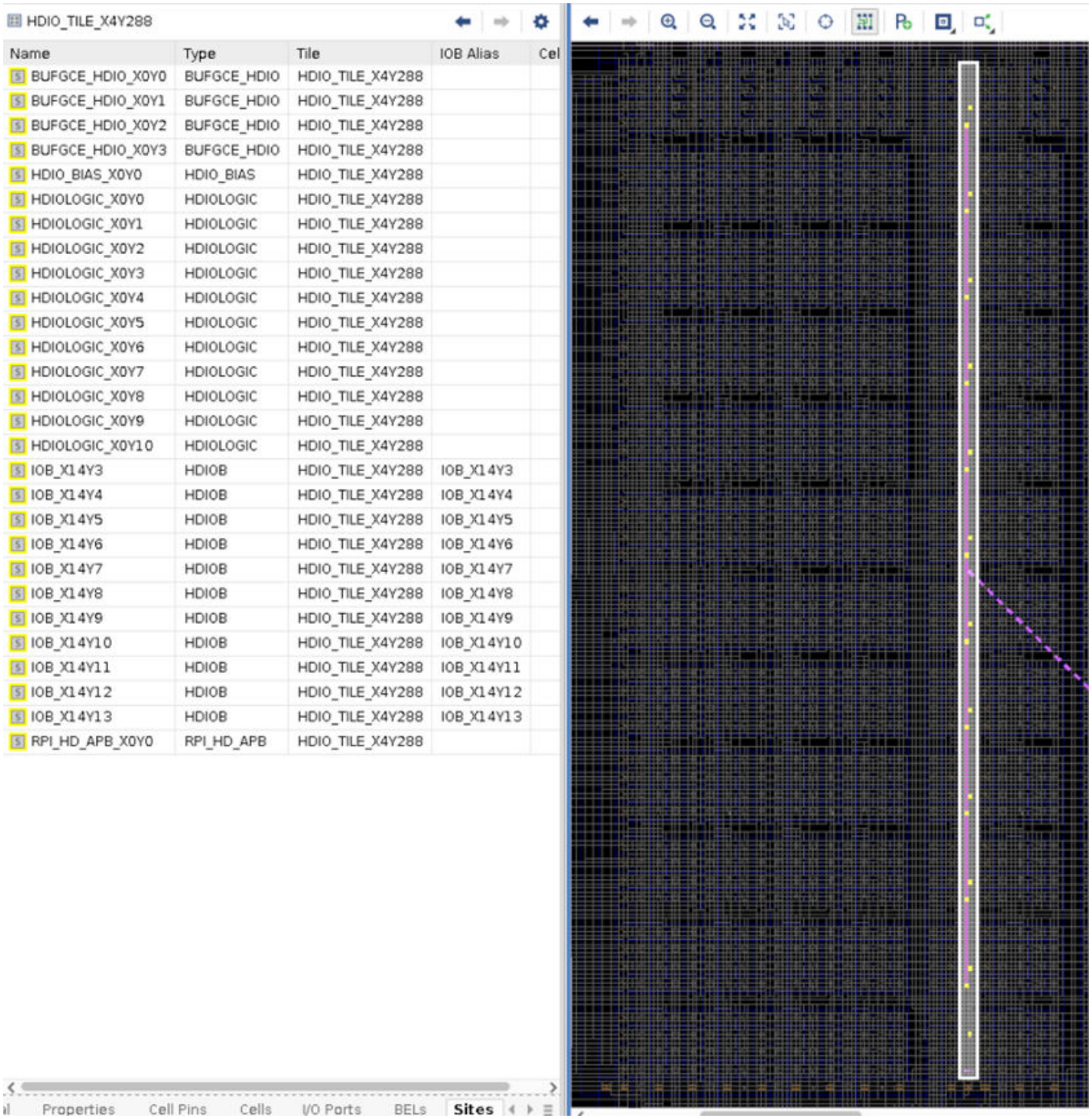
BUFGCE, BUFGCTRL, and BUFGCE_DIV: For the BUFGCE elements in HSR, PU is the CLK_REBUF_BUFGS_HSR_CORE tile.

Figure 92: **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. PU is HDIO_TILE tile.

Figure 93: **Clock Buffers and IOB in HDIO Bank Share Same PU: HDIO_TILE**



Sharing Clock Region 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 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 94: **Sharing Clock Region between Multiple Reconfigurable Pblocks**

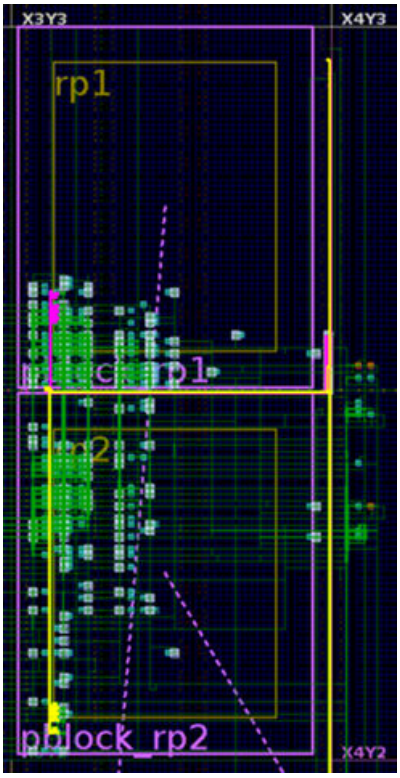


Figure 95: **Schematic View of Two Reconfigurable Modules With An Internal Clock**

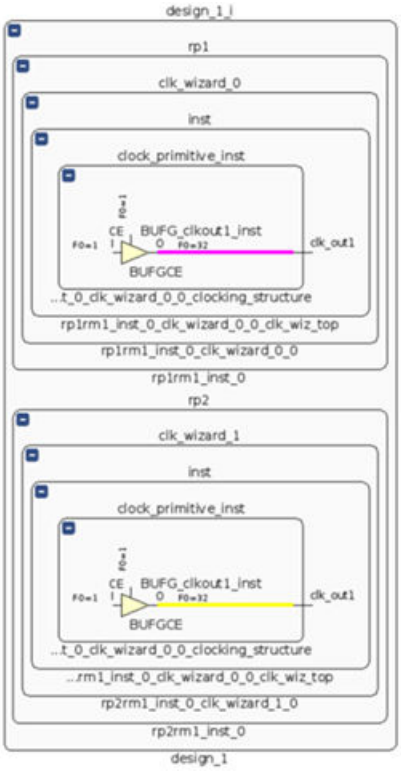
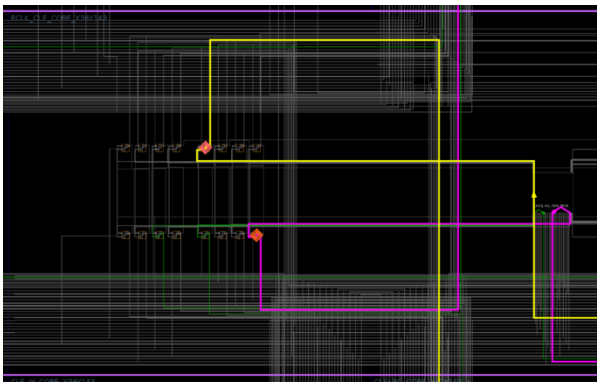


Figure 96: **Magnified View of Device. Top Half of RCLK is Used by One Partition's Clock Net And Bottom Half is Used By Other Partition's Clock Net**



HD_Visual Scripts

Similar to previous architectures, the DFX flow in Versal generates scripts in a folder called `hd_visual` inside the implementation directory. In addition to placement and routing footprint, there are few more Tcl scripts generated by Versal DFX flow which can be used for debugging purpose if needed.

- `Overlapped_AllTiles.tcl` In Versal, Interconnect and Clocking tiles can potentially be shared by two reconfigurable partitions. This contains the list of tiles that are shared by multiple RPs.
- `<pblock_name>_top_SplitPRTiles.tcl`: If `RCLK` row is split into two RPs, Pblock that has the top half of RCLK tow corresponds to this TCL. This contains the list of clock tiles that will be programmed during partial PDI download of that RP.
- `<pblock_name>_bottom_SplitPRTiles.tcl`: If RCLK row is split into two RPs, Pblock that has the bottom half of RCLK tow corresponds to this TCL. This contains the list of clock tiles that will be programmed during partial PDI download of that RP.
- `<pblock_name>_Placement_AllTiles.tcl`: All tiles that are in the placement range of the corresponding RP pblock.
- `<pblock_name>_Routing_AllTiles.tcl`: Routing footprint of a reconfigurable pblock will be different from placement footprint. This is because of features like expanded routing and clock routing expansion. This TCL file provides the list of all tiles in the routing footprint of that RP which will be programmed during partial PDI download. Routing footprint will be a superset of placement footprint.

  Note: Interconnect tiles at the edge of 2 adjacent RPs can be shared by both RPs. However, this situation should be avoided whenever possible as each RP gets only half the routing resources of that interconnect tile causing potential routability issues.

  It is always advised to avoid adjacent RPs. It is recommended to keep a static shim Pblock between two RPs and assign the static endpoints between 2 RPs into that Pblock.

BLI Floorplan Alignment

In Versal, 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 helps optimize timing of interface.

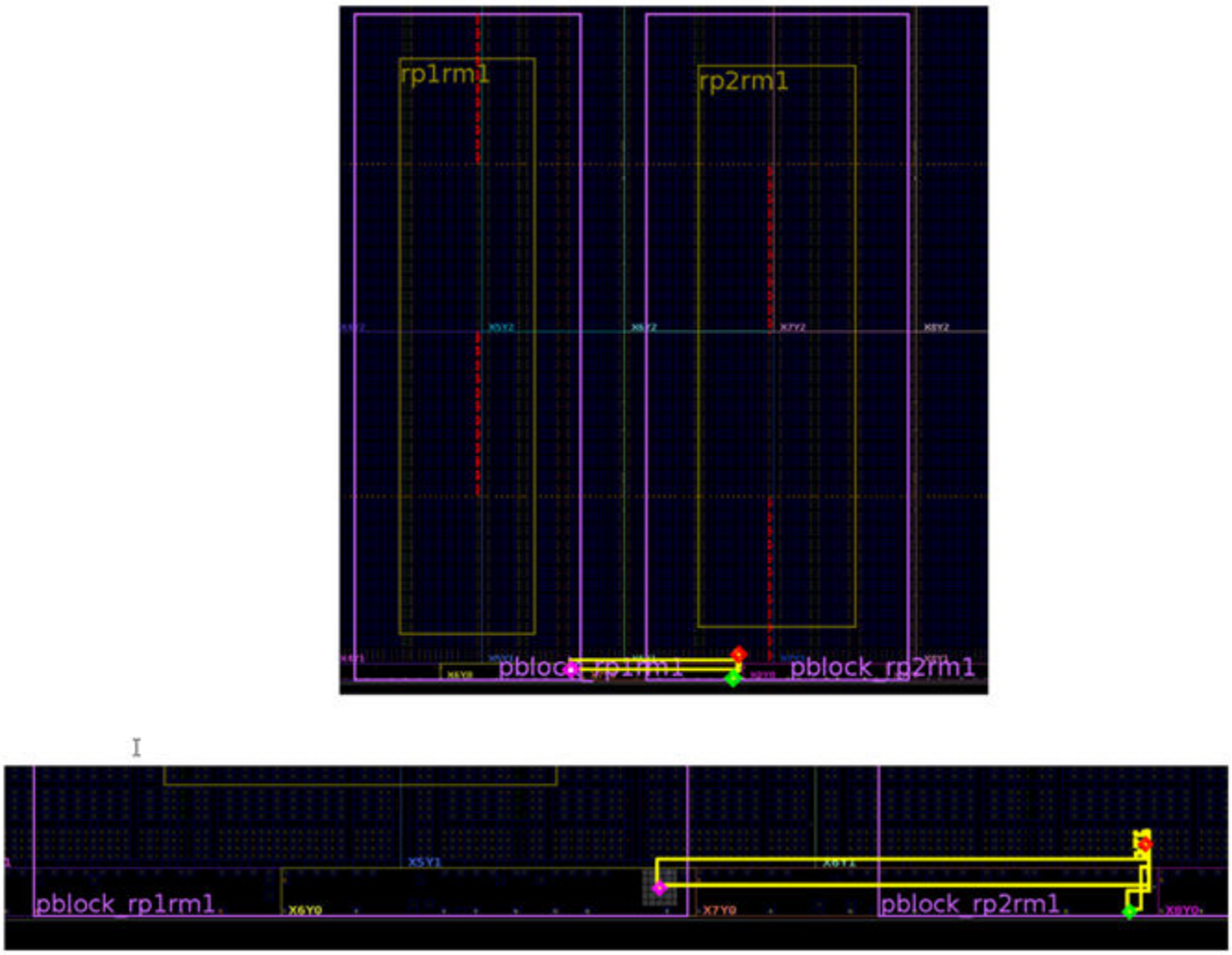
Based on the location, a BLI tile can be used by multiple sites.

- XPHY and XPIOLOGIC sites in `XPIO_NIBBLE` tile.
- DDRMC
- XPLL sites in `CMT_XPLL` tile.
- DPLL sites in `CMT_DPLL` tile.
- BUFGCE sites in `CLK_REBUF_BUFGS_HSR_CORE` tile.
- MMCM sites in `CMT_MMCM` tile.

Since the BLIs are aligned geometrically exactly to each of these site types that uses it, DFX flow automatically pulls the BLIs based on the tiles in the range of Pblock. Here 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, connected BLI will be pulled into the placement footprint.
- If clocking resources like BUFG, MMCM are ranged in a reconfigurable Pblock, connected BLI will be pulled into the placement footprint by the tool.
- If AIE_PL or AIE_NOC sites are ranged in a reconfigurable Pblock, connected BLI will be 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. DRC message also provides the resolution to remove the corresponding tile from the Pblock to avoid the conflict.

Figure 97: **BLI Sharing and Floorplan Alignment**



In figure above, the BLI (red mark) in rp2rm1 is having direct connections to XPIO (green color marked) in rp2rm1 and DDRMC (magenta color marked) in `rp1rm1`. Yellow highlight shows the nodes of BLI which has direct connections to these tiles. This will trigger the DRC mentioned below since BLI is being shared by tiles of two different RPs. User can remove one of the shared tiles to get past the DRC and avoid driving same BLI by two independent RPs.

Resolution for this is to remove the sites of the tile using the following command:

```
resize_pblock -remove pblock_rp2rm1 [get_sites -of [get_tiles
XPIO_NIBBLE_SC_X78Y0]]
```

Expanded Routing

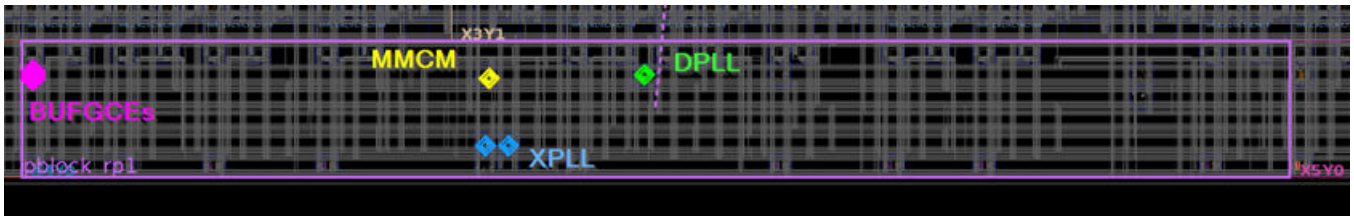
Similar to UltraScale+ architecture, the expansion of routing area also happens in Versal for logical signals. The routing footprint of a reconfigurable Pblock can be understood by sourcing the script called `<pblock_name>_routing_tiles.tcl` in `hd_visual` folder of implementation directory. For clock routing, the necessary clock routing tiles (for example, CLK_VNOC) of RP Pblock are automatically pulled into the routing footprint.

WARNING! *The Expanded Routing feature should not be disabled for Versal devices to ensure the highest possibility of routing success.*

Clocking Resources

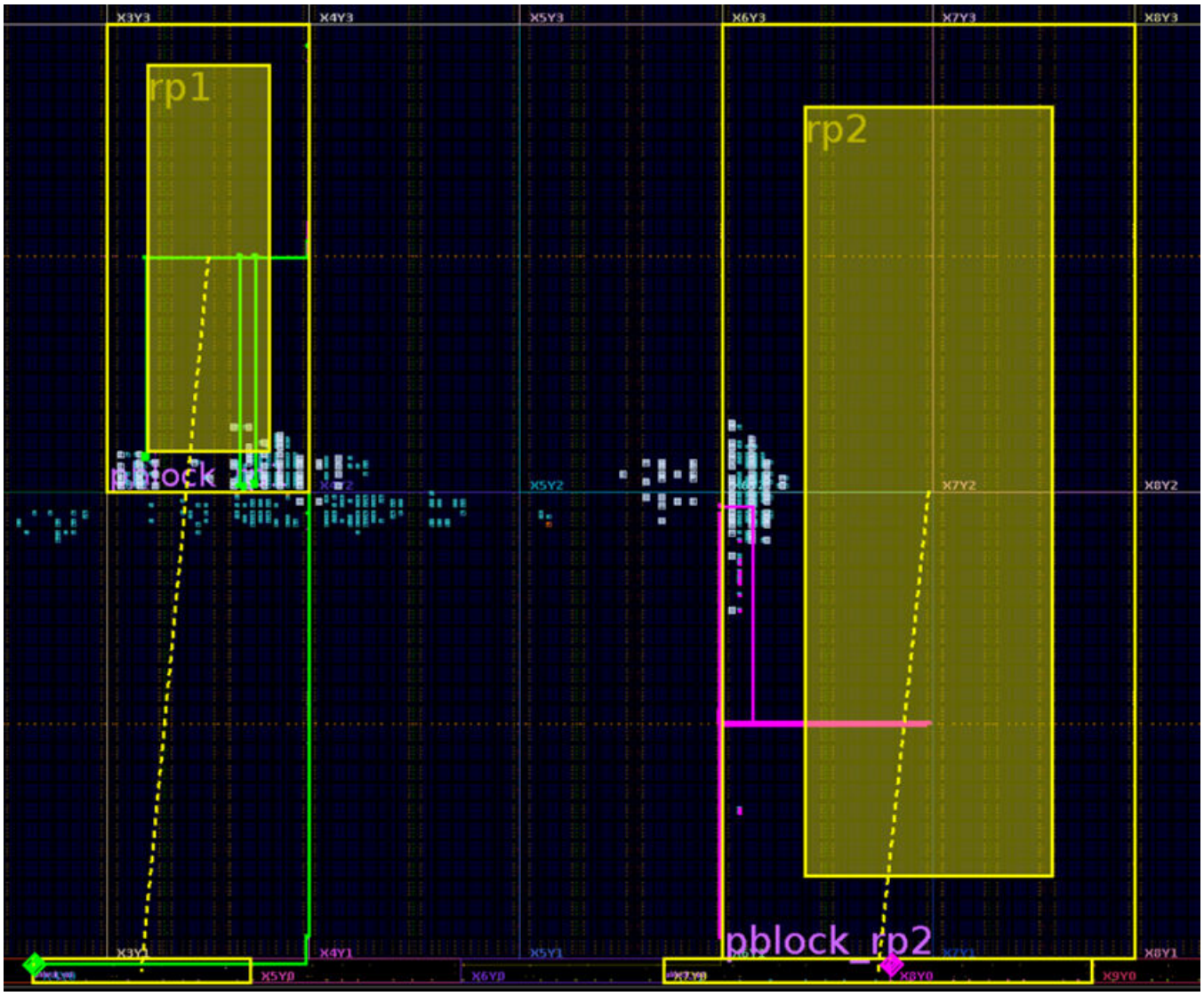
In Versal, majority of clocking resources appears in bottom HSR row. Since most of the clocking resources are located at bottom row of the device, users can have an island for reconfigurable Pblock in the bottom HSR if they need internal clocking resources inside the reconfigurable partition. For example, 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 98: **Clocking Resources**



Clock Routing expansion automatically picks up required clock tiles for routing even though clocking resources are in separate island of Pblock.

Figure 99: **Clock Routing Expansion**



IMPORTANT! *It is recommended to draw contiguous Pblocks like `rp2`. This avoids static region's island-like placement and ease the timing closure.*

Global Clocking Rules

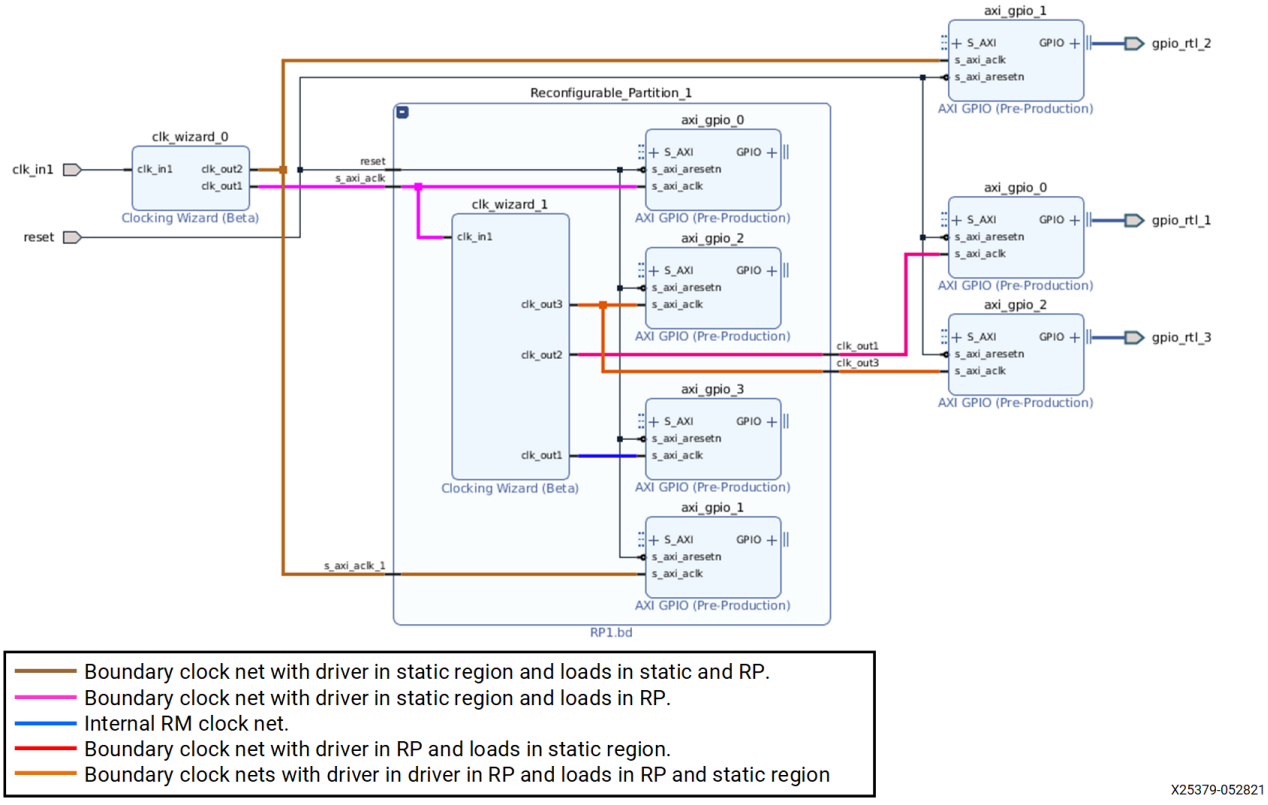
Most of the clocking guidelines of UltraScale+ stay same for Versal. However, Versal Clocking structure provides more clocking primitives and guidelines for better management of skew and meeting timing closure more easily. Refer Design Creation: Clocking Guidelines section in *Versal ACAP Hardware, IP, and Platform Development Methodology Guide* (UG1387) for the detailed understanding of clocking structure in Versal.

This section lists possible clocking topologies supported in the DFX flow. In general, DFX designs clocks can be categorized into two types namely Boundary clocks and Internal clocks.

Table 12: **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's 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 100: **DFX Design Clocks**



DFX behavior for different categories of clock nets is as follows:

- 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

  - This is recommended whenever possible to achieve better skew and optimal clock root placement.

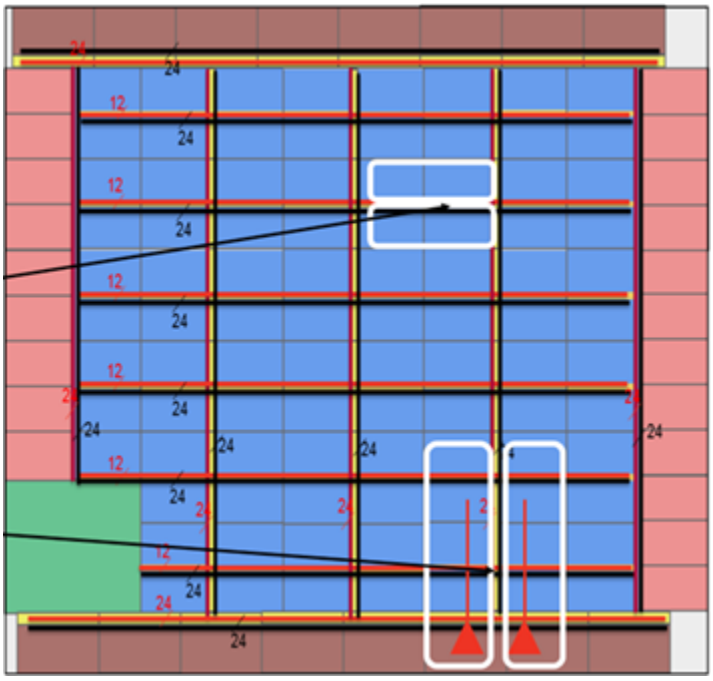
- Boundary Clock Net
  - Boundary clock net track gets locked down after first implementation.
  - The PPLOCs of the boundary clock nets are distributed to all clock regions covered by the RP Pblock.
  - The clock root of the boundary clock net can get placed anywhere in the device, since it can drive both static and RP loads. If the loads of boundary clock are more in static region, it is possible that clock root gets placed in static region.
  - If the first implementation is done using training logic in the RP Pblock, it is possible that boundary clock nets gets locked down after first implementation with sub-optimal clock root location. We recommend using `USER_CLOCK_ROOT` constraint on the boundary clock net to manually constrain the `CLOCK_ROOT` location.

Major enhancements for Clocking in Versal DFX Compared to UltraScale+

This section covers notable changes in Versal clocking. Versal allows clock tile splitting among multiple RPs. However, this is taken care by the tool. User need to range only the clock sources like MMCM and BUFGs to the Pblock and DFX flow automatically pulls in the required clocking tiles for routing.

Figure *Clock Tile Partition* shows `RCLK` row and `VNOC` column sharing among multiple reconfigurable partitions. `RCLK` row sharing among two reconfigurable partitions allow sharing the clock region between two RPs (one above the `RCLK` and one below the RP). The `VNOC` tiles are also shared between multiple reconfigurable partitions. 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 2 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 101: **Clock Tile Partition**



Known Restrictions for Clocking in Versal DFX

- MBUFGCE primitives not allowed for boundary clock nets.
  - MBUFG primitives in Versal allows clock division at the leaf level to reduce clock track utilization and improve timing closure on synchronous CDCs. For DFX designs, MBUFG optimization is allowed only for static clock nets or internal RM clock nets. Boundary clock nets can continue to use BUFGCE_DIV/MMCM/PLL clocking primitives for clock division. However, this will have reduced QoR benefits compared to using MBUFG primitives since latter provides common clock node closer to loads at the leaf level. Hence it is recommended to use MMCM/PLL inside partitions of the DFX design to convert a boundary clock net to internal clock net which can leverage MBUFG optimizations of Vivado.
- Restrictions in clock resource usage due to clock tile splitting.
  - When clock tile is shared between multiple RPs, it is possible that some of the nodes cannot be used for clock routing. To avoid potential unroutability because of tile splitting, DFX flow automatically prohibits usage of few clocking tiles. To avoid this scenario, we recommend keeping atleast one clock region wide gap between multiple RP pblocks if utilization estimation meets the design need.

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.

As a reminder, 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 ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313).

Table 13: **Supported NoC Connectivity in DFX Design**

NoC Endpoints	Connectivity
NoC is completely in static	• RP communicating to NoC in static • RP not communicating to NoC in static
NoC is completely inside RP	• Static communicating to NoC in RP • Static not communicating to NoC in RP
Some NoC endpoints are in static, some NoC endpoints are in RP	• RM use purely for internal communication • With NoC interface path between static and RP
RP to RP NoC communication	Direct NoC to NoC paths between multiple RPs are not allowed in DFX. To keep ownership of such paths in static region, it is required to keep an endpoint in static region (**RP1** → **static** → **RP2**)

NoC Completely in Static

The simplest use case keeps the entire NoC in the static part of the design. An RP may or may not connect directly to the NoC. The NoC itself requires no quiescing, but it is recommended to insert logical decoupling in the PL on the RP boundary. The NoC continues to operate completely during dynamic reconfiguration.

Figure 102: **NoC Completely in Static**

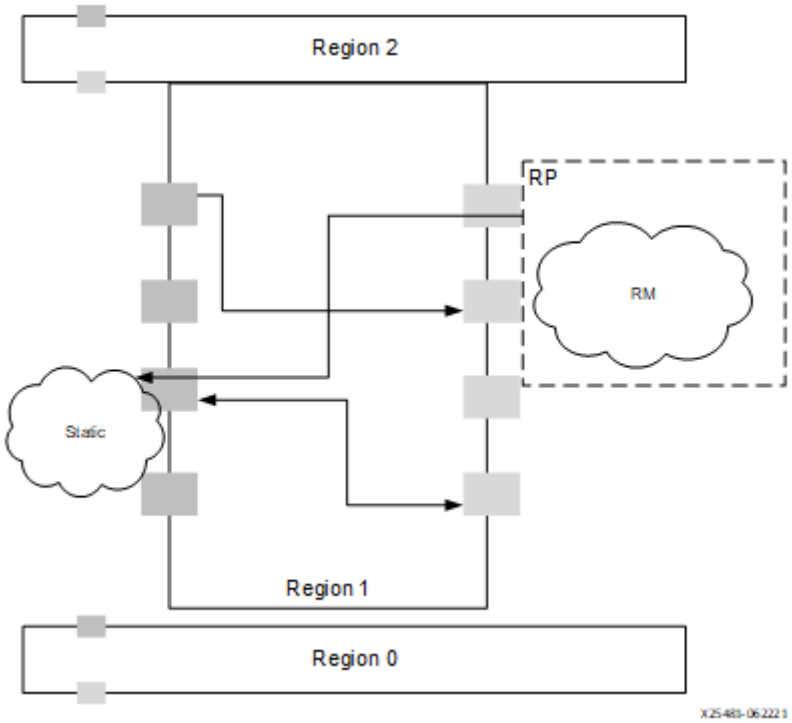


Figure 103: **NoC in Static. No Communication to RP Using NoC. Static-RP Interface is Decoupled Using PL-based Decoupler**

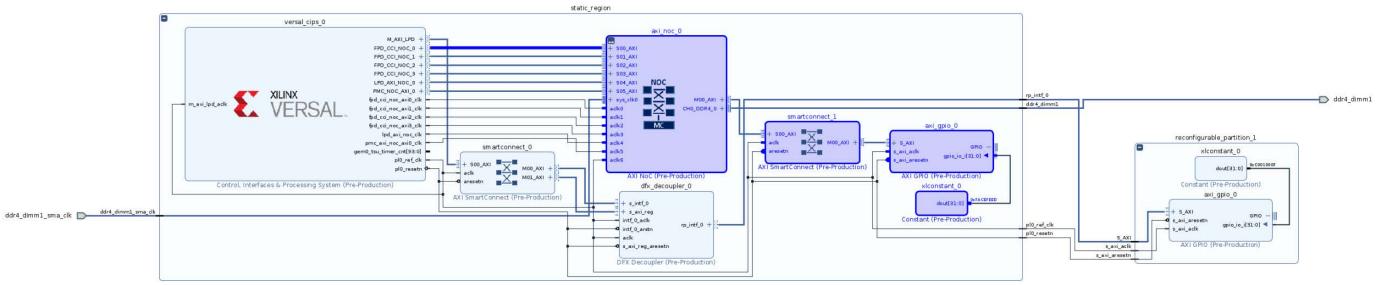
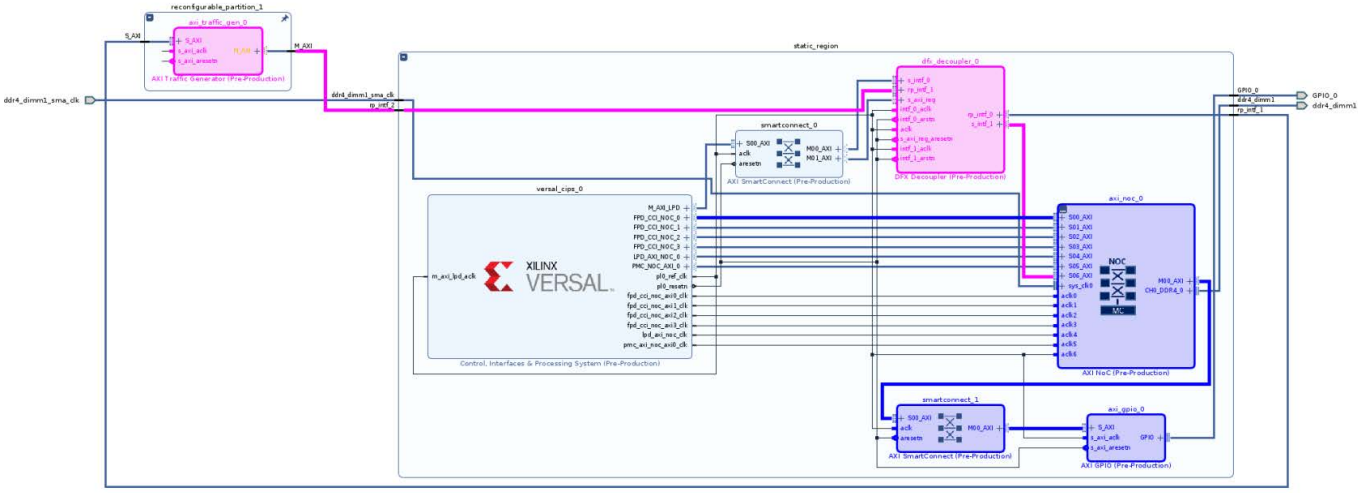


Figure 104: **NoC in Static and RP Communicate to it Using PL-based Decoupler**



NoC Paths Shared between RP and Static

This is the most common use-case with a portion of the NoC in the RP, connected to static portions of the NoC. The NoC configuration details (NMUs, NSUs, and NPS) changes as needed to serve the needs of each RM, but the connections for the static part of the NoC remain fixed.

During reconfiguration, the dynamic part of the NoC is reprogrammed while the static part remains operational. NoC endpoints in the RP quiesce and NoC traffic among them stops during reprogramming. For NoC connections to and from static, the traffic from the RP stops, while the traffic to the RP receives a SLVERR for AXI-MM, drops for AXI-S.

Figure 105: **Part of the NoC in the RP, With Connections to Static NoC**

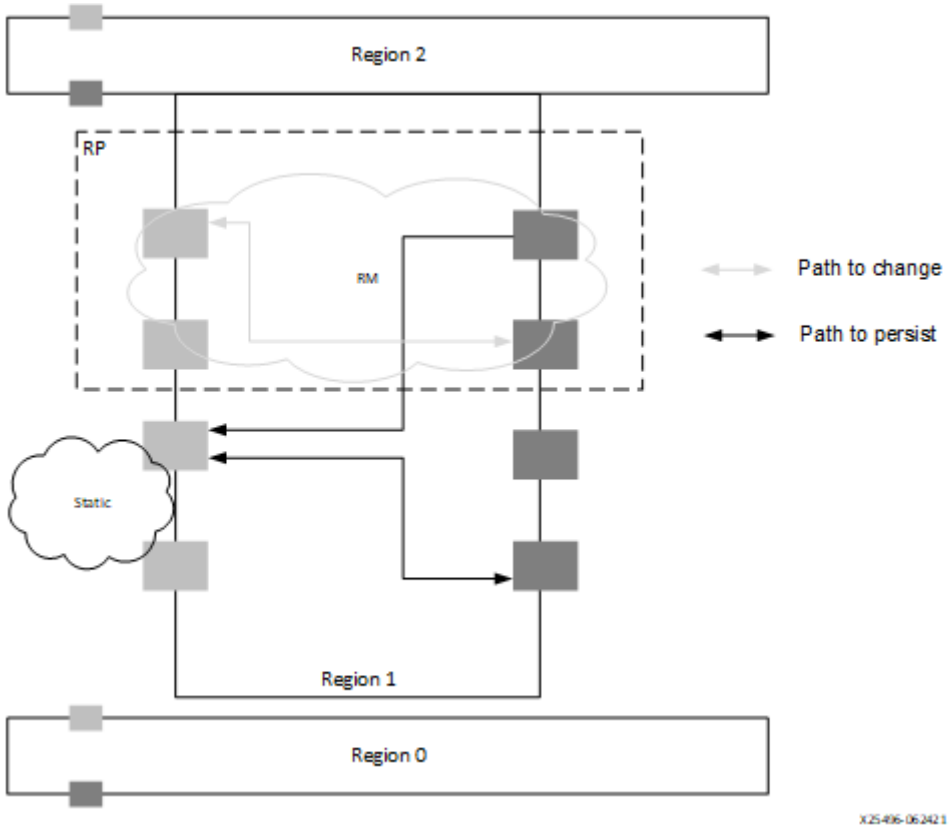
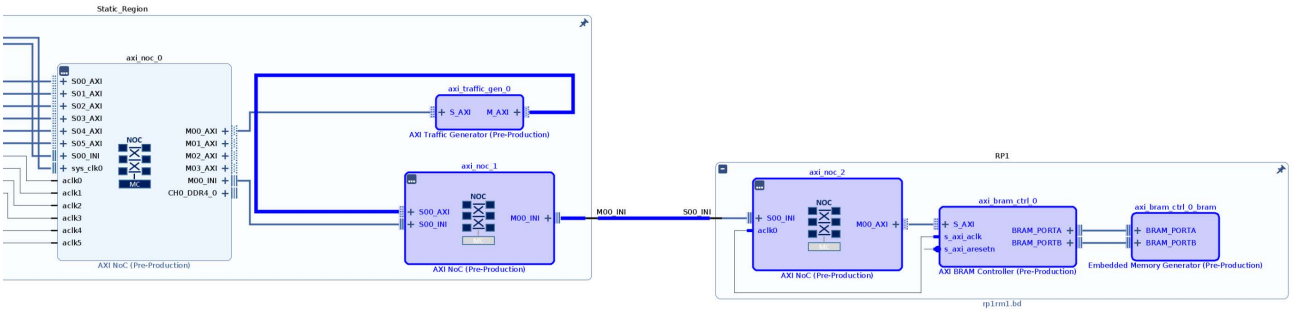
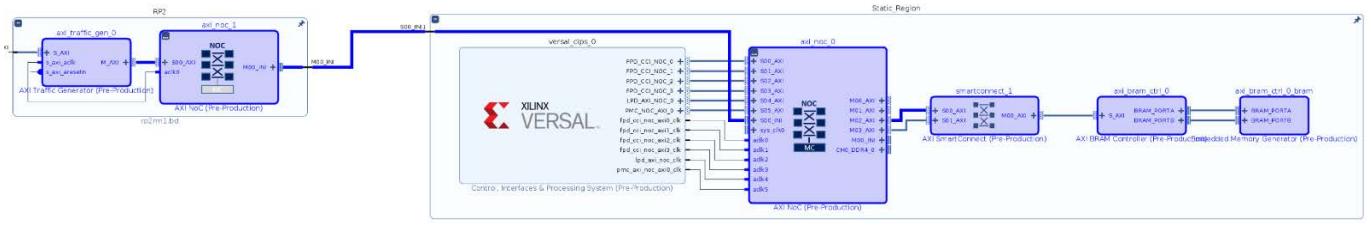


Figure 106: **NMU in RP Communicating to NSU in Static Using NoC INI interface**



***Note*:** No PL- based decoupler is required for INI interface between Static and RP.

Figure 107: **NMU in Static Communicating to NSU in RP Using NoC INI interface**



***Note*:** No PL-based decoupler is required for INI interface between Static and RP.

RP to RP NoC Communication With a Static Endpoint NoC

In this use case, the NoC is used to directly connect two independent RPs. For this case, ensure to establish the interface path in the first design configuration and then not change this path for subsequent configurations.

***Note*:** Ensure to always keep a static logical NoC instance when connecting multiple RPs.

Figure 108: **Part of the NoC in the RP, Connecting to Another RP**

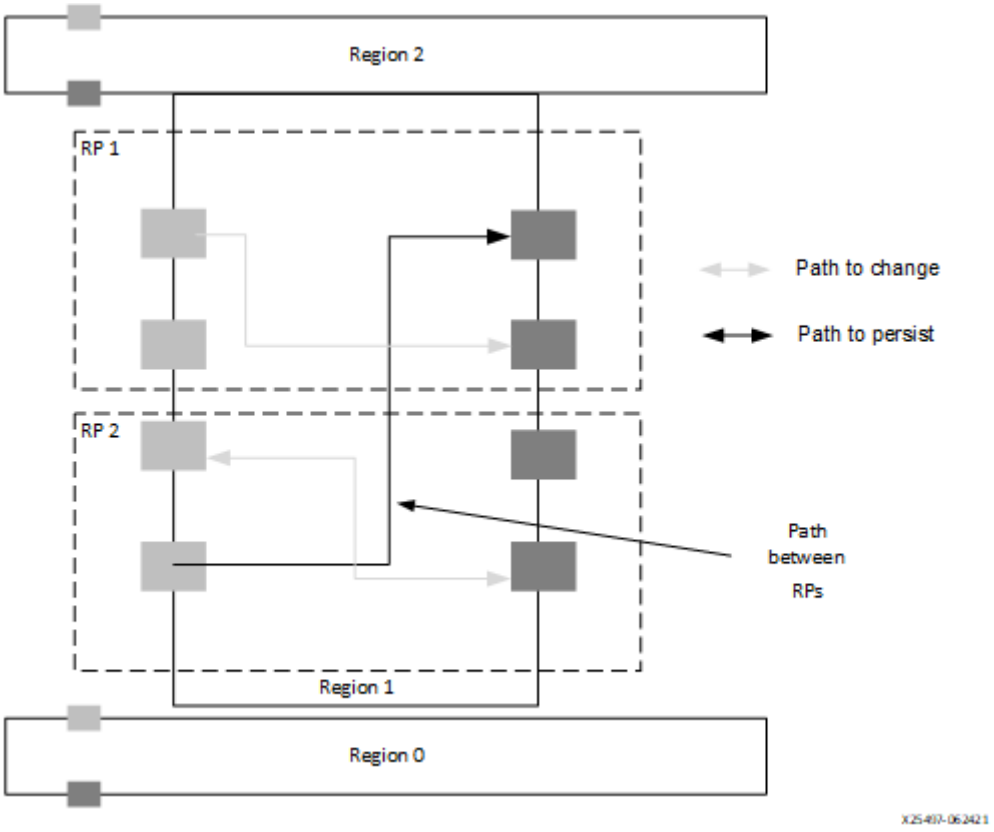
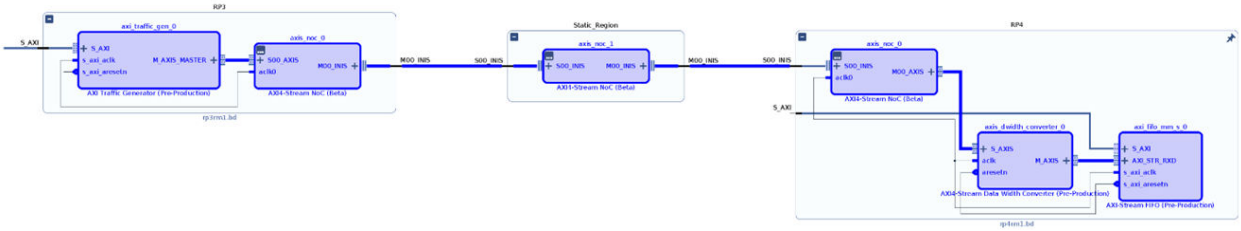


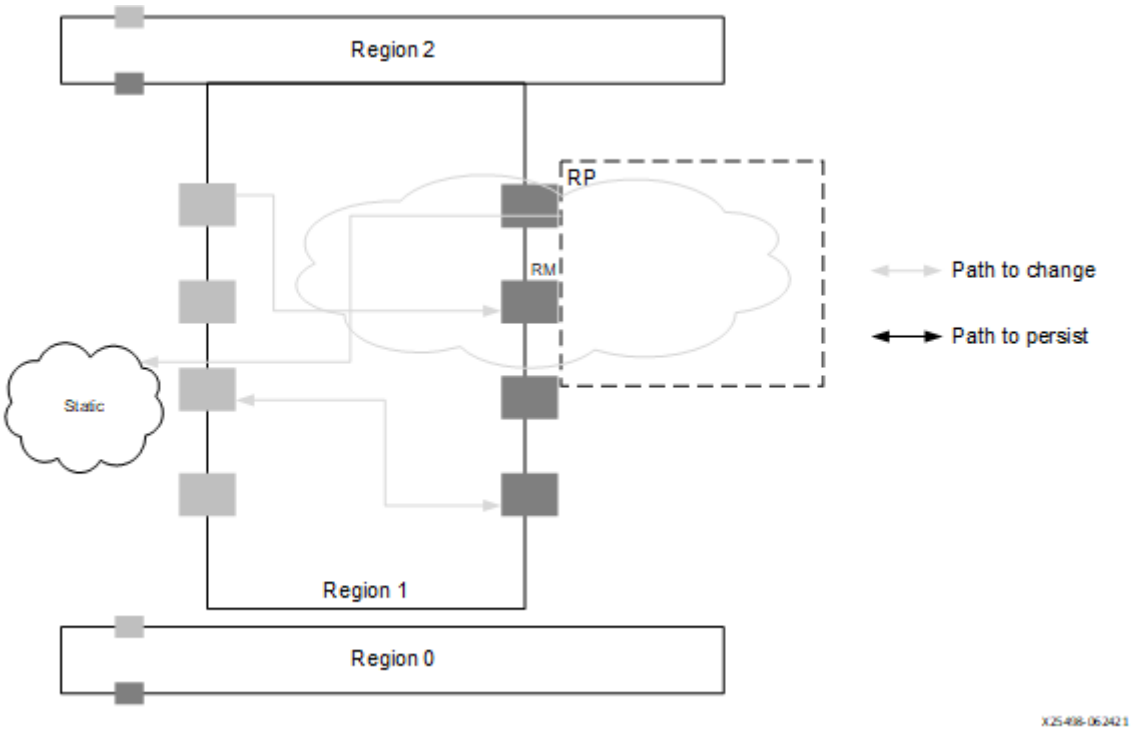
Figure 109: **NMU in RP3 Communicating to NSU in RP4 With a Static Endpoint in between**



NoC Completely in an RP

This is a rare use case, typically reserved for platform designs where the entirety (or close to it) of the programmable logic is in a dynamic region. NoC configuration details (NMUs, NSUs, connectivity) are free to change to service the needs of each unique RM. The NoC will be completely reprogrammed during dynamic reconfiguration. All NoC endpoints quiesce and all NoC operations cease during reprogramming.

Figure 110: **NoC Completely in an RP**



NoC DFX DRCs in IP Integrator: Problems and solutions

NoC Paths must be Owned by Static if they have a Static NMU.

- **NoC DFX DRC**:

```
[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).
The path from NoC Master (axi slave) /static_region/axi_noc_0/S00_AXI to
/rp1/S00_INI (INI strategy set by user) is not static.
To fix: set the INI strategy on INI interfaces for this path to 'load'
```

- **Reason**: The NMU (for example, corresponding to NoC S00_AXI) is configured to route to a specific NSU. By having the static NMU own the path, the part of the NMU that determines the destination NSU is also static and non-changing. If the RM owned the path then a part of the static NMU would also have to be reconfigured when the path is changed. In order to keep the static NMU fully static the NMU must own the path.

  Path ownership of the NoC path is crucial in determining the Quality of Serivce (QoS) for that path. For the NoC Paths across the DFX boundary, if the NoC master is in the static region, Quality of Service should be determined by the static region NoC instance. According to NoC rules, QoS is decided by the NoC instance who owns the path. This also enables the user to selectively tune the QoS and Bandwidth to required value across the DFX boundary and lock it down for multiple RM variants.

- **Solution**: When the NoC INI strategy is selected as Single Load, the Driver (NMU) owns the DFX path and decide the QoS. This is required for the boundary paths of DFX when driver NMU is in the static region. When the NoC INI strategy is selected as Single Driver, the Load (NSU) will own the DFX path. This is not allowed for DFX boundary path since the NSUs in reconfigurable partition can changed over different RM variants and boundary path cannot be locked down with a definite QoS and Bandwidth if the ownership is not at the static side driver.

  The IP integrator provides the NoC INI strategy option **Auto** where it automatically fixes the strategy to the legal value. For a NoC INI based DFX boundary path where master is in static region, User can select NoC INI strategy in static region to be **Auto**. This is the default option. IP integrator during validation will automatically compute INI strategy to the right value.

  ***Note*:** Automatic computation of NoC INI strategy is currently supported only for top BDs and not for child BDCs, unless the child BD's strategy can be determined unambiguously due to either multiple drivers or multiple loads within the child BD.

Static NoC Slave can be Driven by only 1 RP When the NoC Path is Wwned by RP

- **NoC DFX DRC**:

```
[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). NoC Slave (axi
master)
/static_region/axi_noc_1/M00_AXI connects to >1 RP that owns noc paths:
/rp1/axi_noc_2/S00_AXI
/rp2/axi_noc_3/S00_AXI. To fix: Only have one RP drive the NoC slave
```

- **Reason**: Reconfigurable Partitions may be configured with different reconfigurable modules. When there are multiple reconfigurable partitions in a design, both the RPs can be swapped independently. Hence, for a NoC Path between static and RM, if the NoC slave is in the static region and the RM owns the path, then the path must be owned by only one RM. Multiple RPs cannot take the ownership of a single NoC path.

- **Solution**: You are may hit the above situation when NoC INI strategy of NMU in RPs and NSU in Static region of that boundary path is set to **Single Load, Driver (NMU) owns the DFX path and has QoS**. A solution is to keep the path ownership of path to NMU in DFX region by adding another logical NoC in Static region: Add a new logical NoC in the Static Region and separate the boundary NoC path into two separate parallel boundary NoC paths from RPs to Static region.

NoC Endpoints in the RP Boundary can Connect to Only One Type of INI Strategy

- **NoC DFX DRC**:

```
[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 Slave (axi master)
/static_region/axi_noc_3/M00_AXI has both INI strategy load /rp2/
M01_INI_0 and
INI strategy driver /rp2/M00_INI.
```

- **Reason**: Each reconfigurable partition is expected to be Out of Context synthesized independently. When they are stitched together at the top level, they should have matching boundary pins. The boundary should meet the NoC rules with respect to Driver and Load. This is more critical when both NMU and NSU are in different reconfigurable partitions where respective OOC synthesis might not be aware of the NOC INI strategy of the other RP. Because of this reason, Recommendation is to keep all INIs of NoC at the boundary of the RP to be of same strategy. This also helps to prevent duplicate paths of NoC which is illegal in DFX design.
- **Solution**: Recommendation is to keep all INIs of the NoC at the boundary inside RP to be of same strategy. With this, the ownership of the boundary path can be handed over to the NoC outside the RP. If all the RPs' boundary NoC's INI keeps the same strategy, after stitching back individual OOC RPs, the INI strategy of AXI NoC in the static will own these boundary paths. This is recommended as QoS of the boundary paths will be locked down by the static region NoC.

NoC Endpoint can be Exported Only Once Across a DFX Boundary

- **DRC for NMU ERROR**:

```
[BD 41-2577] NoC DFX Rule: A NoC endpoint can only be exported once
across a DFX boundary.
This means a NoC Master can only have one INI with strategy=driver
crossing a DFX boundary.
NoC Master (AXI slave) /h1/axis_noc_0/S00_AXIS has
the following strategy=driver INIs crossing a DFX boundary: /axis_noc_1/
S01_INIS /axis_noc_1/S00_INIS
```

- **DRC for NSU**:

```
ERROR: [41-2577] NoC DFX Rule: A NoC endpoint can only be exported once
across a DFX boundary.
This means a NoC Slave can only have one INI with strategy=load crossing
a DFX boundary.
NoC Slave (AXI master) /h2/axis_noc_1/M00_AXIS has the following
strategy=load
INIs crossing a DFX boundary: /axis_noc_0/M00_INIS/axis_noc_0/M01_INIS
```

- **Reason**: For a boundary NoC path between Static and RP, For NSU, there can be only one INI with strategy **Single Load, Driver (NMU) owns the DFX path**. For NMU, there can be only one INI with strategy **Single Driver, Load (NSU) owns the DFX path.**

  If there are more than 1 INI with (strategy = Load for the NSU) or (strategy = Driver for the NMU) in boundary path, it means the path from NMU to the specific NSU is coming through multiple NoC INI endpoints. This is not allowed in the DFX design because In DFX flow, we must lock down the boundary path with one unique INI endpoint to NSU that later iterations of the RMs can use. One conceptual model is to consider that for strategy=load, the load (NSU/MC) is exported at the RM boundary, and likewise for strategy=driver, the driver (NMU) is exported at the RM boundary. The endpoint can only be exported once.

- **Solution:** Ensure that an INI path crosses the RM boundary only once.

A Reconfigurable Partition cannot have NoC Passthrough

- **NoC DFX DRC**:

```
[BD 41-2476] NoC DFX Rule: An RP cannot have a NoC passthrough.
Path across '/r4_mid' from /r4_mid/S00_INI_0 to /r4_mid/M00_INI_0 is a
passthrough.
To fix: Ensure an RP has a noc endpoint.
```

- **Reason**: To avoid NoC path issues when reconfiguring an RM with a passthrough, passthroughs in an RM are not allowed. Please note that a NoC passthrough is allowed in the static.

- **Solution**: If you are not expecting to have any NoC Endpoints in the reconfigurable partition for a boundary NoC path, you should not keep a passthrough of NoC path through RP. However, if you are planning to reserve a NoC path in the reconfigurable partition for future RM iterations, you should also add a NoC endpoint in the RM in the first implementation itself as a place holder.

NoC Path between Multiple RPs should have Static Logical NoC Instance

- **NoC DFX DRC**:

```
[BD 41-2478] NoC DFX Rule: NoC path between two RPs must be static.
Path from /r5_rp1/axi_noc_0/S00_AXI to /r5_rp2/axi_noc_1/M00_AXI is not
owned by the static (or parent).
It is owned by /r5_rp1. 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.
```

- **Reason**: In this case there is an NMU in one RP connected through a passthrough in the static to a NSU in another RP. The static must own the NoC path. Each RP can be reconfigured independently and the NoC path between them must be static so it can remain valid regardless of the configuration of each RP.

  NoC paths that communicate between multiple RPs should have static logical NoC instances. Otherwise it is not possible to lock down the NoC path with a specific requirement. Since both RPs can be independently OOC synthesized, it is possible to create a final NoC solution which may be not compatible. Hence it is critical to have a Static endpoint for all NoC paths that communicate between RPs.

- **Solution**: Configure the a NoC pass-through in the static so it owns the path, by having the INI input with `strategy=driver` and the INI output with `strategy=load`.

Static NSU/MC must be driven by an RM must have an NMU in the RM

- **NoC DFX DRC**:

```
[BD-41-2616]: Static NSU/MC should be driven by an NMU in the RM.
```

- **Reason**: The tool requires a complete NoC Path for NoC configuration and validation. This error should not be seen because the tool will auto-add a NoC NMU when needed. During design development, you can have a NoC with a non-driven INI output, where that output exits the RM and drives a static NSU/MC. In this case the NoC with the non-driven INI output will auto-add a stub NMU to complete the NoC path. It is not legal to have an INI external output port that is not driven.
- **Solution**: Even if a specific reconfigurable module does not use that specific NoC to talk to static region, it is recommended to add a place holder NoC NMU in the RP so that NoC path is complete and tool can do NoC validation.

Logical Decoupling

Because the reconfigurable logic is modified while the device is operating, the static logic connected to outputs of RM ignores 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 reset. 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. A boundary could be within the PL, within the NoC, at the PS-PL interface, or a RP could have interface ports in each of these categories.

Decouple at the PL Boundary

A common design practice to mitigate this issue is to register all output signals (on the static side of the interface) from the RP. An enable signal is used to isolate the logic until it is completely reconfigured. Other approaches include a simple 2-to-1 MUX on each output port, or to higher level bus controller functions.

Two pieces of IP are available from Xilinx to provide decoupling capabilities in the PL. First, the DFX Decoupler IP allows user to insert muxes to easily and efficiently decouple AXI4-Lite, AXI4-Stream, and custom interfaces. This IP simply disables key signals to prevent unwanted activity on the RP boundary. Second, the DFX AXI Shutdown Manager IP <to fill>. More information about the DFX Decoupler IP is available on the Xilinx website.

Decouple in the NoC

If the boundary falls within the NoC, NoC quiescing is automatic only if NMU/NSU is inside RP. If NOC/PL AXI interface is at RM boundary, decoupler is still required. In such case, it is recommended to put the interface NOC in RP to avoid the use of AXI decoupler. Then the connection across partition is not AXI anymore, but NoC internal path through INI.

Decouple at the PS-PL interface

If the boundary between static (PS) and dynamic (PL) is specifically at the PS-PL interface, following two use cases arise:

- If the entire PL is included in the dynamic region, partial reconfiguration includes a power domain shutdown such that the entire PL is temporarily powered down.
- 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). It is also useful to consider the down-time performance effect of a dynamic module (that is, the unavailability of any shared resources included in the dynamic module during or after reconfiguration). This is the most common scenario.

Versal Use Cases

The unique device structure of Versal ACAP open new opportunities for dynamic reconfiguration and related solutions. Versal-specific use cases are listed in this section.

Classic SoC Boot

Classic SoC Boot solution enable designers to boot the processors in the Scalar Engines of a Versal device and access DDR memory before the programmable logic (PL) in the Adaptable Engines is configured. This allows DDR-based software like Linux to boot first followed by the PL, which can configured later if needed via any primary or secondary boot device or through a DDR image store. The Classic SoC Boot feature is intended to treat Versal boot sequences similar to Zynq UltraScale+ MPSoCs. This solution is built using a Dynamic Function eXchange flow through the Vivado IP integrator, which includes automatic floorplan generation and flow-specific design rule checks (DRCs). The entire PL is dynamic and can be completely reloaded while any operating system and DDR memory access remain active.

Classic SoC Boot is incompatible with use of the CPM, including PCIe controller, DMA features, and dynamic reconfiguration of sub-regions of the PL is not yet supported. For more information on Classic SoC Boot including design requirements and a tutorial walkthrough, see the Classic SoC Boot Tutorial available from the Xilinx GitHub repository.

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.
- **MCAP:** (UltraScale™ and UltraScale+™ devices only) Provides a dedicated connection to the configuration engine from one specific PCIe® block per device.
- **PCAP:** The primary configuration mechanism for Zynq®-7000 SoC and Zynq UltraScale+ MPSoC designs.
- **JTAG:** A good interface for quick testing or debug. Can be driven with the 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 14: **Supported Configuration Ports**

Configuration Mode	7 series	Zynq	UltraScale	UltraScale+	Zynq UltraScale+ MPSoC
JTAG	Yes	Yes	Yes	Yes	Yes
ICAP	Yes	Yes	Yes	Yes	Yes
PCAP	N/A	Yes	N/A	N/A	Yes
MCAP	N/A	N/A	Yes	Yes	Yes
Slave Serial	Yes	N/A	Yes	Yes	N/A

Table 14: **Supported Configuration Ports** *(cont'd)*

Configuration Mode	7 series	Zynq	UltraScale	UltraScale+	Zynq UltraScale+ MPSoC
Slave SelectMap	Yes	N/A	Yes	Yes	N/A
SPI (any width)[1]	No	N/A	No	Yes	N/A
BPI sync mode	No	N/A	No	Yes	N/A
BPI async mode	Yes	N/A	Yes	Yes	N/A
Master modes	No	N/A	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.

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 this link in 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)

Bitstream Type Definitions

When designs are compiled for Dynamic Function eXchange in Xilinx 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

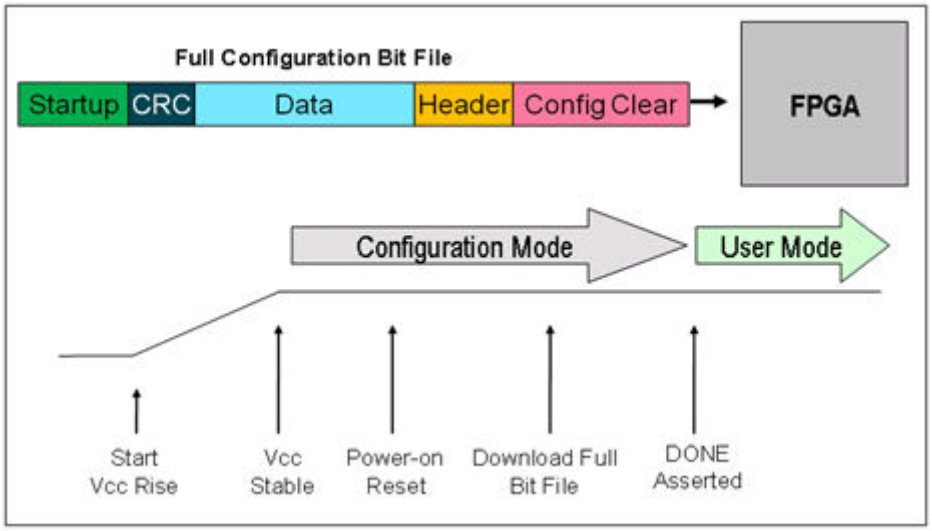
All DFX designs start with standard configuration of the full device using a full configuration bitstream. The format and structure is no different than for a flat design solution (with one exception), and there is no difference in how this bitstream can be used to initially program the FPGA. The one 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 111: **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).

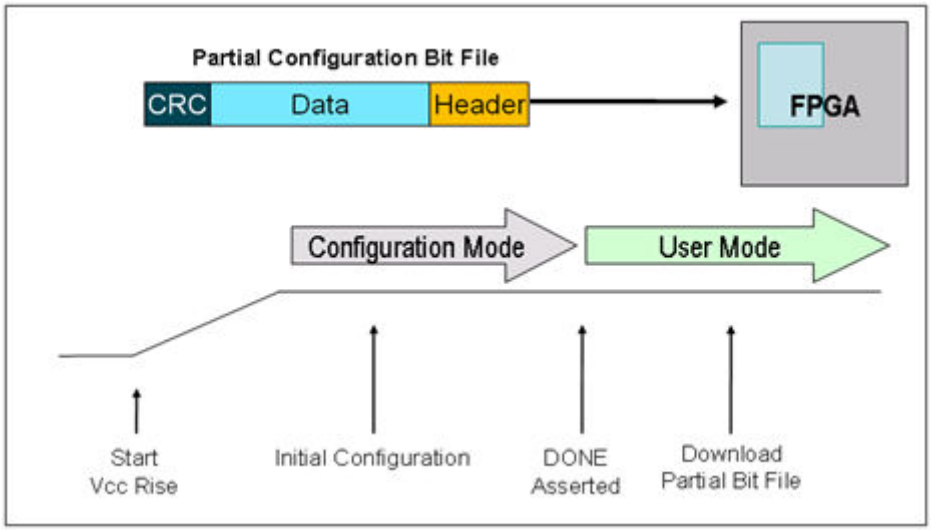
Partial bitstreams are automatically created when `write_bitstream` is run on a DFX configuration. Each partial bitstream file name references your top-level design name, plus the Pblock name for the RP, plus `_partial`. For example, for a full design bit file `top_first.bit`, a partial bit file could 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. Partial Bitstreams illustrates this process.

Figure 112: **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 graybox, 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 graybox 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 graybox 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 graybox 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, just as they can be for 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. Note that 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).

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 a Xilinx 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 utilizes 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:

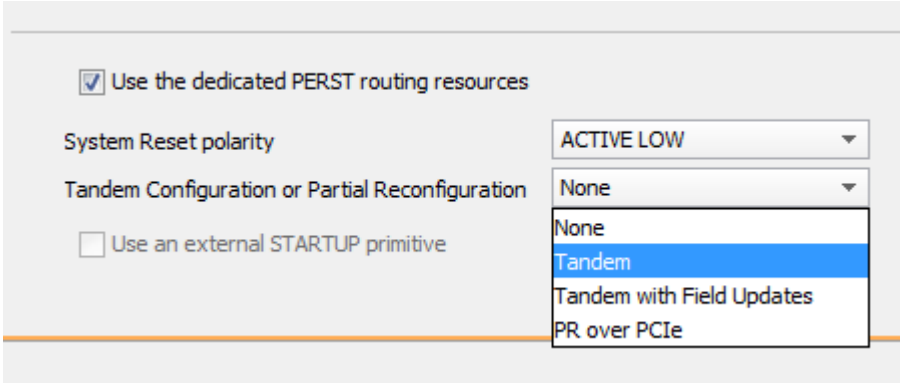
- **Tandem PROM:** Loads the single two-stage bitstream from the flash.
- **Tandem PCIe:** Loads the first stage bitstream from flash, and deliver the second stage bitstream over the PCIe link to the MCAP.
- **Tandem with Field Updates:** After a Tandem PROM (UltraScale only) or Tandem PCIe (UltraScale 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
   - **PR 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+
   - **PR 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 below in Table 15: UltraScale: PCIe Block and Reset Locations Supporting DFX, by Device. 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 may 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 will start 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 this link 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 15: **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
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 16: **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
VU7P	PCIE40E4_X1Y0	Production
VU9P	PCIE40E4_X1Y2	Production
VU11P	PCIE40E4_X0Y0	Production
VU13P	PCIE40E4_X0Y1	Production
VU19P	PCIE4CE4_X0Y2	Production
VU23P	PCIE4CE4_X0Y0	Production

Table 16: **UltraScale+: PCIe Block Locations Supporting DFX, by Device** *(cont'd)*

Device	Package PCIe Block	Status[1]
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
Zynq 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 Xilinx 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 the answer record, *Bitstream Loading across the PCI Express Link in UltraScale Devices for Tandem PCIe and Partial Reconfiguration* (AR# 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 (v1.0)* (XAPP1338) for more information and an example design.

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.

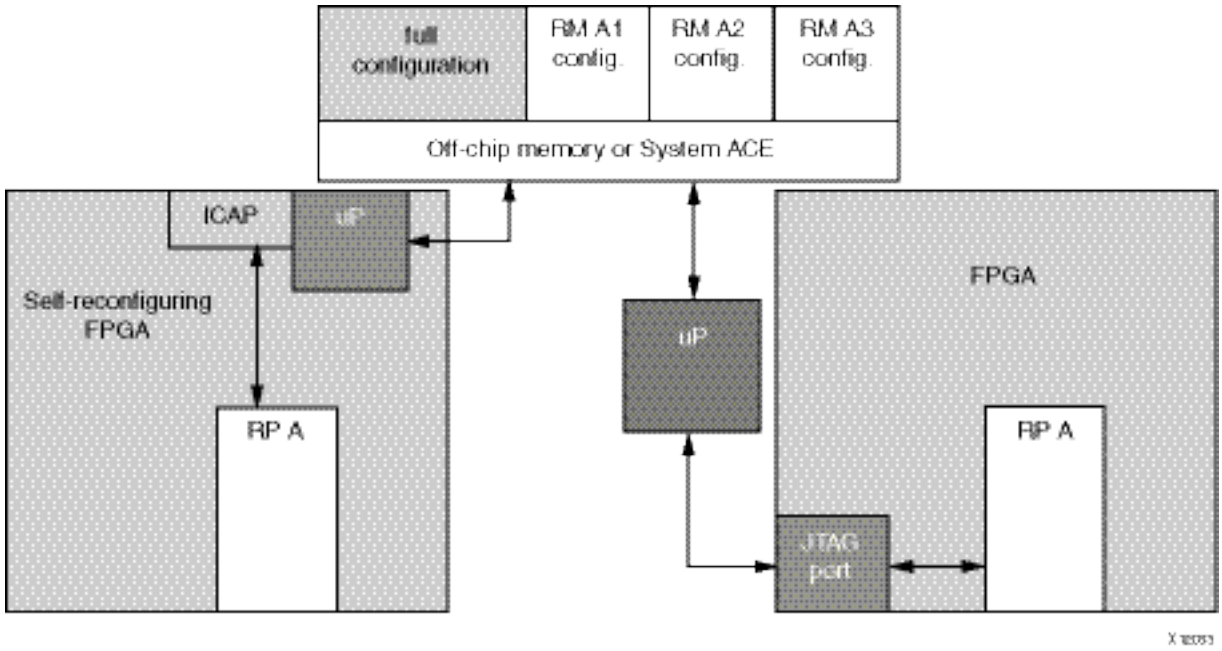
System Design for Configuring an FPGA

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. System Design for Configuring an FPGA shows the process of configuring through a microprocessor.

Figure 113: **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, but only when the initial configuration was also encrypted. The same key must be used for all bitstreams.
- If the initial configuration of a device is encrypted, unencrypted partial bitstreams can be used only if they are delivered to the ICAP.
- Bitstream authentication for partial bitstreams is not supported for or Kintex devices, only for Zynq devices.
- 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 Xilinx 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 Xilinx 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.

Xilinx 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.

- Base Regions in 7 series FPGAs are:
  - **CLB:** 50 high by 1 wide
  - **DSP48:** 10 high by 1 wide
  - **Block RAM:** 10 high by 1 wide
- Base Regions in UltraScale and UltraScale+ FPGAs are:

- **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)

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 Table 17: Maximum Bandwidths for Configuration Ports in 7 Series Devices,Table 20: ICAP "O" Port Bits, and Table 21: ICAP Sync Bits.

Table 17: **Maximum Bandwidths for Configuration Ports in 7 Series Devices**

Configuration Mode	Max Clock Rate	Data Width	Maximum Bandwidth
ICAP	100 MHz	32 bit	3.2 Gb/s
SelectMAP	100 MHz	32 bit	3.2 Gb/s
Serial Mode	100 MHz	1 bit	100 Mb/s
JTAG	66 MHz	1 bit	66 Mb/s

Table 18: **Maximum Bandwidths for Configuration Ports in UltraScale Devices**

Configuration Mode	Max Clock Rate	Data Width	Maximum Bandwidth
ICAP/MCAP	200 MHz	32-bit	6.4 Gb/s
ICAP/MCAP (SSI)[1]	125 MHz	32-bit	4.0 Gb/s
SelectMAP	125 MHz	32 bit	4.0 Gb/s
Serial Mode	150 MHz	1 bit	150 Mb/s
Serial (SSI Devices)	100 MHz	1 bit	100 Mb/s
JTAG	50 MHz	1 bit	50 Mb/s
JTAG (SSI Devices)	20 MHz	1 bit	20 Mb/s

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. Note that configuration clock frequency maximums may be less than these values depending on speed grade or operating voltage. Consult the Data Sheet for your target device for more information.

Table 19: **Maximum Bandwidths for Configuration Ports in UltraScale+ Devices**

Configuration Mode	Max Clock Rate	Data Width	Maximum Bandwidth
ICAP/MCAP	200 MHz	32-bit	6.4 Gb/s
ICAP/MCAP (SSI)[1]	125 MHz	32-bit	4.0 Gb/s

Table 19: **Maximum Bandwidths for Configuration Ports in UltraScale+ Devices** *(cont'd)*

Configuration Mode	Max Clock Rate	Data Width	Maximum Bandwidth
BPI	125 MHz	16-bit	2.0 Gb/s
QSPI	125 MHz	4-bit	500 Mb/s
SelectMAP	125 MHz	32-bit	4.0 Gb/s
Serial Mode	125 MHz	1-bit	125 Mb/s
JTAG	66 MHz	1-bit	66 Mb/s
JTAG (SSI Devices)	20 MHz	1-bit	20 Mb/s

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. Note that configuration clock frequency maximums may be less than these values depending on 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 2019.2
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 20: **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.

Table 20: **ICAP "O" Port Bits** *(cont'd)*

ICAP "O" Port Bits	Status Bit	Meaning
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.

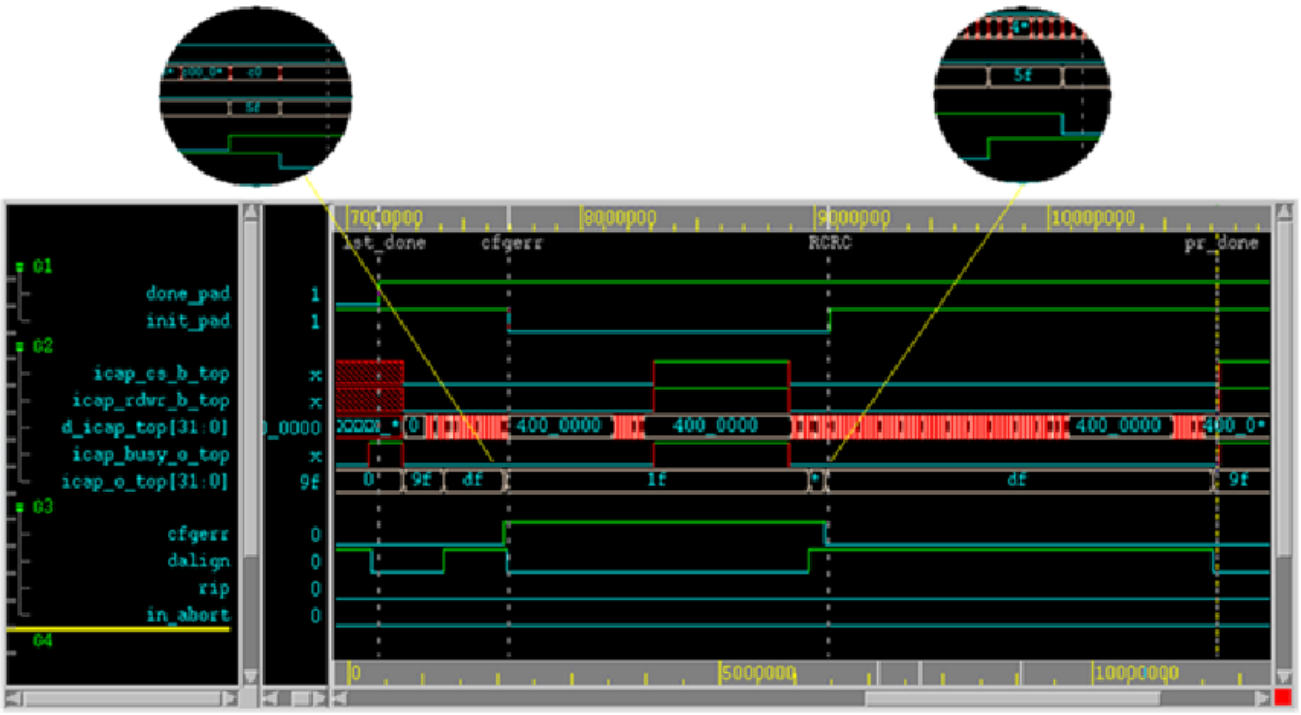
Table 21: **ICAP Sync Bits**

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.

Configuration Debugging 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 114: **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 will see multiple PRDONE events due to the construction of these partial bitstreams. Instead, use the EOS pin on the STARTUP block 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.

Using Vivado Debug Cores

Vivado debug cores (ILA, VIO, etc.) 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 will be inferred if these exact names are used. Examples in Verilog and VHDL are below.

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()
);
```

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` will 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 graybox 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 Vivado implementation tools.

In the syntax shown below, do not change anything other than the port names in order to assign Debug ports. These attributes are to be used in all RM top level source files.

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;
```

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";
```

There are currently two limitations to this solution:

1. All debug cores within RMs must be instantiated. The MARK_DEBUG insertion flow is not yet supported.
2. A graybox 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 this link in *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947).

Configuration Solutions for Versal Devices

Dynamic Function eXchange is managed by PLM services running in the PMC. These services control the events needed before, during, and after dynamic reconfiguration of the NPI and CFI resources through-out 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 ACAP System Software Developers Guide* (UG1304) for detailed discussion on primary and secondary boot modes.

Partial PDI Creation and Details

Unlike UltraScale+ and prior architectures, bitstreams are not used to program Versal devices. The Versal ACAP PMC uses a proprietary boot and configuration file format called the programmable device image (PDI) to program and configure the Versal ACAP. The PDI consists of headers, the PLM image, and design data image partitions to be loaded into the Versal ACAP. 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 ACAP System Software Developers Guide* (UG1304).

Supported Boot Modes and Performance

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. They can also be passed over slave boot interfaces for JTAG or PCIe, or SelectMAP use cases.

Table 22: **Primary and Secondary Boot Interfaces**

Configuration Mode	Type	Bandwidth	Authentication and Encryption
Primary			
JTAG	Slave	12.5 MB/s	No
QSPI	Master	150 MB/s	Yes
OctalSPI	Master	400 MB/s	Yes
SD	Master	100 MB/s	Yes
eMMC	Master	200 MB/s	Yes
SelectMap x32	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

Programming Image Compression

By default, all full and partial PL programming images are compressed. Versal uses a new compression algorithm that is a different approach than the prior architectures. Versal PDI compression reduces the configuration time in most cases.

Reduced configuration time and decreased configuration storage size requirements is observed when compression is used for the following interfaces:

- Less than 2.56 GB/s (in the slowest speed grade)
- 3.2 GB/s (in medium and fast speed grades) for QSPI, OSPI, SD, eMMC, JTAG, and SelectMAP

For this reason, compression is enabled by default.

Note: The configuration time is affected when Versal programmable device image (`.pdi`) 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 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 the high-speed interfaces that requires shortest programming time when images are delivered from DDR memory or through PCIe.
- **Use compression:** For the 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 TRUE [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 supported for certain DFX use cases.

- For DFX designs that contain two or more RP, this feature is automatically disabled.
- For DFX designs that contain a single RP and target the VC1902 and VM1802, the feature is supported and it is ON by default.
- For DFX designs that contain a single RP and target the VP1202 or VP1802, the NoC Clock Gating feature must be disabled by the user. Execute the following procedure to disable the feature:

  ```
  set_param noc.enableNOCClockGating 0
  ```

  Disabling this feature allows the partial PDI to 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.

Known Issues and Limitations

This is a list of issues that might be encountered when using Dynamic Function eXchange (DFX) in the current Vivado® Design Suite release. If you encounter any of these issues, or discover any others, contact Xilinx® Support and send an example design that shows the issue. These test cases are very helpful to improve the overall solution.

Report to Xilinx 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.

- 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 Xilinx Support for a workaround.
- 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 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.
- Engineering Silicon (ES) for UltraScale™ or UltraScale+™ devices does not officially support Dynamic Function eXchange. To investigate the capabilities of DFX on ES devices, contact Xilinx Support for advice.

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. 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, then the Debug Hub can be inferred as described in Using Vivado Debug Cores.
- UpdateMEM does not support partial or Tandem bitstreams. To associate memory files with DFX designs, the ELF Association flow must be applied. See this link in the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) for details.
- The Soft Error Mitigation (SEM) IP core is supported in conjunction 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, as its clock will stop once 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 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 may 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 PR 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.

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 graybox 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, Xilinx recommends using the `-no_partial_bitfile` switch of the `write_bitstream` command to avoid producing partial bitstreams.

Chapter 13

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

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

List of Supported Devices

Table 23: **List of Supported Devices in the Current Release**

Device	Variants*
Artix-7	
xc7a75t	A, L
xc7a100t	A, L, Q
xc7a200t	L, Q
Kintex-7	
xc7k70t	L
xc7k160t	A, L
xc7k325t	L, Q, QL
xc7k355t	L
xc7k410t	L, Q, QL

Table 23: **List of Supported Devices in the Current Release** *(cont'd)*

Device	Variants*
xc7k420t	L
xc7k480t	L
Virtex-7	
xc7v585t	Q
xc7v2000t	
xc7vx330t	Q
xc7vx415t	
xc7vx485t	Q
xc7vx550t	
xc7vx690t	Q
xc7vx980t	Q
xc7vx1140t	
xc7vh580t	
xc7vh870t	
Zynq-7000	
xc7z007s	
xc7z010	A
xc7z012s	
xc7z014s	
xc7z015	
xc7z020	A, Q
xc7z030	A, Q
xc7z035	
xc7z045	Q
xc7z100	Q
Kintex UltraScale	
xcku025	
xcku035	
xcku040	Q
xcku060	Q, QR
xcku085	
xcku095	Q
xcku115	Q
Virtex UltraScale	
xcvu065	
xcvu080	
xcvu095	
xcvu125	
xcvu160	

Table 23: **List of Supported Devices in the Current Release** *(cont'd)*

Device	Variants*
xcvu190	
xcvu440	
Kintex UltraScale+	
xcku3p	
xcku5p	Q
xcku9p	
xcku11p	
xcku13p	
xcku15p	Q
xcku19p	
xcvu3p	Q
xcvu5p	
xcvu7p	Q
xcvu9p	Q
xcvu11p	Q
xcvu13p	Q
xcvu19p	
xcvu23p	
xcvu27p	
xcvu31p	
xcvu33p	
xcvu35p	
xcvu37p	Q
xcvu45p	
xcvu47p	
xcvu57p	
Zynq UltraScale+ MPSoC	
xczu2cg	
xczu2eg	A
xczu3cg	
xczu3eg	A, Q
xczu4cg	
xczu4eg	Q
xczu4ev	A
xczu5cg	
xczu5eg	
xczu5ev	A, Q
xczu6cg	
xczu6eg	

Table 23: **List of Supported Devices in the Current Release** *(cont'd)*

Device	Variants*
xczu7cg	
xczu7eg	A
xczu7ev	A, Q
xczu9cg	
xczu9eg	Q
xczu11eg	A, Q
xczu15eg	Q
xczu17eg	
xczu19eg	Q
Zynq UltraScale+ RFSoC	
xczu21dr	Q
xczu25dr	
xczu27dr	
xczu28dr	Q
xczu29dr	Q
xczu39dr	
xczu42dr	
xczu43dr	
xczu46dr	
xczu47dr	
xczu48dr	Q
xczu49dr	Q
xczu55dr	
xczu57dr	
xczu58dr	Q
xczu59dr	Q
xczu65dr	
xczu67dr	
Versal	
xcvc1802	
xcvc1902	
xcvp1202	ES1 only
xcvm1802	

Variants

- 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.

Documentation Navigator and Design Hubs

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

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

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

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

  Note: For more information on Documentation Navigator, see the Documentation Navigator page on the Xilinx website.

References

1. *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947)
2. *Dynamic Function eXchange Controller IP LogiCORE IP Product Guide* (PG374)
3. *Dynamic Function eXchange Decoupler IP LogiCORE IP Product Guide* (PG375)
4. *Dynamic Function eXchange Bitstream Monitor IP LogiCORE IP Product Guide* (PG376)
5. *Dynamic Function eXchange AXI Shutdown Manager IP LogiCORE IP Product Guide* (PG377)
6. *Demonstration of Soft Error Mitigation IP and Partial Reconfiguration Capability on Monolithic Devices* (XAPP1261)
7. *Bitstream Loading across the PCI Express Link in UltraScale Devices for Tandem PCIe and Partial Reconfiguration* (AR# 64761)
8. *Partial Reconfiguration of a Hardware Accelerator with Vivado Design Suite (v1.1)* (XAPP1231)

9. *Fast Partial Reconfiguration Over PCI Express Application Note (v1.0)* (XAPP1338)
10. *7 Series FPGAs Configuration User Guide* (UG470)
11. *UltraScale Architecture Configuration User Guide* (UG570)
12. *Zynq-7000 SoC Technical Reference Manual* (UG585)
13. *Partial Reconfiguration User Guide (v14.5)* (UG702) - For ISE Design Tools
14. *UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs* (UG949)
15. *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054)
16. *Virtex-7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023)
17. *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156)
18. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
19. *Vivado Design Suite User Guide: Synthesis* (UG901)
20. *Vivado Design Suite User Guide: Using Constraints* (UG903)
21. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906)
22. *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476)
23. *7 Series FPGAs GTP Transceivers User Guide* (UG482)
24. *MMCM and PLL Dynamic Reconfiguration Application Note (v1.8)* (XAPP888)
25. *UltraScale Architecture Clocking Resources User Guide* (UG572)
26. *UltraScale Architecture GTH Transceivers User Guide* (UG576)
27. *UltraScale Architecture GTY Transceivers User Guide* (UG578)
28. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
29. *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194)
30. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
31. *DMA/Bridge Subsystem for PCI Express Product Guide* (PG195)
32. *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)
33. *Zynq UltraScale+ Device Technical Reference Manual* (UG1085)
34. *Zynq UltraScale+ Device Register Reference* (UG1087)
35. *Bitstream Identification with USR_ACCESS using the Vivado Design Suite Application Note (v1.0)* (XAPP1232)
36. *Local Partial Reconfiguration Using Embedded Processing for 3D ICs* (XAPP1099)
37. *Design Advisory for Techniques on Properly Synchronizing Flip-Flops and SRLs* (AR# 44174)
38. Vivado Design Suite Documentation

Training Resources

Xilinx 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. Xilinx Partial Reconfiguration Tools and Techniques

Please Read: Important Legal Notices

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.