# Isolation Design Flow for UltraScale+ FPGAs and Zynq UltraScale+ MPSoCs

XAPP1335 (v2.1) March 9, 2021

## Summary

This application note describes how to implement security- or safety-critical designs using the Xilinx® Isolation Design Flow (IDF) with the Xilinx Vivado® Design Suite. Design applications include information assurance (single-chip cryptography), avionics, automotive, and industrial applications. This document explains how to address the following objectives:

- Implement isolated functions in a Xilinx UltraScale+™ FPGA or a Zynq® UltraScale+™ MPSoC
- Verify the isolation using the Xilinx Vivado Isolation Verifier (VIV)

You can purchase the Security Monitor IP core Product Brief developed by Xilinx to add additional security to your design. For more information, contact your local Xilinx® representative to access these documents.

This application note specifically covers UltraScale+ FPGAs and Zynq UltraScale+ MPSoCs using Vivado Design Suite 2018.3, and builds on earlier IDF concepts.

An example design is provided in the *Isolation Design Example for Zynq Ultrascale+ MPSoC Application Note* (XAPP1336). For more information, see Isolation Design Example.

## Introduction

The flexibility of programmable logic affords security-critical and safety-critical industries many advantages. However, before Isolation Design Flow (IDF) was developed, in applications, such as information assurance, government contractors and agencies could not realize the full capability of programmable logic due to isolation, reliability, and security concerns, and were therefore forced to use multichip solutions.

To address these concerns, the IDF was developed to allow independent functions to operate on a single chip. Examples of single chip applications include, but are not limited to, redundant Type-I cryptographic modules, or resident safety- and non safety-critical functions.

The successful completion of the Xilinx® Isolation Design Flow allows Xilinx to provide new technology for the information assurance (IA) industry, as well as, provide safety-critical functions in avionics, automotive, and industrial applications.

# Isolation Design Flow

Developing a safe and secure single chip solution that contains multiple isolated functions in a single FPGA is made possible through Xilinx® isolation technology. Special attributes, such as HD.ISOLATED and the features it enables, are necessary to provide controls to achieve the isolation needed to meet certifying agency requirements.

To better understand the details of the Isolation Design Flow (IDF), the designer needs to have a solid understanding of the hierarchical design flow, see *Vivado Design Suite User Guide: Hierarchical Design* (UG905). Many of the terms and processes in the partition flow are used in the IDF. Areas that are different supersede the partition design flow and are identified in this application note.

## Common Terminology

Some of the common terms used extensively in this document are ownership, function, logic, region, and fence. These terms are defined as follows:

- **Ownership (physical/logical):** The concept of physical versus logical ownership is an important concept to understand when using the IDF. This concept is described in detail in the section Concept of Ownership.
- **Function:** A collection of logic that performs a specific operation (for example, an AES encryptor).
- **Logic:** Circuits used to implement a specific function (e.g., flip-flop, look up table, and RAM).
- **Isolated Region/Pblock:** A physical construct used to restrict placement of logic to a specific region of the device.
- **Fence:** The concept of physical versus logical ownership is an important concept comprised of a set of unused tiles in which no routine or logic is present.
- **Trusted Routing:** The routes that satisfy the rules outlined in the following figures. Trusted routing is automatically enabled after the HD.ISOLATED attribute is set to *TRUE* on at least one isolated module. These routes are a subset of existing routing resources that meet the following restrictions:
  - No entry or exit point in the fence between isolated regions
  - One source and one destination region
  - Its entirety stays contained in the source/destination regions
  - It does not come within one fence tile from another unintended isolation region

  These rules act as a filter to all available routes in a given design. An example of routes that would be filtered are shown in the following figures. Example routes excluded for programmable interconnect points (PIPs), outside the intended isolation regions or proximity to unintended isolation regions, are also shown.

*Figure 1:* **Trusted Routes within an Isolated Module**

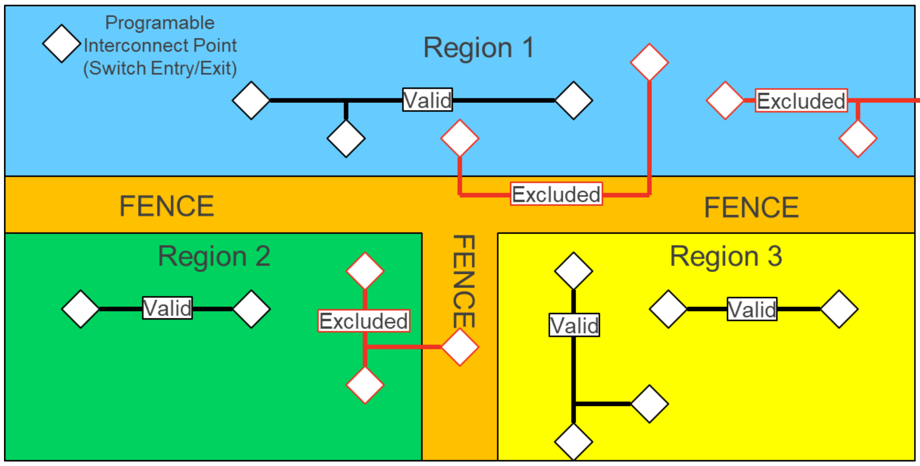
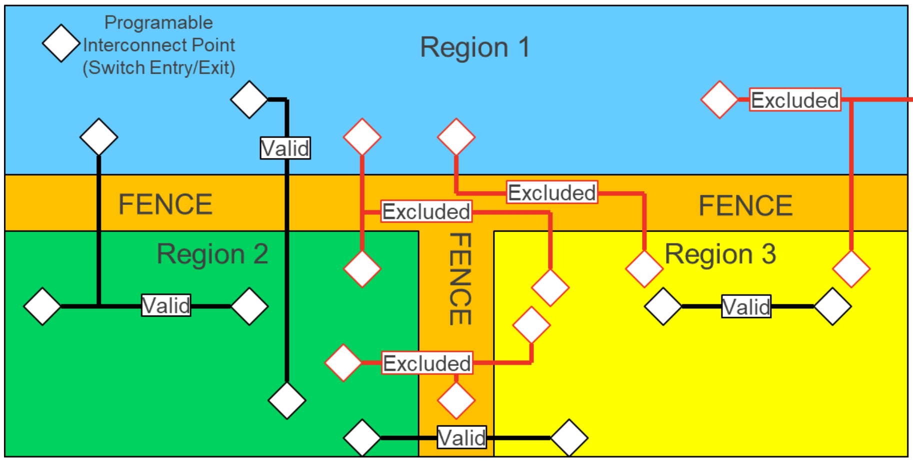


*Figure 2:* **Trusted Routes between Isolated Modules**



## Rules

A secure or safety-critical solution can be achieved while using FPGA design techniques and coding styles with only moderate modifications to the development flow. Xilinx® Isolation Design Flow (IDF) development requires the designer to consider floorplanning much earlier in the design process to ensure that proper isolation is achieved in logic, routing, and I/O buffers (IOBs). In addition to early floorplanning, the development flow is based on hierarchy. Each function that is being isolated must be at its own level of hierarchy. Although this flow requires additional steps, the hierarchical approach has certain advantages.

There are a few unique design details that must be adhered to achieve an FPGA-based IDF solution. Carefully consider all aspects of the design details explained in subsequent sections of this application note. These considerations include:

- Keep each function that is being isolated at its own level of hierarchy.
- Keep top level or non-isolated logic to a minimum. It is strongly encouraged to have only global clocks and resets at this level.

- Use a fence to separate isolated functions.
- IOBs must be instantiated inside isolated modules for proper isolation of the IOB. This can be achieved by manual user instantiation or automatically by the tools.

  ***Note*:** Automatic logical inferencing by the tools is unique to the Vivado® Design Suite.
- On-chip communication between isolated functions is achieved through the use of trusted routing (Tools automatically choose trusted routes along coincident physical borders).

## Top Level Logic

Isolated designs must keep the amount of top level logic to a minimum. In a typical Isolation Design Flow (IDF) design, the only logic at the top level should be clock logic. Any component that is not part of an isolated module in the design hierarchy is optimized to the top level. Because isolation is defined by the HD.ISOLATED attribute being set on a hierarchical module, all top logic is, by default, *not* isolated. This has the following implications:

- There are no placement constraints on top level logic other than it will not be placed in the fence.
  - Top level logic can be placed in any isolated Pblock.
- There are no routing restrictions on top level logic other than it will not violate the fence with used programmable interconnect points (PIPs).
  - Top level routes can route to, from, and through any isolated Pblock.

**IMPORTANT!** *Vivado tools automatically take care of the placement of top logic. It ensures that top level logic is not placed in the fence. If there is a large empty space in a design that is not encompassed by any Pblock, Vivado might place top logic in that empty space*

## Isolation Properties

The Vivado tool uses two specific properties to create an isolated design: HD.ISOLATED and HD.ISOLATED_EXEMPT.

The Vivado tool enables isolation of a specific function by the application of the HD.ISOLATED property on the function. This property can be set either in the XDC file or the Vivado Integrated Design Environment (IDE).

The Tcl command is:

```
set_property HD.ISOLATED true [get_cells <function_hierarchical_path>/
<function_instance_name>]
```

or

```
set_property HD.ISOLATED 1 [get_cells <function_hierarchical_path>/
<function_instance_name>]
```

You can also set the property in the Vivado GUI. See Adding Property HD.ISOLATED under Elaboration for setting the property via Vivado IDE.

By default, when **HD.ISOLATED** is enabled on a function/module, all components and routing that belong to that module are isolated. This means that unless it is communicating with another isolated module, all routing is contained within that module. This also means that all components of that module are placed in its corresponding isolated Pblock.

***Note*:** You do not explicitly add any isolation property to a Pblock. When an isolated module having the property HD.ISOLATED set is mapped to a Pblock during the floorplanning stage, that Pblock becomes an *isolated* Pblock. For further details, see Mapping the Logical Ownership to the Physical Ownership.

When the HD.ISOLATED property is set on a module, global logic instantiated within that isolated module cannot be routed globally because the module has been isolated. But, Vivado IDF allows you to instantiate global logic at any level. To support this and enable routing of the global logic, the property HD.ISOLATED_EXEMPT should be set on the global instances to override the default isolation behavior. By setting this property on the global logic instance, Vivado routes and treats it as global logic instead of isolated logic.

The format to exempt a global logic instance from isolation is:

```
set_property HD.ISOLATED_EXEMPT true [get_cells
<function_hierarchical_path>/
<function_instance_name>]
```

or

```
set_property HD.ISOLATED_EXEMPT 1 [get_cells <function_hierarchical_path>/
<function_instance_name>]
```

See Guidelines for Controlling Global Clocking Logic for more information regarding the HD.ISOLATED_EXEMPT property, including an example of setting HD.ISOLATED_EXEMPT on all global logic in the design.

***Note*:** Only global logic can be at the top of an isolated design, but it can also be instantiated in an isolated hierarchy if the HD.ISOLATED_EXEMPT property is used. By setting this property on global logic, the Vivado tool effectively treats them as top level logic.

**IMPORTANT!** *Every physical component in the FPGA including IOBs needed by an isolated module, must be owned by its corresponding isolated Pblock. The designer needs to create Pblocks in such a way that it has all the resources needed by the corresponding isolated module. This implies that IOBs needed by an isolated module must also be included in the corresponding isolated Pblock. For details on floorplanning, see Floorplanning.*

## Isolation Modules

Each function/module requiring isolation must be in its own hierarchical block. The reason for this is that each module that has the isolation attribute set has the attribute applied to that module and its entire logical hierarchy (sub-modules under the parent module on which HD.ISOLATED property is set).

***Note*:** Nested isolated modules are not supported.

IDF has many additional constraint rules as opposed to traditional Partial Reconfiguration (PR), or hierarchical design flows from *Vivado Design Suite User Guide: Hierarchical Design* (UG905). It is necessary to define one key attribute for each module to be isolated to invoke the IDF rules to create isolated designs. As discussed in the above section, this attribute is HD.ISOLATED and needs to be set for each module that needs isolation for the backend Vivado implementation tools to use the IDF rules. This allows effective Floorplanning of an isolated design in IDF, and protects redundant functions from undesired optimization.

## Communication between Isolated Modules

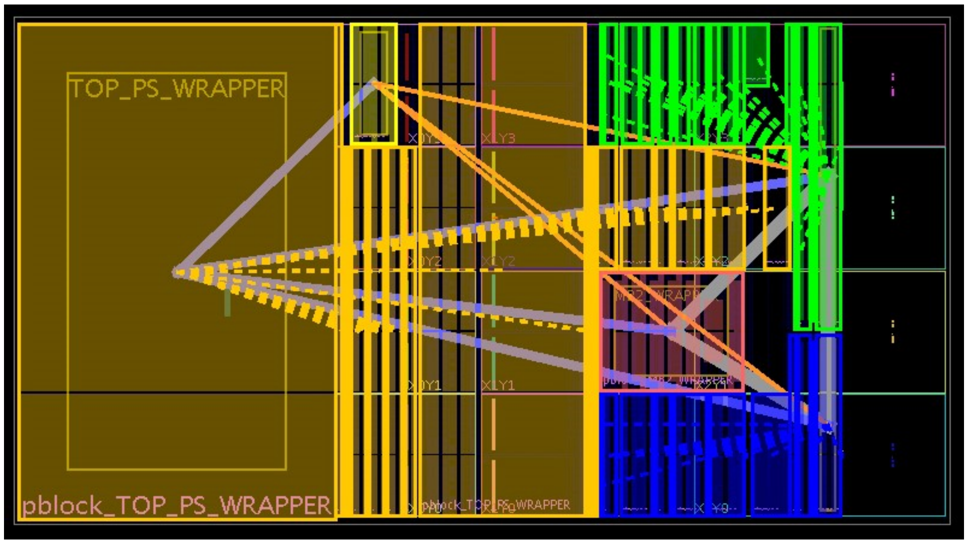
When communication between isolated modules is required, there are two possible solutions:

- Trusted Routing is the preferred method. For details on the user rules, when using trusted routing, refer to Trusted Routing Rules in the Design Guidance section. Trusted routing is automatically taken care of by Vivado tools hence this is the preferred method.
- Signals can be taken off-chip from one isolated Pblock, routed through the PCB, and then brought back on-chip, in a separate isolated Pblock. Although possible, this method is not preferred because it is complex and impractical.

## Reference Design

For clarity, an example single-chip 2 channel functional safety design is used throughout this application note to describe the design details and tool flow. The following figure shows the floorplan for the lab design as implemented in an XCZU5CG-SFVC784-1-e device. It consists of five isolated regions. In addition, this design has been implemented with Vivado® Design Suite 2018.3, verified by the Vivado Isolation Verifier (VIV) 2.0, and provided to the designer as a reference.

*Figure 3:* **IDF Design Floorplan**



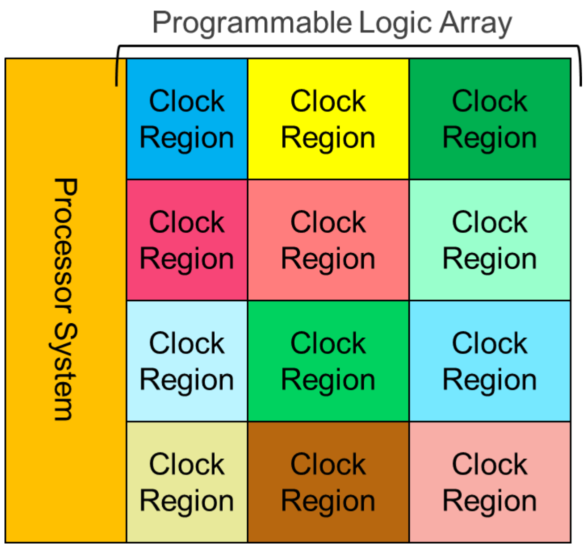
Access the isolation design for this application note from *Isolation Design Example for Zynq Ultrascale+ MPSoC Application Note* (XAPP1336).

# Architecture Overview

## MPSoC Chip Layout

Xilinx® MPSoC devices are made up of several sections connected to each other. The processor system (PS) is a monolithic block which is connected to the programmable logic (PL) through a set of interconnect tiles. The PL is made up of columns of tiles organized into clock regions as shown in the following figure for the ZU5 architecture.

*Figure 4:* **ZU5 Layout**



## Clock Regions and Columns

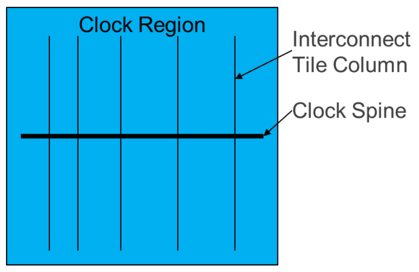
Each MPSoC is built using similar architecture. The ZU5 programmable logic (PL) uses four rows of three clock regions. Each clock region contains multiple columns of PL and dedicated block. Each column is composed of a single type of tiles. The number of tiles per column depends on the height of the tile type. A column of configurable logic block (CLB) tiles has 60 tiles. The PCIe® tile is an example of the tallest tile used. A column of PCIe has one tile.

The column width changes based on the tile type for that tile. It is important to understand that the column width for clock regions vertically arranged is constant. In other words, a column of CLB tiles in the blue clock region extends to the same column in the red clock region below it all the way to the last clock region going vertically down so the same width of the column is maintained from clock region to clock region. See Figure 32: FPGA Layout for details.

## Clock Spines and Interconnect Columns

The clock spine for each clock region is located in the center of each clock region as shown in the following figure. The clocks are distributed vertically using the columns of interconnect tiles which then drive local logic devices in other tiles. Refer to *UltraScale Architecture Clocking Resources User Guide* (UG572) for more details on the UltraScale clock architecture.

*Figure 5:* **Clock Spine Location**



Interconnect tiles are used to route connections between logic tiles. These interconnect tiles are the equivalent to route channels in ASIC designs, and also provide a convenient method to isolate logic regions.

## Pblocks and Programmable Units

A key difference between the UltraScale+ architecture and previous architectures supported by IDF is the use of back-to-back (B2B) interconnect tiles. Where previously one logic tile (CLE, BRAM, DSP, etc.) was associated with one or more dedicated interconnect tiles (INT column), UltraScale+ interconnect tiles service two distinct logic tiles (one on the left and one on the right of the INT column). This group of tiles along with their shared interconnect is called a Programmable Unit.

Thus, a Programmable Unit (PU) is a set of logical tiles such as CLEs, BRAMs, DSPs along with their shared interconnect tiles. For example, one Block RAM, five CLEs, and five interconnect tiles shared between the Block RAM and CLEs constitute one PU. For the sake of convenience, this can be called BRAM PU. Another instance of a PU can be a CLE and CLM tiles on either side of a shared Interconnect (INT) tile. Example of PUs have been listed under Programmable Unit Sizes.

You can think of a Pblock as a region in the FPGA which is made up of Programmable Units (PUs), while creating isolated designs. For IDF, a PU is the smallest logic building block that can be assigned to a Pblock. A Pblock is created by adding multiples PUs (as required) into it.

You must include the whole PU inside the Pblock while reserving resources during floorplanning. From the IDF perspective, having a part of the PU outside of the Pblock, for example, having a CLE tile of BRAM PU not included inside the Pblock renders the whole PU (in this case the whole BRAM PU - BRAM, the interconnects, and the CLE tiles) unusable for the Vivado tools during implementation stage. This is deliberate in IDF design as all the tiles within a PU share interconnect (INT) tiles, and isolation tools cannot place logic in a tile unless the complete PU of that tile is in an isolated Pblock. To grab all the resources of a PU inside a Pblock, the snapping mode property of the Pblock must be set to FINE_GRAINED. See Derived Range and Snapping Mode for detailed explanation.

## Programmable Unit Sizes

It is important to include enough required programmable unit (PU) resources in a Pblock to support the assigned module. This is verified by reviewing the resources table when the Pblock is drawn. A Pblock can be of any shape; the shape is defined by the combination of PUs that are assigned to it.

The following table lists out the minimum fence size in terms of PU for the user tiles. For more details on Fence, refer to Isolation Fence.

*Table 1:* **PU Unit Sizes**

| Tile | User Tile Description | Fence Size |
|---|---|---|
| SLICEL (CLBL) | Configurable Logic Block. The key logic unit of an FPGA. | Vertical: 1 PU<br>Horizontal: 1 PU |
| SLICEM (CLBM) | Configurable Logic Block. The key logic unit of an FPGA. | Vertical: 1 PU<br>Horizontal: 1 PU |
| DSP | Digital Signal Processor. A programmable math function (DSP tile is two DSP48E1 slices; analysis was done on DSP48E1 slice; Vivado® tools only allow selection of a DSP tile). | Vertical: 1 PU<br>Horizontal: 1 PU |
| BRAM | Block RAM. User-accessible high speed RAM (BRAM tile is RAMB36 which is two RAMB18 blocks; Vivado tools only allow selection of a BRAM tile). | Vertical: 1 PU<br>Horizontal: 1 PU |
| URAM | High density block RAM. User-accessible high speed RAM | Vertical: 1 PU<br>Horizontal: 1 PU |
| HDIO | General purpose I/O block | Vertical: 1/2 PU (12 IO)<br>Horizontal: 1/2 PU (12 IO) |
| HPIO | High Performance I/O block | Vertical: 1 PU (52 IO)<br>Horizontal: 1 PU (52 IO) |
| GTX/GTY QUAD | High speed transceiver. The GTX or GTY Quad tile is made up of four channels. PR Units in different Pblocks are allowed to abut as long as one channel along the abutment is not used. | Vertical: 1 PU<br>Horizontal: 1 PU PU GTX/Y Channel |
| PCI-E | PCI Express® Endpoint Block | Vertical: 1 PU<br>Horizontal: 1 PU |
| SYSMON | System Monitor. Contains analog-to-digital converters (ACDs). | Vertical: 1 PU<br>Horizontal: 1 PU |
| CMAC | Centralized Media Access Control Block | Vertical: 1 PU<br>Horizontal: 1 PU |
| INTERLAKEN | High Speed Chip to Chip pack transfer port | Vertical: 1 PU<br>Horizontal: 1 PU |

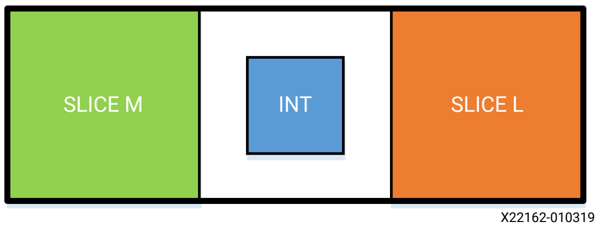
**Notes:**

1. If a fence is needed between two adjacent HPIO banks, it should be an unused HPIO PU between them. This renders all of the I/Os of the fence HPIO bank to be unusable.

## Programmable Unit Size Examples

The smallest programmable unit (PU) is the CLB PU unit shown in the following figure. The CLB PU unit is 1/60th of a clock region high and two columns wide.

*Figure 6:* **CLB PU**



The next smallest programmable units are the DSP PU UNIT and the BRAM PU UNIT shown in the following figures.

*Figure 7:* **DSP PU**

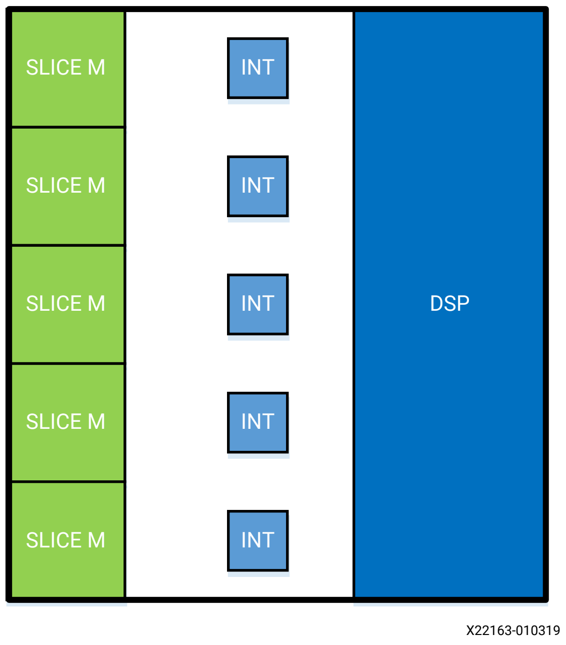
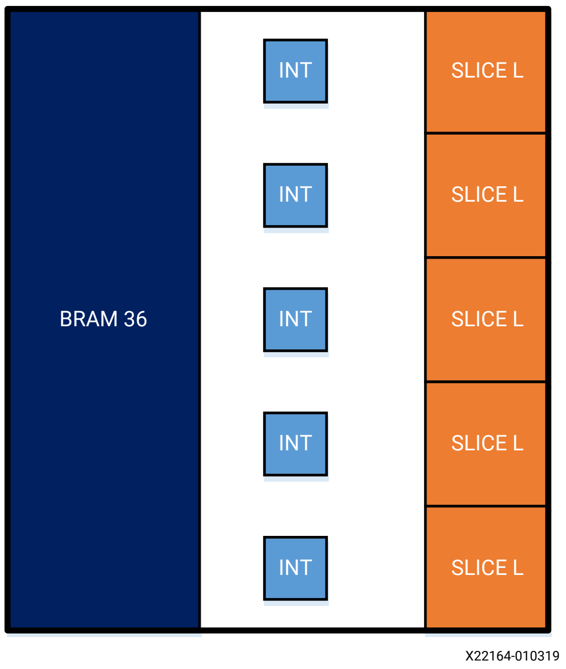
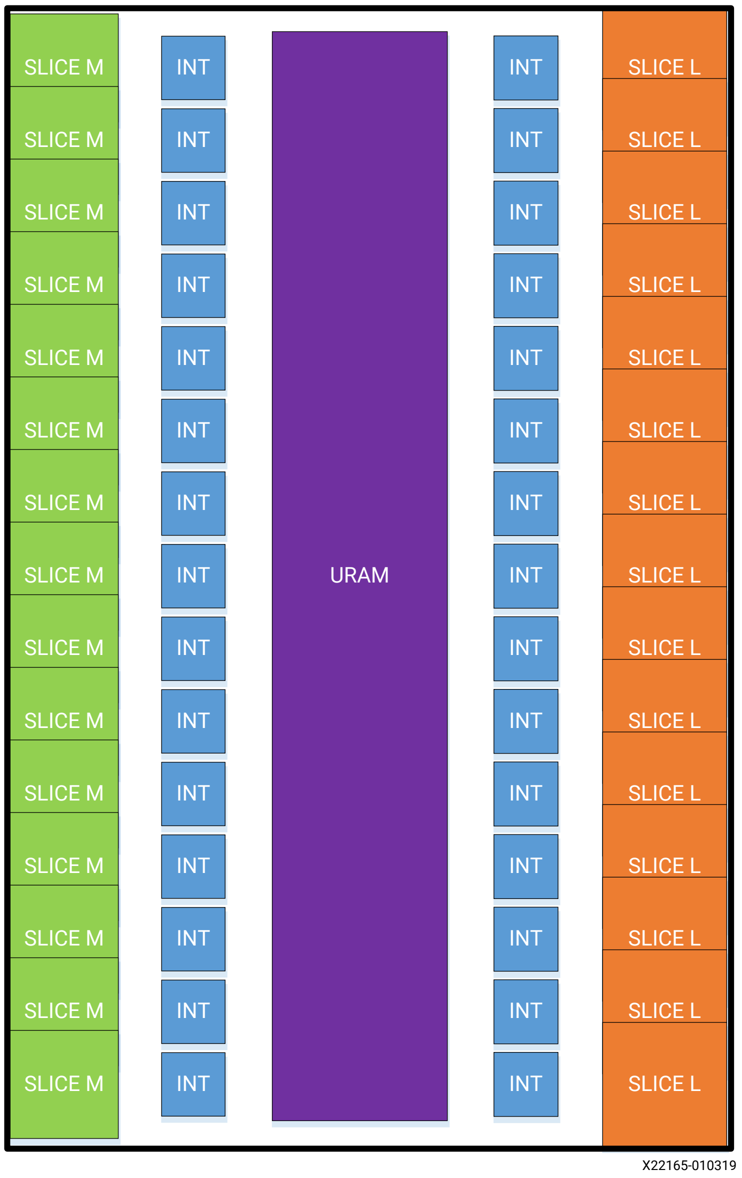


*Figure 8:* **BRAM PU**



The URAM PU is shown in the following figure.

*Figure 9:* **URAM PU**



The tallest PUs are:

- **52 HPIO PU:** One clock region high and two columns wide
- **4 Channel GTH PU:** One clock region high and three columns wide
- **SYSMON PU:** One clock region high and also three columns wide

Other one clock high PU units include PCIe® and CMAC.

*Figure 10:* **HPIO PU**

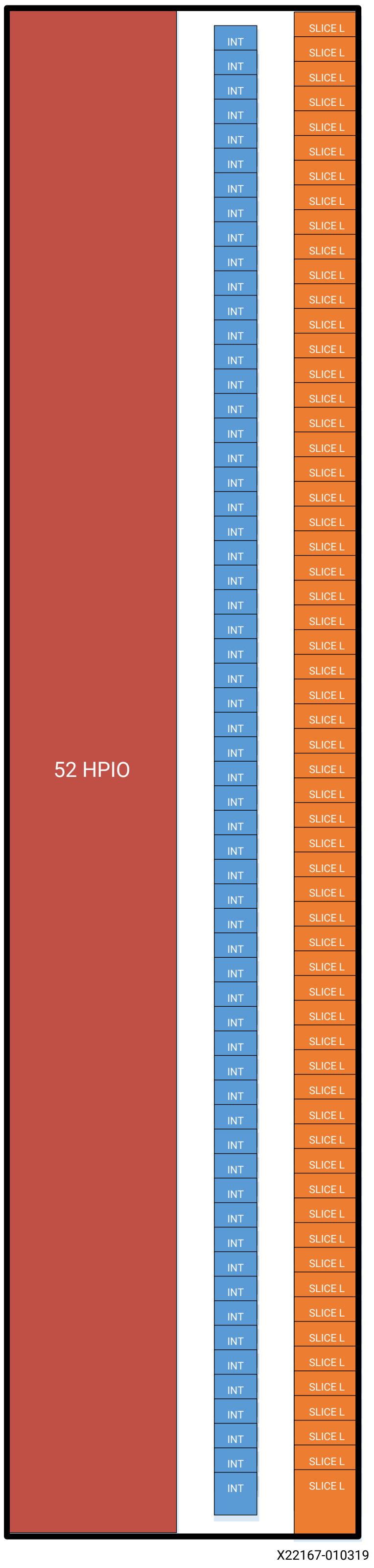
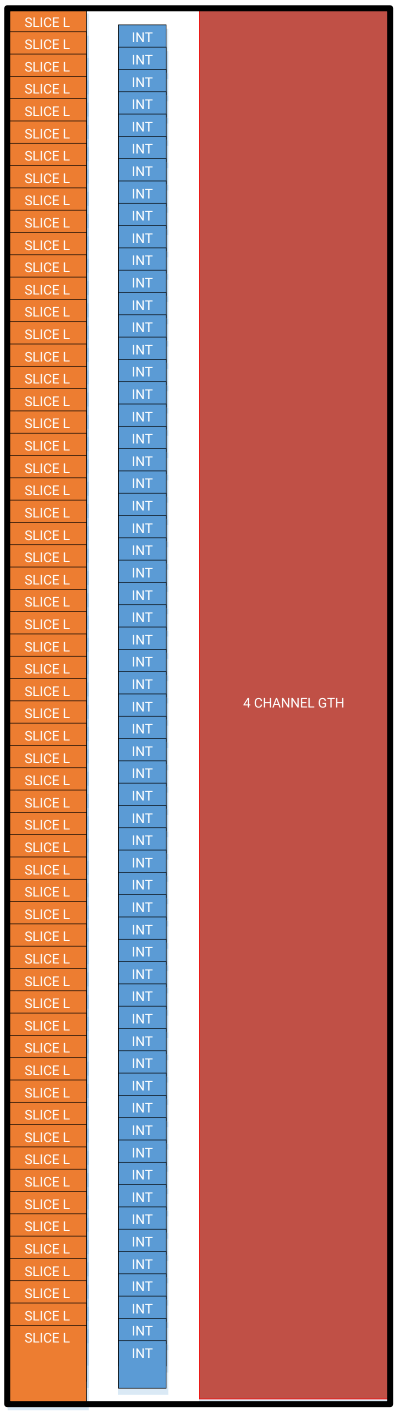


*Figure 11:* **GTH QUAD PU**



X22168-010319

*Figure 12:* **SYSMON PU**

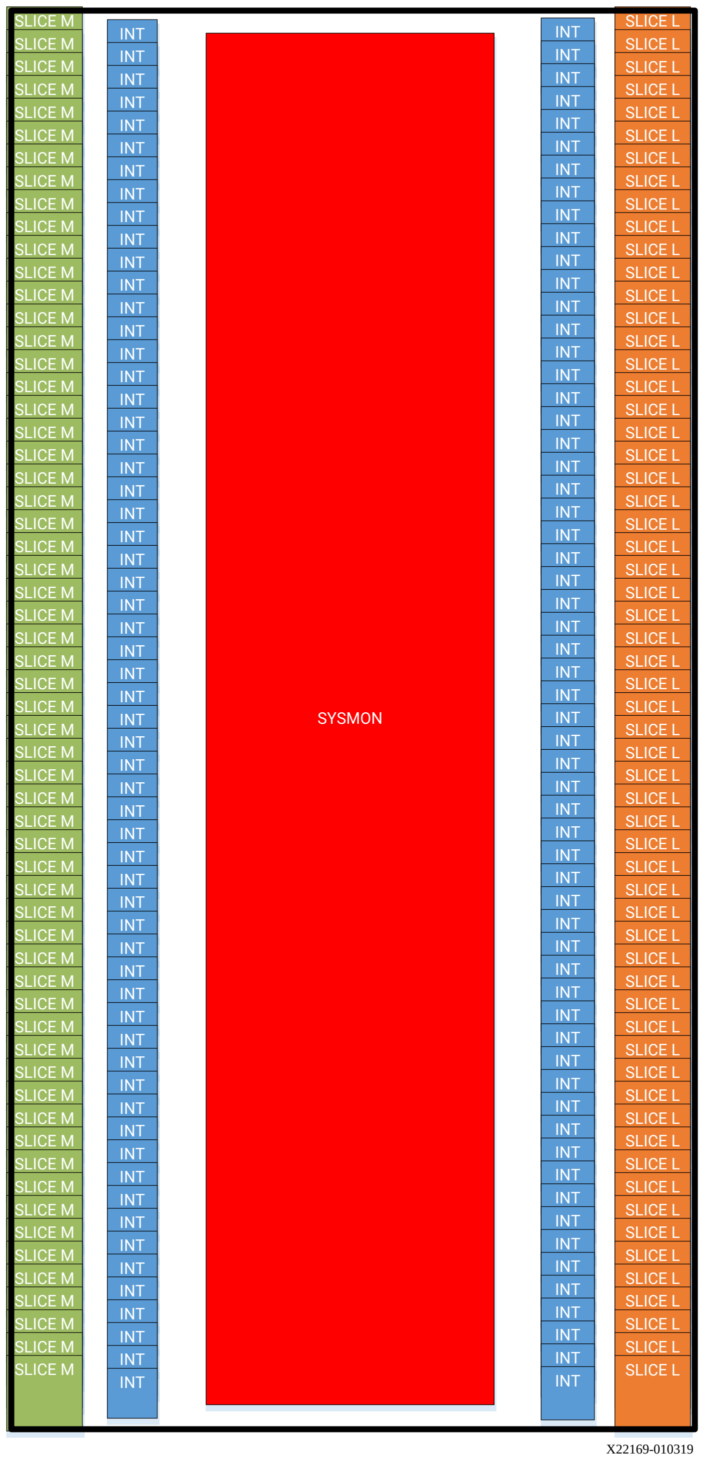
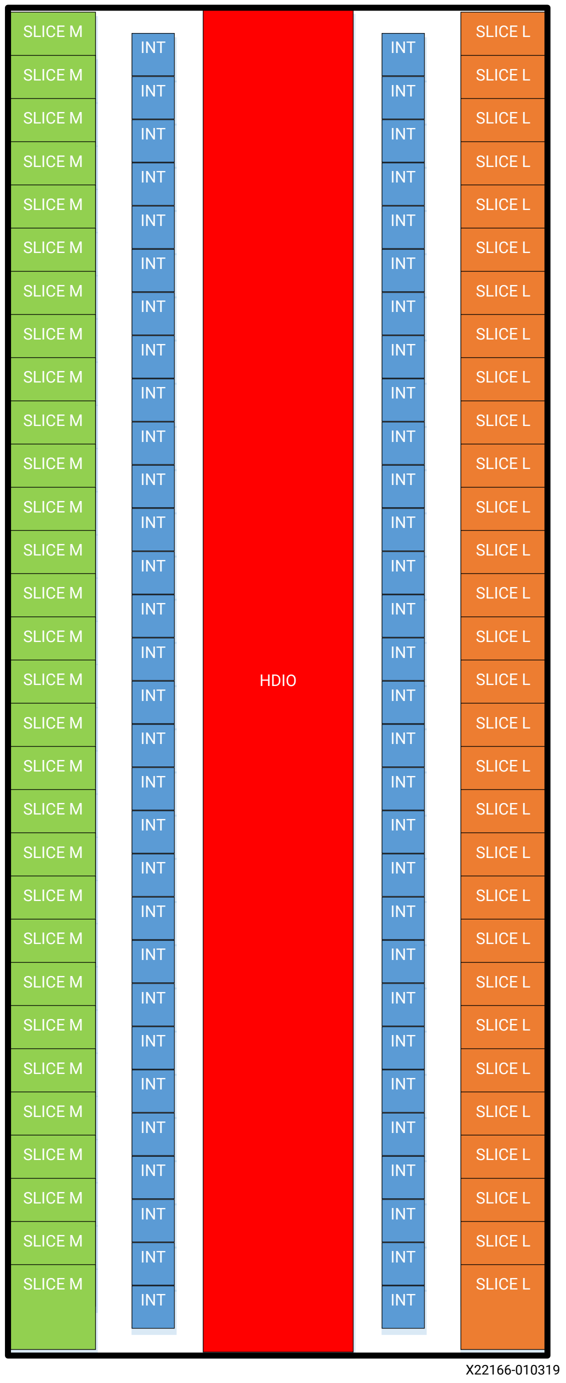


*Figure 13:* **HDIO PU**



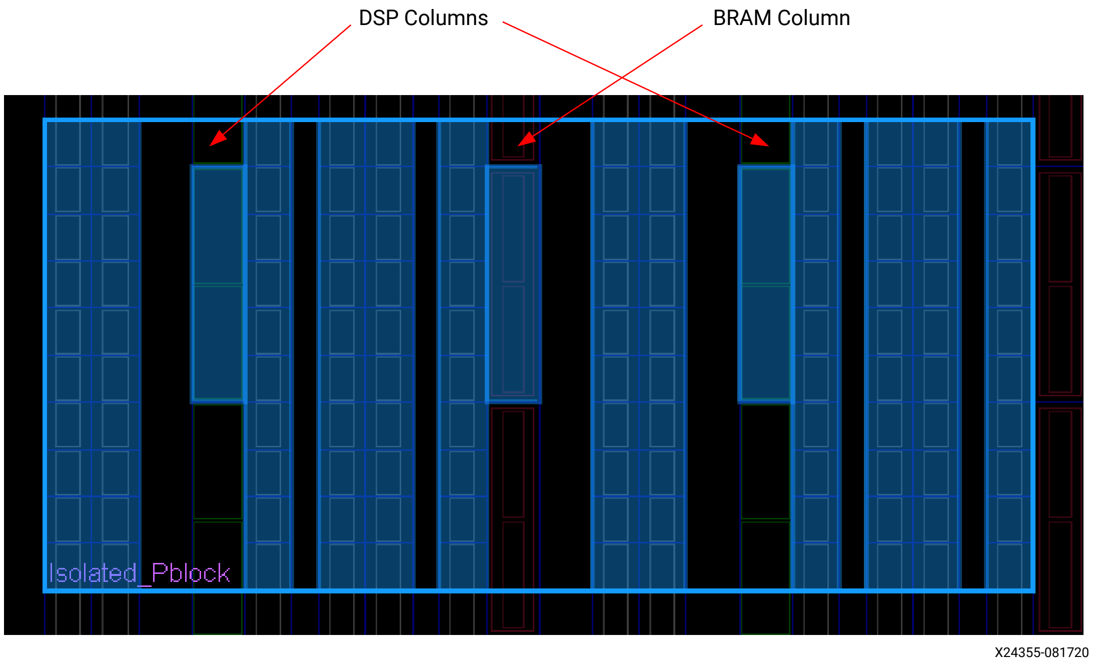
Send Feedback

## Derived Range and Snapping Mode

Derived range is the boundary of a Pblock after considering all the tiles of the Programable Units (PU) which are included in that specific Pblock. When the snapping mode property of a Pblock is set to OFF, then both derived range (calculated by internal tools), and XDC range (provided by the user), compute the same Pblock boundary. But when the Snapping Mode property of a Pblock is set to FINE_GRAINED, the derived range might differ from the user-specified XDC range. This is because, if Snapping Mode property of a Pblock is set to FINE_GRAINED and if some of the tiles of a PU are left outside in the Pblock's XDC range, then all the tiles in that PU are completely excluded from the Pblock boundary in the derived range computation. Understanding the concept of FINE_GRAINED SNAPPING_MODE is important as IDF only works with this specific property.

In the following figure, the Pblock's snapping mode property is set to OFF, and all the tiles (irrespective of PU inclusion) are part of the Pblock. The only exceptions are the DSP and the BRAM tiles since entirety of these tiles is not inside the Pblock boundary. This can be verified by looking at the blue shaded portion in the Pblock. The DSP and BRAM tiles are not part of the Pblock because they are not highlighted; the whole tile needs to be included in the PBlock, and not part of it. It can be observed in the following figure only one of the sites of the DSP and BRAM is included, and hence the tiles that are not highlighted got excluded in the derived range computation.

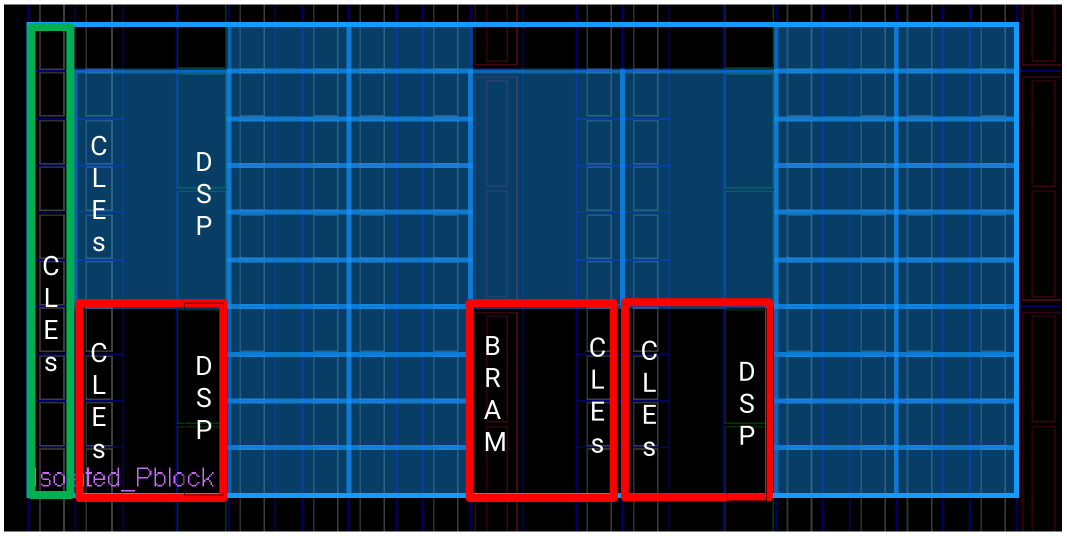
*Figure 14:* **A Pblock with SNAPPING_MODE Set to OFF**



The following figure displays the same Pblock as the above figure but with the Pblock's SNAPPING_MODE set to FINE_GRAINED. It can be observed that some of the tiles have been excluded (seen by the black regions inside blue shading) from the Pblock which were previously included when snapping mode was OFF. In the following figure, those specific tiles have been excluded whose PU is partially inside the Pblock. For example, in the portion of Pblock marked with the red boxes, the CLEs got excluded because the DSP tiles and BRAM tile of the PU they are part of, are not included in the Pblock boundary. Similarly, the CLEs surrounded by green box are not included in the Pblock as they are part of a BRAM PU (the BRAM is towards the left and is not visible in the screen capture).

***Note***: In the following figure, the DSPs and BRAM inside the red box are not included as part of the Pblock in the first place because for a tile to be part of a Pblock (even with snapping mode OFF), the entirety of the tile must be inside the Pblock boundary. In this case, only one of the sites of the DSP and BRAM is inside of the Pblock boundary.

*Figure 15:* **Pblock Shading with SNAPPING_MODE Set to FINE_GRAINED**



**IMPORTANT!** *IDF works on derived range so ensure that Pblock's SNAPPING_MODE property is set to FINE_GRAINED. This property can be set either in the Vivado GUI under Pblock property listed as SNAPPING_MODE, or via Tcl command as:* `set_property SNAPPING_MODE FINE_GRAINED [get_pblocks <Pblock_name>]`

***Note***: The default value of SNAPPING_MODE property of isolated Pblocks is FINE_GRAINED.

## Mapping the Logical Ownership to the Physical Ownership

One of the most difficult concepts in IDF is the relationship between the *logical ownership* and the *physical ownership* of a user design.

*Logical Ownership* refers to the actual Hardware Description Language (HDL) of the design. *Physical Ownership* refers to where that design resides in an FPGA, i.e., the actual placement of the HDL logics in the FPGA fabric. The mapping from the logical ownership to the physical ownership happens when a Pblock is created and a logical instance is assigned to it. This binding of a logical instance or a portion of hierarchy, to a physical location in the device, is the foundation of the Isolation Design Flow. This is achieved with the following commands:

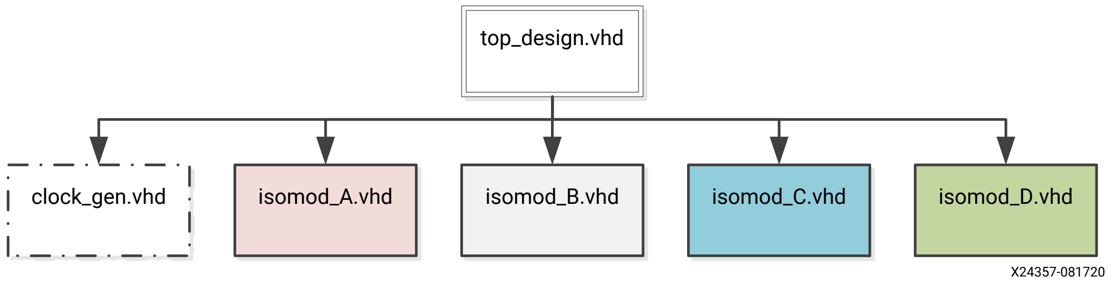
```
create_pblock <Pblock_name>

add_cells_to_pblock [get_pblocks <Pblock_name>] [get_cells -quiet [list */
<isolated_module_name>]]
```

After the Pblock has been created and logic has been assigned to it, the Pblock must be defined (floorplanned) to add the necessary resources as discussed in Floorplanning. Note that any FPGA physical component - PU or routing resource that is not ranged / included in the Pblock definition cannot be used by the logic (module) assigned to that.

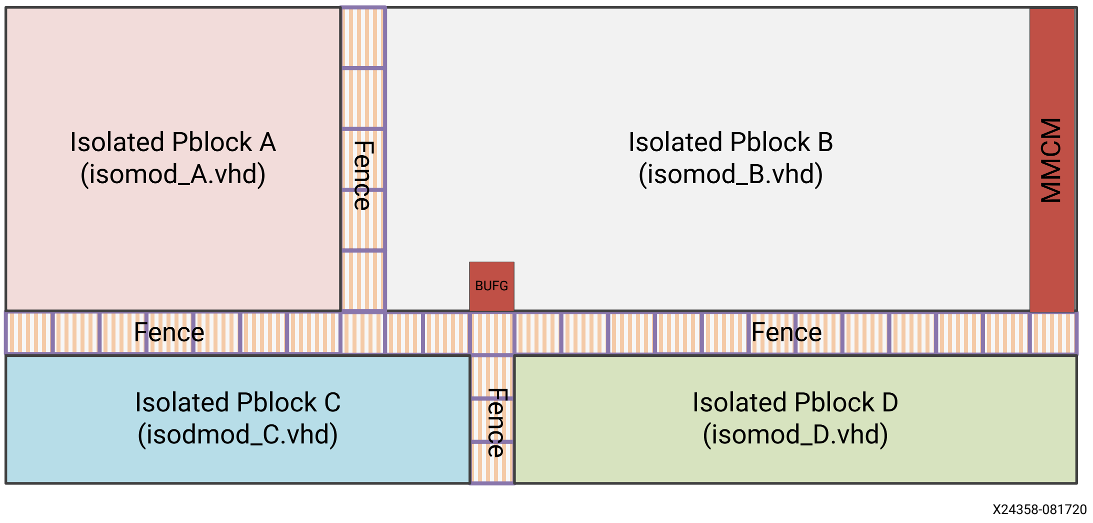
The following figure shows the logical ownership of four isolated modules – A, B, C, and D and the global clock.

*Figure 16:* **Logical View of the Four Isolated Modules**



The following figure shows the physical ownership for the four isolated modules of the previous figure. The logical modules have been assigned Pblocks in the FPGA using Vivado tools with a valid fence between them.

*Figure 17:* **Physical View of the Four Isolated Modules**



**IMPORTANT!** *Routing resources (interconnect tiles) associated with Pblock ranged components are added automatically by the tools. Interconnect tiles are not specifically defined in the Pblock definition (i.e., the XDC constraint file). When you add a PU to a Pblock, the associated corresponding interconnect tile resources are automatically added to the Pblock by Vivado tools.*

## Off-Chip Communication – Input / Output Buffer Control

If an isolated module has inputs or outputs that must come from or go off-chip, these signals must have their IOBs inferred or instantiated inside an isolated module. While this is different from standard FPGA design practice, it is required in order to have control over the routing of the signals from the IOB to the function. If the IOB is not part of the isolated logic, there is no control on how the signal is routed from the IOB to that isolated logic.

***Note***: If an isolated module's IOB is not instantiated inside the isolated module, Vivado will automatically attempt to move the IOB into the isolated module's netlist. A more detailed description of Vivado tools insertion of IOBs is described in Hints and Guidelines under Automatic Movement of IOB into Isolated Hierarchy.

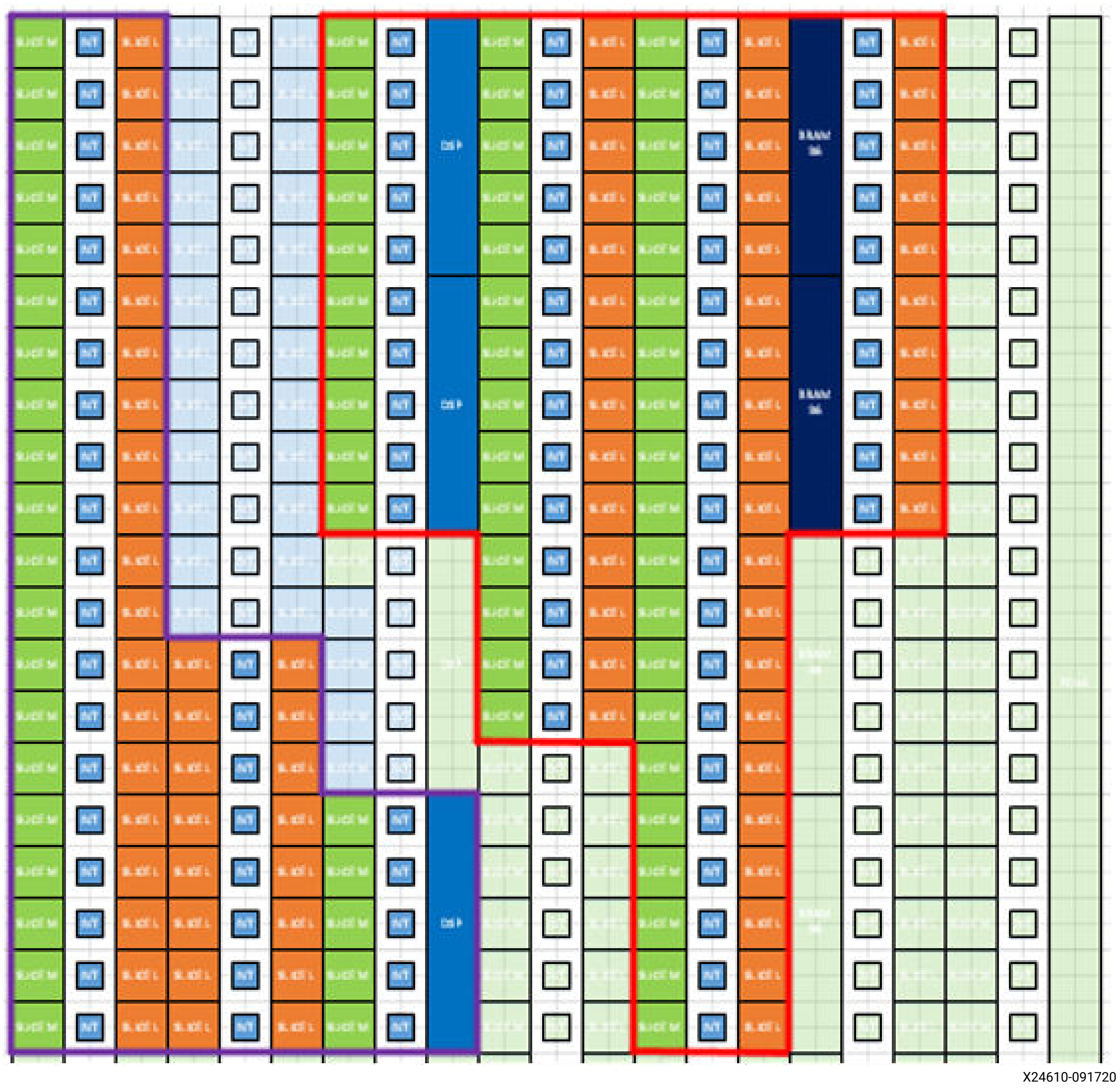
**IMPORTANT!** *The general rule is that every physical component in the FPGA, including the IOBs, that is needed by an isolated module must be owned by its corresponding isolated Pblock. It is the responsibility of the designer to create Pblocks in such a way that it has all the resources needed by the corresponding isolated module. This implies that the IOBs needed by an isolated module must also be included in the corresponding isolated Pblock. For details on floorplanning, see the Floorplanning section.*

## Isolation Fence

Table 1 section lists out the fence sizes in terms of PUs for the various user tiles.

To achieve isolation within a single device, the concept of a fence is introduced. The fence is just a set of unused PUs in which no logic is present. The results of the isolation analysis performed by Xilinx shows that if specific PUs is placed between isolated Pblocks, it guarantees that no single point of failure exists that can compromise the isolation between the two Pblocks. An example of fence placement between two isolated modules is shown in the following Figure 18. In this figure, two Pblocks are defined (each outlined in purple and red). The translucent PUs are part of the fence. The fence, like a Pblock, can be of any shape. The Vivado tools do not place any logic in the fence PUs.

*Figure 18:* **Pblocks and Fences**



As mentioned in Pblocks and Programmable Units, UltraScale+ architecture consists of Back-To-Back (B2B) interconnects. Due to the presence of these B2B interconnects, a fence encompasses both logical tiles on either side of the interconnect and their associated interconnects i.e., the whole Programmable Unit (PU). While this might lead to loss of usable resources like IOBs or BRAM and DPS etc., not having the whole PU as part of isolation fence does not guarantee isolation (protection against single point of failure).

**IMPORTANT!** *The fence location is not directly specified by the designer, rather it is created indirectly by applying the appropriate physical constraints of the isolated module's PU resources to the corresponding module's Pblocks. The PU resources that are not included in any Pblock, forms the Isolated Fence. It is up to the designer to ensure that this fence provides isolation to the isolated Pblocks in accordance with the fence rules. For guidance on which PUs can be used as a fence and the fence size, see the Table 1.*

***Note***: Except for GT programmable units, any Pblock that abuts to another Pblock creates a fence violation. When GT PUs abut, one I/O channel from either GT PU cannot be used at the abutment edge. This unused I/O channel becomes part of the fence. Other than the PUs listed in Table 1, no other PU or any tile resource can be used as a fence.

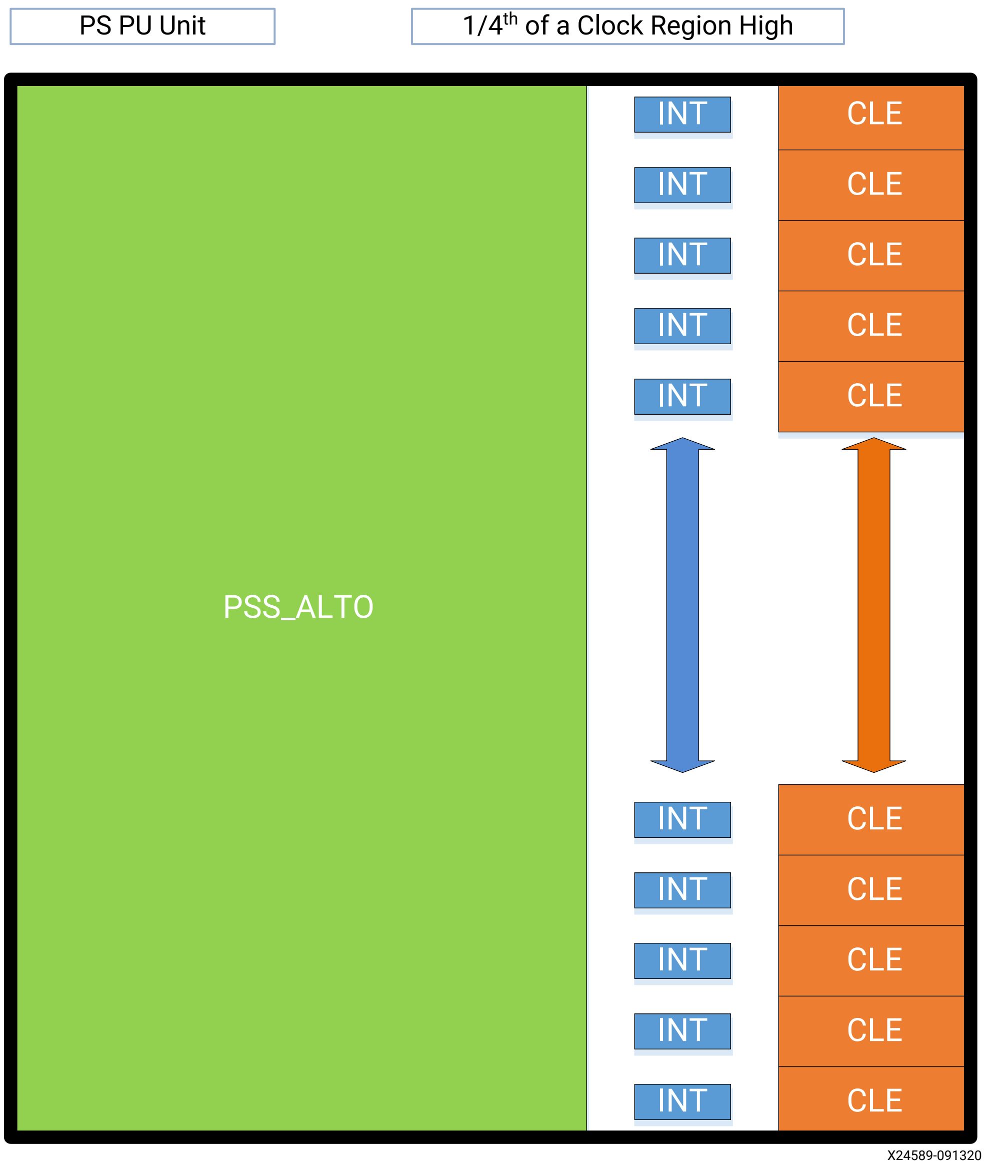
The minimum fence width is determined by a detailed schematic analysis. This analysis provides for a minimum of two faults (though in most cases it is much more) before the separation between two isolated modules is violated. While it is not mandatory to keep the size of the fence to this minimum as listed in Table 1, you can incur a stiff penalty for using larger fences. Since IDF rules prevent routing touchdowns in the fence (i.e., having a PIP in the fence), having a fence size larger than the minimum does not provide any more protection against faults but rather might lead to difficulty in meeting routing and timing constraints.

There are Trusted Routing rules that act as a filter to the router when implementing isolated designs (see Trusted Routing Rules). Any route that violates the Trusted Routing rules gets removed from the list of valid resources visible to the router. This is what creates the trusted routes – i.e., all possible routes that adheres to the rules such as no touchdown in fence, no PIPs in fence, and so on. If the fence in the design is too wide, all routes are removed and then the design will not be able to route (if communication across that boundary is intended). Hence, it is highly recommended to keep the minimum size fence width to maximize the routing resources available between isolated Pblocks. Additionally, if two isolated modules communicate with each other, then their corresponding Pblocks must be adjacent, i.e., side by side, along with a valid fence between them. It can give rise to issues if this is not the case.

## Fence Around Processor Subsystem for Zynq UltraScale+ Devices

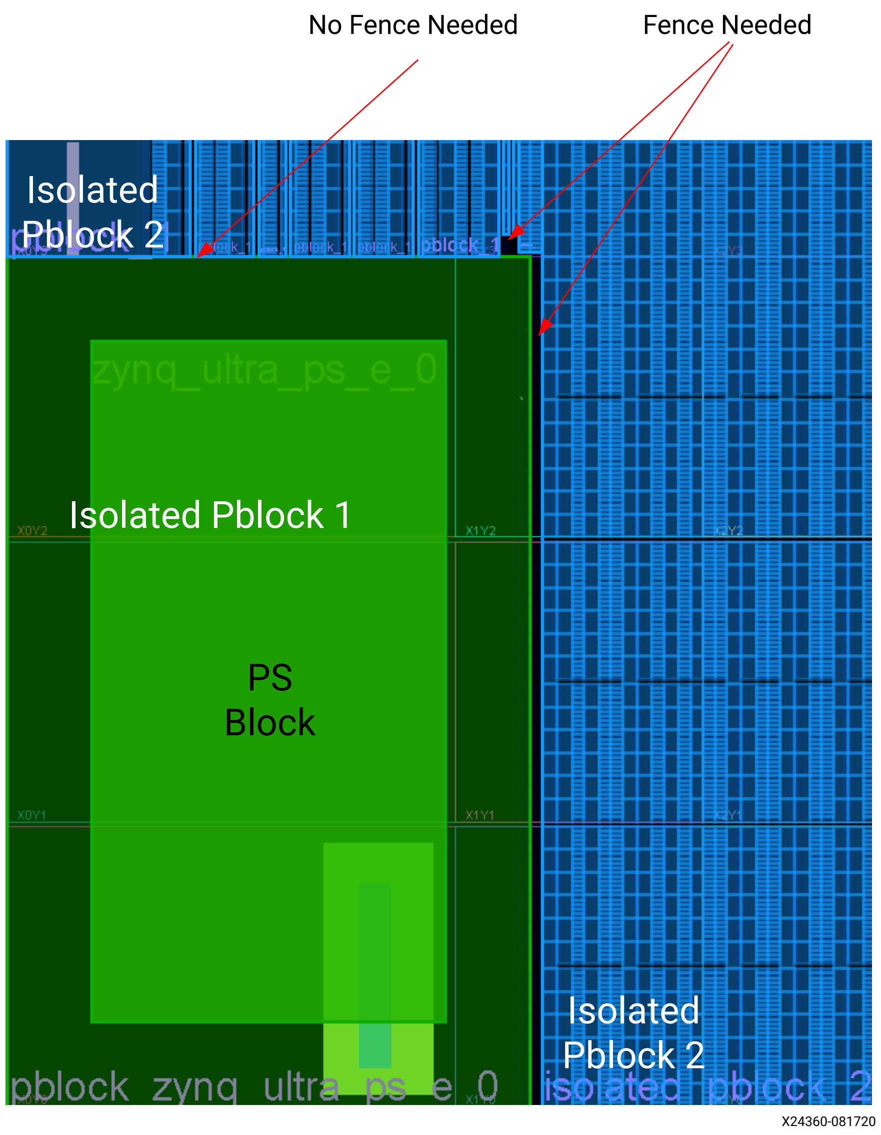
In Zynq UltraScale+ devices, there is a PSS Programmable Unit. This PU consists of the PSS_ALTO Tile, Interconnects, and a column of CLEs as high as the PSS tile. The following figure shows the PSS PU:

*Figure 19:* **PU of PSS Alto Tile for Zynq UltraScale+ Device**



The PSS PU cannot be a fence because of its fixed placement on the device. Additionally, it is not applicable as a fence because it does not sit between different logic tiles. But to isolate the PS logic from the rest, it does need a fence towards the right side of the PU – after the CLEs tiles of the PU and around the top right corner. The following figure shows how the fence can be drawn around the PSS PU.

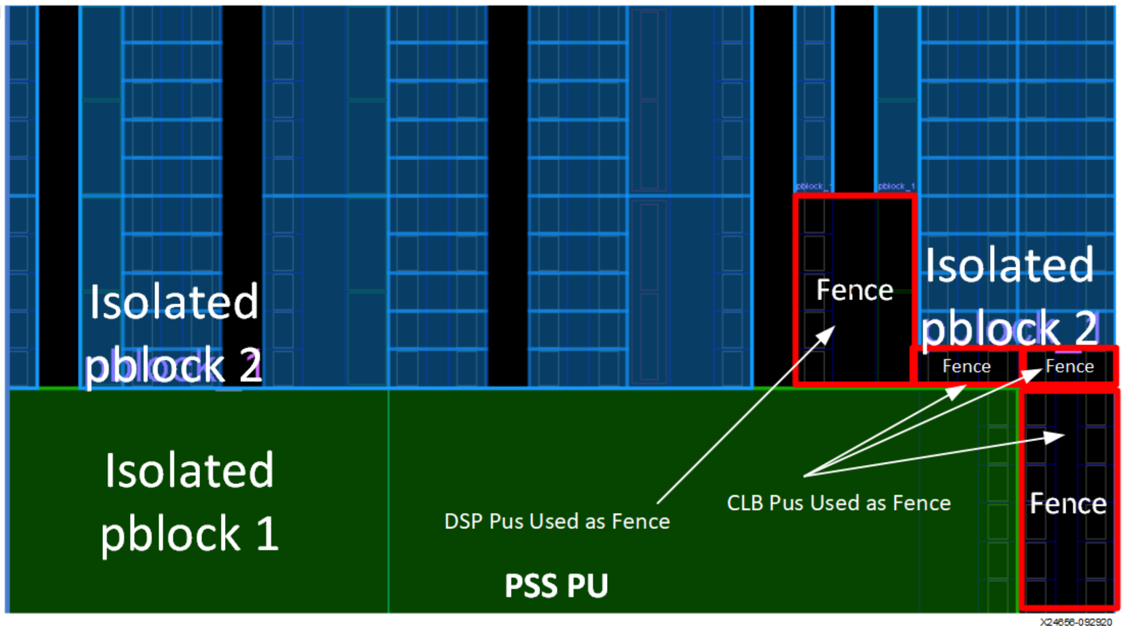
*Figure 20:* **How to Draw a Fence Around PSS PU**



As shown in these following figures, no fence is needed towards the top of the PU. This is because there are no interconnects or connection for the top part of the PU with the rest of FPGA resources.

The following figure shows a zoomed in part for the regions where the fence is needed.

*Figure 21:* **Zoomed in Image for Top-Right Part of the PSS PU**



So, to isolate the PSS isolated Pblock (shown here in green color) with another adjacent isolated Pblock (shown here in blue color), you will need one column PU fence towards the right of the PSS PU, from top to bottom. In Figure 20: PU of PSS Alto tile for Zynq UltraScale+ Device, CLB PUs (the next available PU after the PSS PU) has been used as fence starting from the top of the PSS PU to the bottom (entire right side), of the PSS PU. The previous figure shows a zoomed in part of the same. Additionally, a fence comprising of 3 PUs is needed towards the top right to account for adjacency violation between the PSS Pblock (green) and the adjacent Pblock (blue). One PU fence is directly on top of the CLEs of the PSS PU. The other two PU fences are on the left and right sides. In the previous figure, the middle fence is made of a CLB PU, the left of it is a DSP PU, and the right of it is a CLB PU.

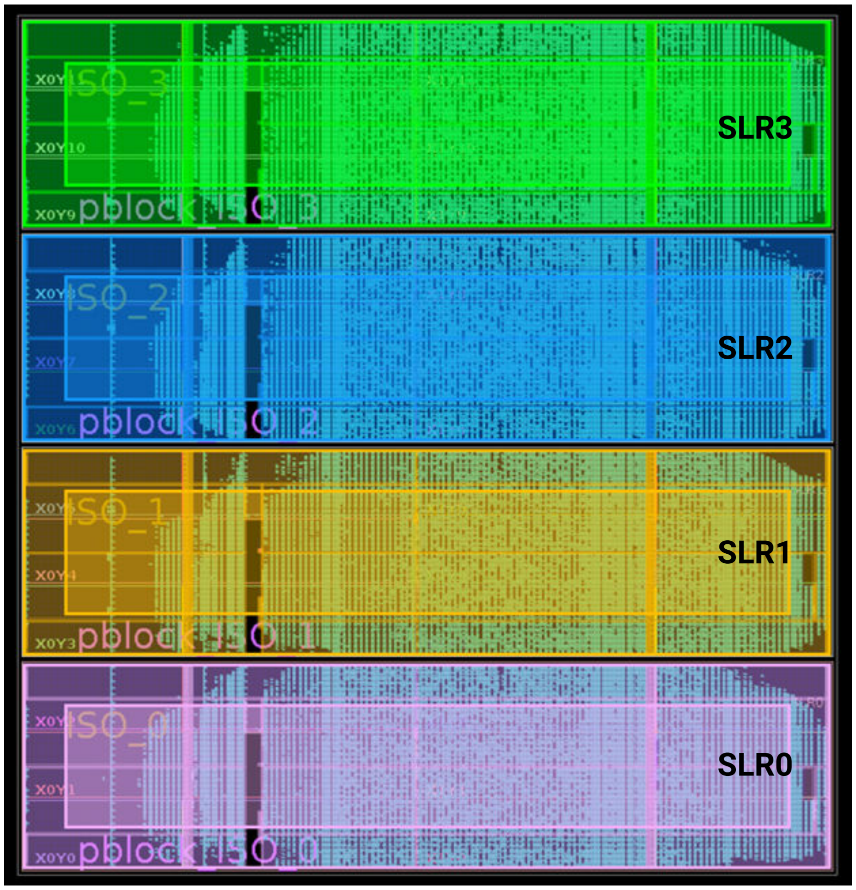
***Note***: The above screen captures are just for reference. The actual PUs for the fences might differ from part to part, e.g., instead of the DSP PU, the top left fence can be a CLB PU. However, you need to ensure that there is fence towards the whole right columns, as well as top right corner of the PSS PU (if used as part of isolation design). In smaller Zynq UltraScale+ devices there might not be any user tiles on top of the PSS PU, the whole left side of the part is the PSS PU. In that case, you may need a fence only towards the right column of the PSS PU (and not the top corner).

## Fence for SSIT Devices

In case of SSIT Devices, the SLR boundaries serve as natural isolation boundary/fence. No additional fence is required between SLRs if a SLR region contains only one isolated Pblock. If every SLR region consists of a single isolated Pblock, no additional fence is needed. However, if a SLR region contains multiple isolated Pblocks, the appropriate fence is needed between them

The following figure shows (four) 4 isolated Pblocks, highlighted in (four) 4 different colors, drawn in each SLR regions of an SSIT device. No explicit fence is needed between the Pblocks as the SLR boundaries serve as isolation fence.

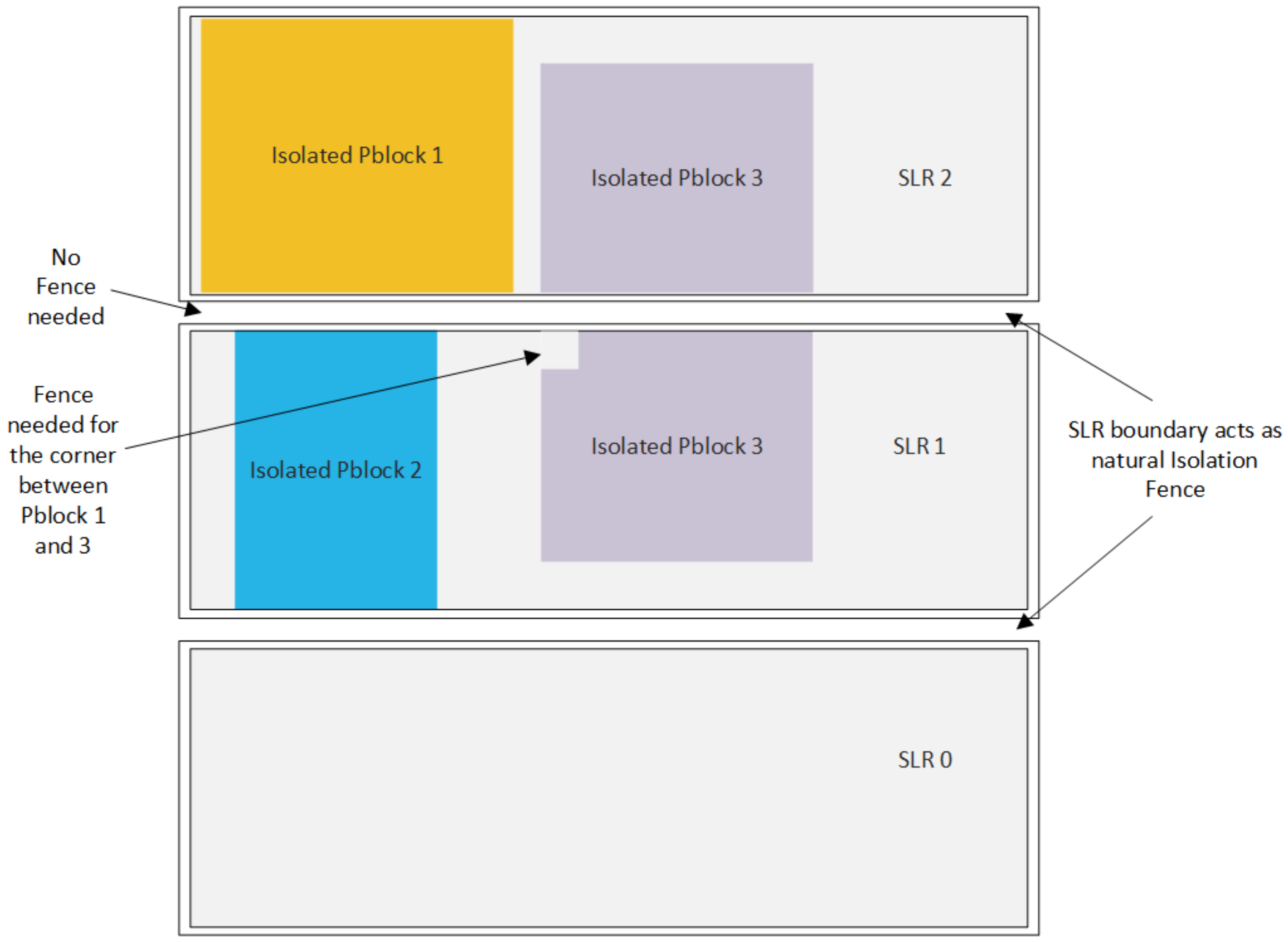
*Figure 22:* **Vivado Screen Capture of an IDF Design Implemented on an SSIT Device**



X24362-081720

In the following figure, no fence is needed between Isolated Pblock 1 and Isolated Pblock 2 since they have the SLR boundary of region 2 and 1 as natural isolation fence. But a fence will still be required between Isolated Pblock 3 and Isolated Pblock 1, and between Isolated Pblock 3 and Isolated Pblock 2. Note that the light grey color in the diagram is the FPGA logic. A fence is required between the bottom right corner of the Isolated Pblock 1, and the top left corner of the second half of Isolated Pblock 3, to account for diagonal adjacency violation.

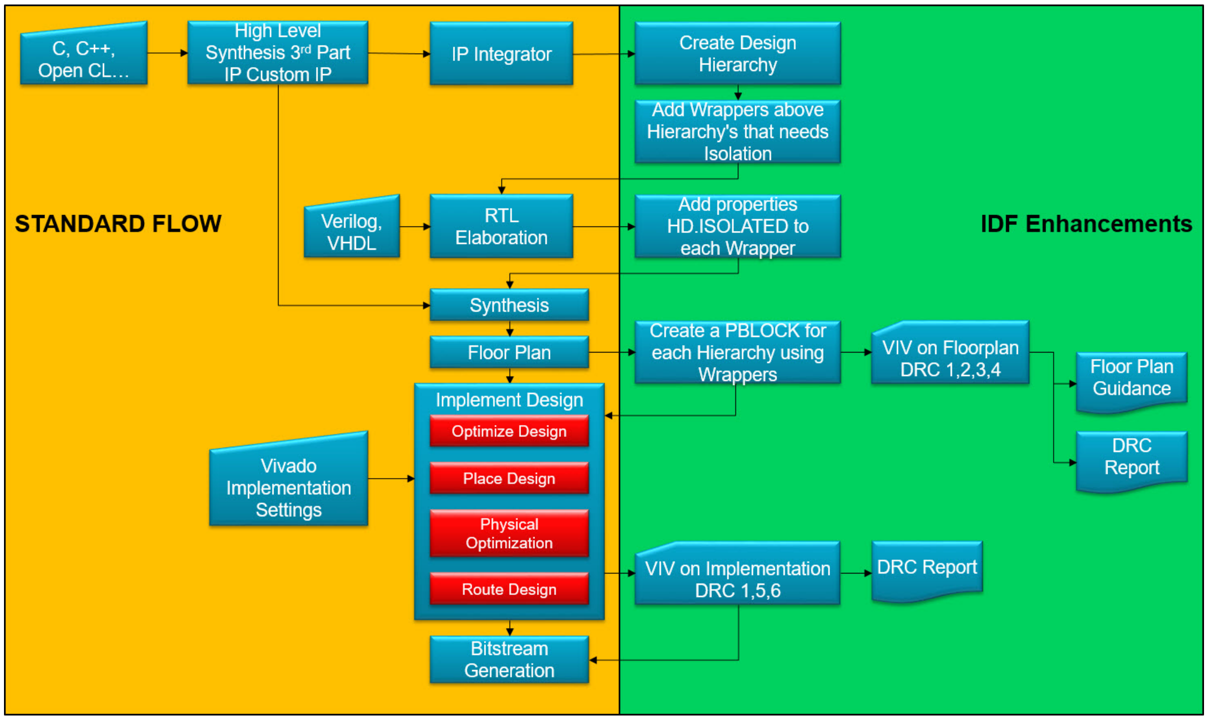
*Figure 23:* **Isolated Design with 3 Modules on an SSIT Device**



## Design Process

The Isolation Design Flow (IDF) software methodology allows for logical and physical separation of hierarchical designs. As mentioned before, there are many applications requiring IDF such as government-grade cryptographic systems, safety-critical applications, and high-availability systems. The following figure shows the recommended process for a design using the Isolation Design Flow (IDF). These steps are key to achieve single-fault hardware isolation required in Information Assurance and Functional Safety designs. IDF is not that different from a traditional design flow. The following figure shows a flow diagram of both flows - Standard and IDF side by side for the Vivado tools.

*Figure 24:* **Design Flow Chart for IDF**


C, C++, Open CL…
High Level Synthesis 3rd Part IP Custom IP
IP Integrator
Create Design Hierarchy
Add Wrappers above Hierarchy's that needs Isolation
STANDARD FLOW
Verilog, VHDL
RTL Elaboration
Add properties HD.ISOLATED to each Wrapper
IDF Enhancements
Synthesis
Floor Plan
Create a PBLOCK for each Hierarchy using Wrappers
VIV on Floorplan DRC 1,2,3,4
Floor Plan Guidance
DRC Report
Implement Design
Optimize Design
Place Design
Physical Optimization
Route Design
Vivado Implementation Settings
VIV on Implementation DRC 1,5,6
DRC Report
Bitstream Generation

## Design Capture

The is where the design may be captured in the Vivado® Design Suite. As shown in Figure 24, instead of using the IP integrator, Verilog, or VHDL *code* can also be used for creating the isolated modules. After the design is captured, a functional hierarchy is established, based on your isolation strategy with respect to how data is flowing in your design. These create the logical boundaries which will define the physical isolation as the design flow is implemented. This step is critical as all other activities are based on this hierarchy.

The isolated modules require an additional wrapper after the hierarchy gets established. These modules are captured via an IP integrator. This additional wrapper is required to enable port splitting that might not be allowed by tool generated items, such as the hierarchy created by the triple modular redundancy (TMR) management tool. See *Isolation Design Example for Zynq Ultrascale+ MPSoC Application Note* (XAPP1336) for further details.

*Figure 25:* **IP Integrator Tool Generated Design Example**

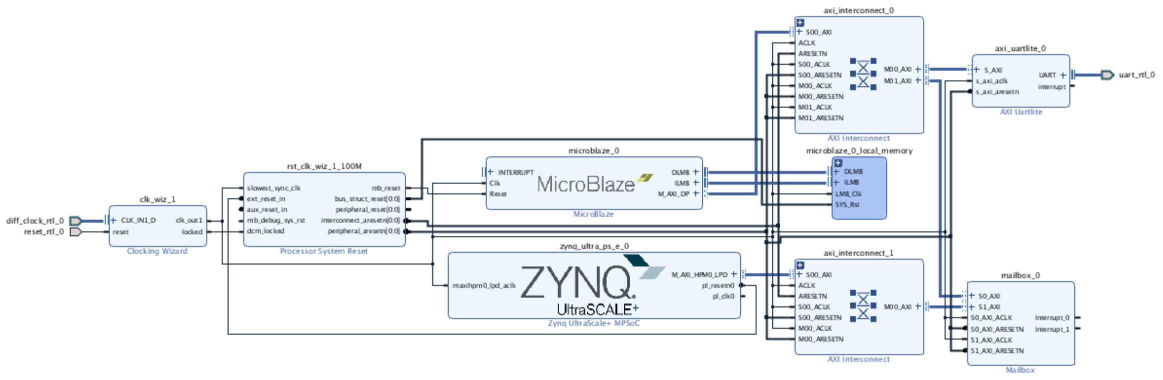


*Figure 26:* **MicroBlaze™ Hierarchy for TMR Manager**

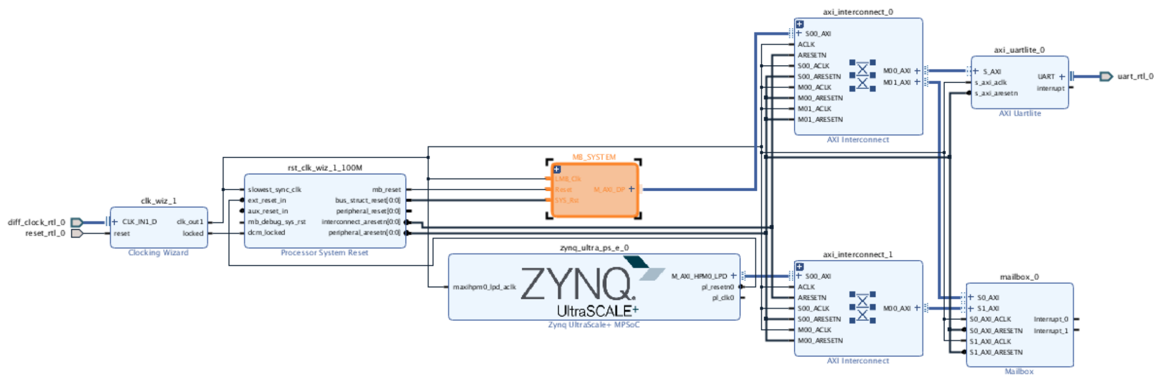


*Figure 27:* **MB_SYSTEM Hierarchy**

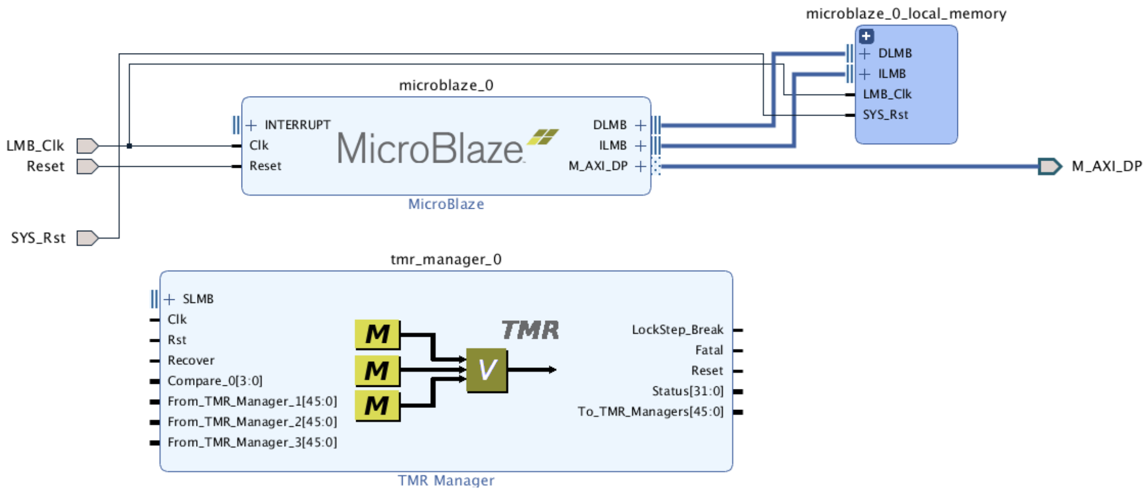


*Figure 28:* **MicroBlaze TMR Completed Automation**

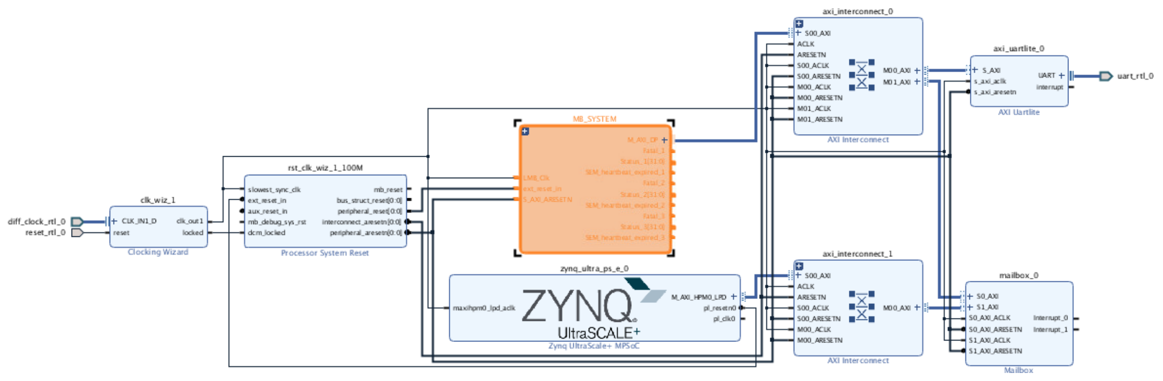


*Figure 29:* **Updated MB_SYSTEM Hiearchy with TMR Implemented**

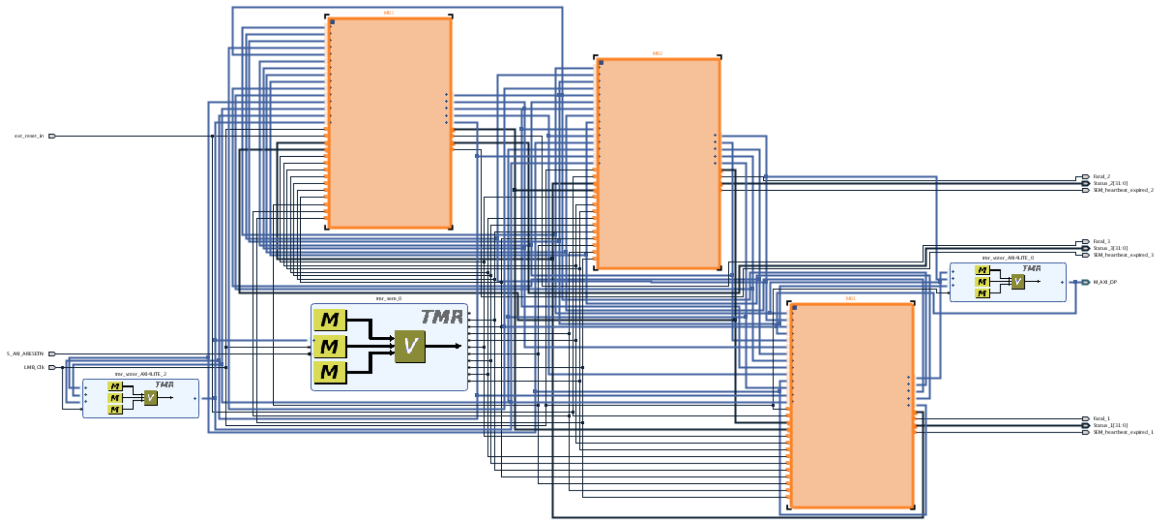


*Figure 30:* **MB_SYSTEM Hierarchy Removed to Flatten Design for Wrappers**

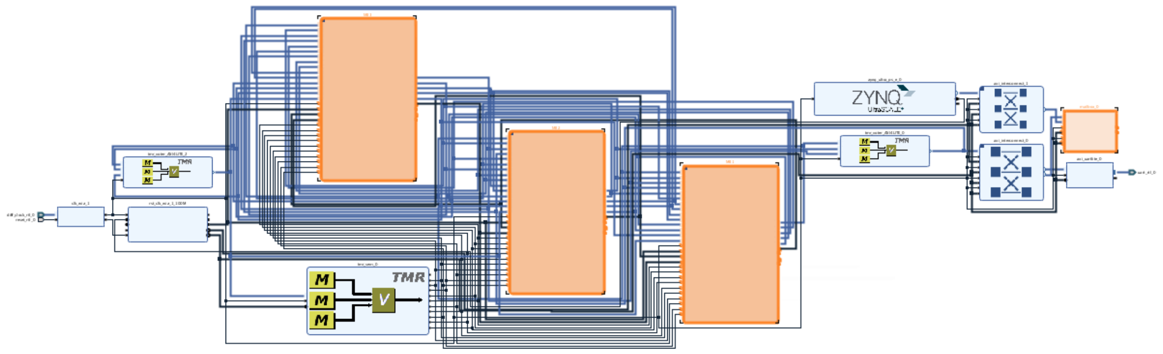
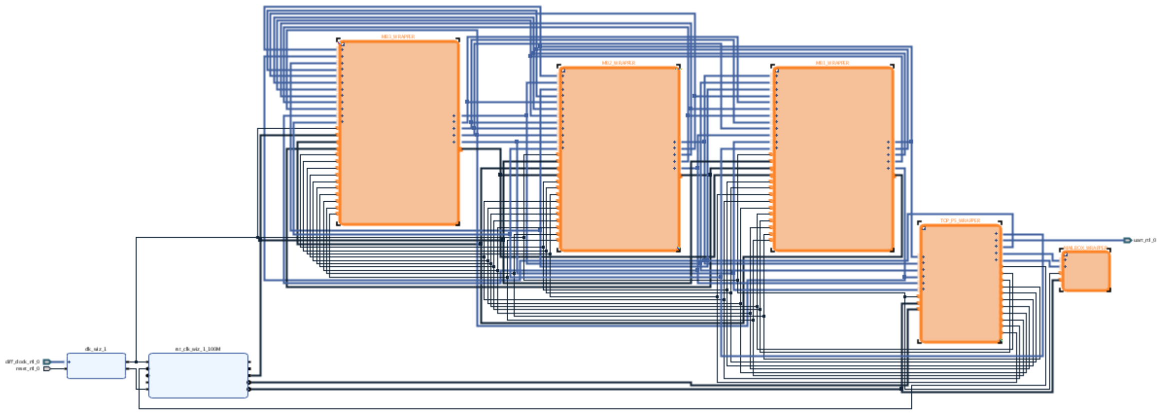


*Figure 31:* **5 User-Created Hierarchy (Wrappers)**



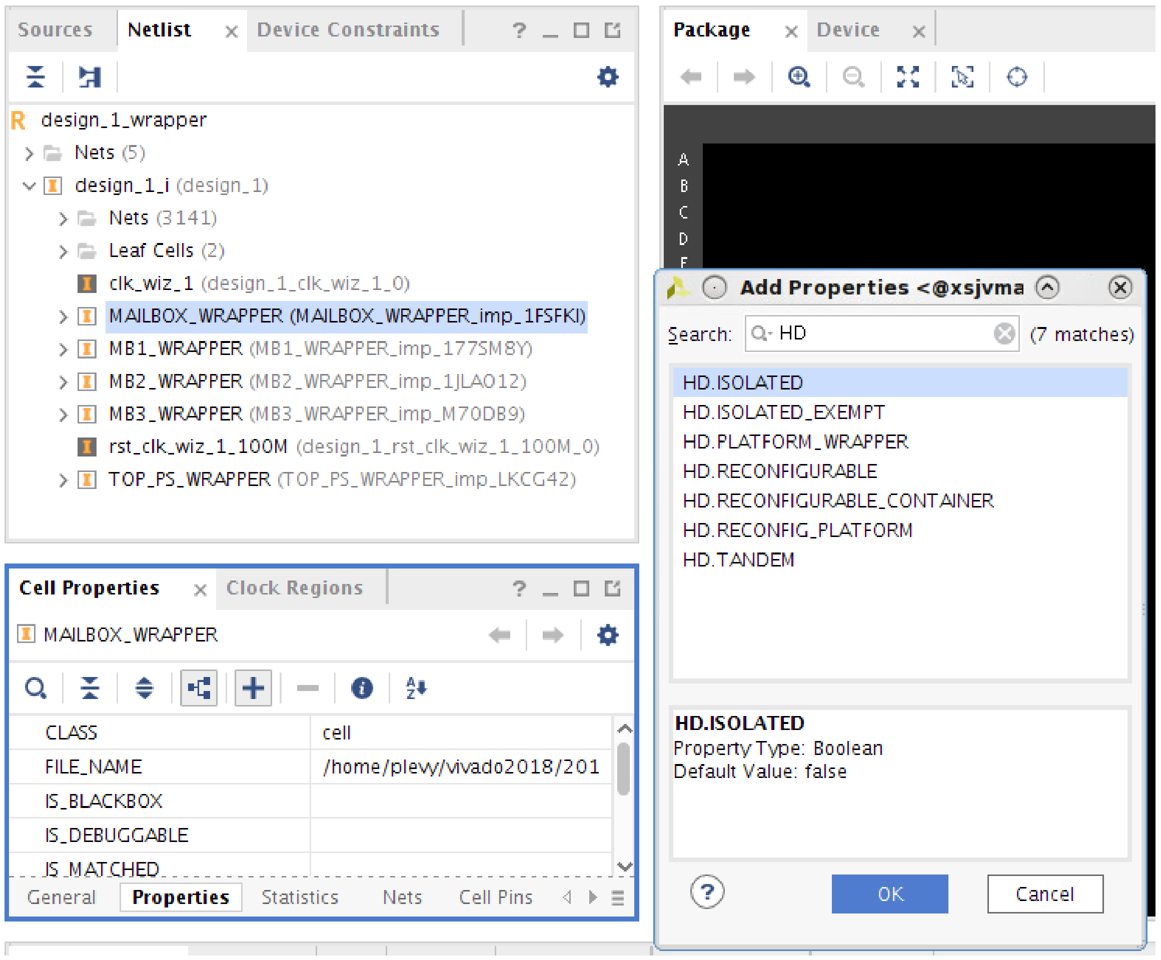
## Simulation

This step verifies functional expectation of your design. There are no additional actions required to comply with the isolation design flow (IDF).

## Elaboration

In Isolation Design Flow (IDF), the HD.ISOLATED property is added and enabled for each user-created wrappers that need to be isolated from each other. All other modules that do not require isolation do not get this property added. Such modules will get optimized to top level and have no place and route restrictions beyond keeping the fence intact. At this point in the design flow, this property is added on each wrapper, so synthesis knows which hierarchy is isolated, preventing optimization across isolation boundaries.

*Figure 32:* **Adding Property HD.ISOLATED**



design_1_wrapper
Nets (5)
design_1_i (design_1)
Nets (3141)
Leaf Cells (2)
clk_wiz_1 (design_1_clk_wiz_1_0)
MAILBOX_WRAPPER (MAILBOX_WRAPPER_imp_1FSFKI)
MB1_WRAPPER (MB1_WRAPPER_imp_177SM8Y)
MB2_WRAPPER (MB2_WRAPPER_imp_1JLA012)
MB3_WRAPPER (MB3_WRAPPER_imp_M70DB9)
rst_clk_wiz_1_100M (design_1_rst_clk_wiz_1_100M_0)
TOP_PS_WRAPPER (TOP_PS_WRAPPER_imp_LKCG42)

MAILBOX_WRAPPER

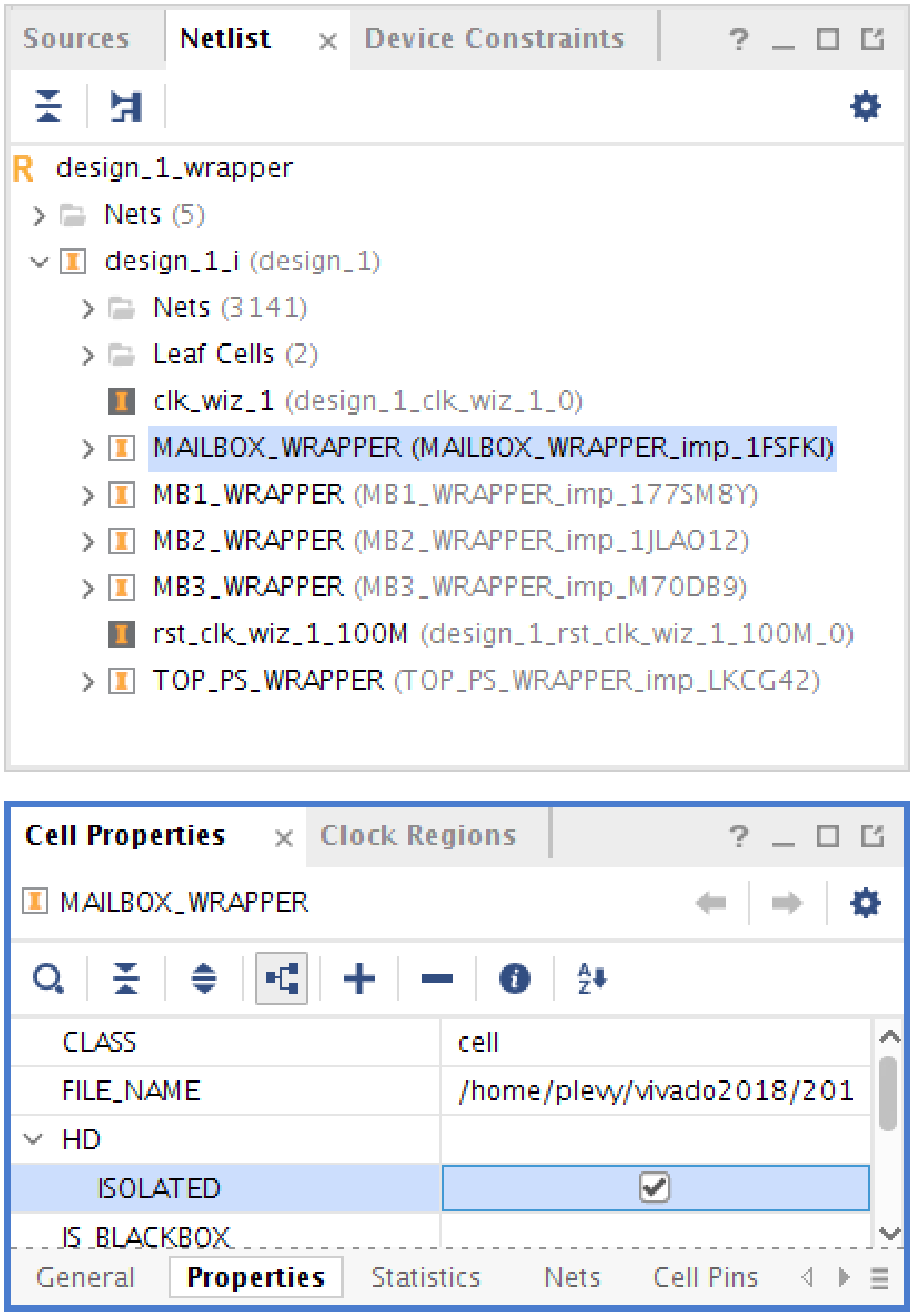
| | |
|---|---|
| CLASS | cell |
| FILE_NAME | /home/plevy/vivado2018/201 |
| IS_BLACKBOX | |
| IS_DEBUGGABLE | |
| IS_MATCHED | |

Search: HD (7 matches)

HD.ISOLATED
HD.ISOLATED_EXEMPT
HD.PLATFORM_WRAPPER
HD.RECONFIGURABLE
HD.RECONFIGURABLE_CONTAINER
HD.RECONFIG_PLATFORM
HD.TANDEM

**HD.ISOLATED**
Property Type: Boolean
Default Value: false

*Figure 33:* **Enabling HD.ISOLATED**



```
design_1_wrapper
  Nets (5)
  design_1_i (design_1)
    Nets (3141)
    Leaf Cells (2)
    clk_wiz_1 (design_1_clk_wiz_1_0)
    MAILBOX_WRAPPER (MAILBOX_WRAPPER_imp_1FSFKJ)
    MB1_WRAPPER (MB1_WRAPPER_imp_177SM8Y)
    MB2_WRAPPER (MB2_WRAPPER_imp_1JLAO12)
    MB3_WRAPPER (MB3_WRAPPER_imp_M70DB9)
    rst_clk_wiz_1_100M (design_1_rst_clk_wiz_1_100M_0)
    TOP_PS_WRAPPER (TOP_PS_WRAPPER_imp_LKCG42)
```

MAILBOX_WRAPPER

| | |
|---|---|
| CLASS | cell |
| FILE_NAME | /home/plevy/vivado2018/201 |
| HD | |
| ISOLATED | ✓ |
| IS_BLACKBOX | |

## Synthesis

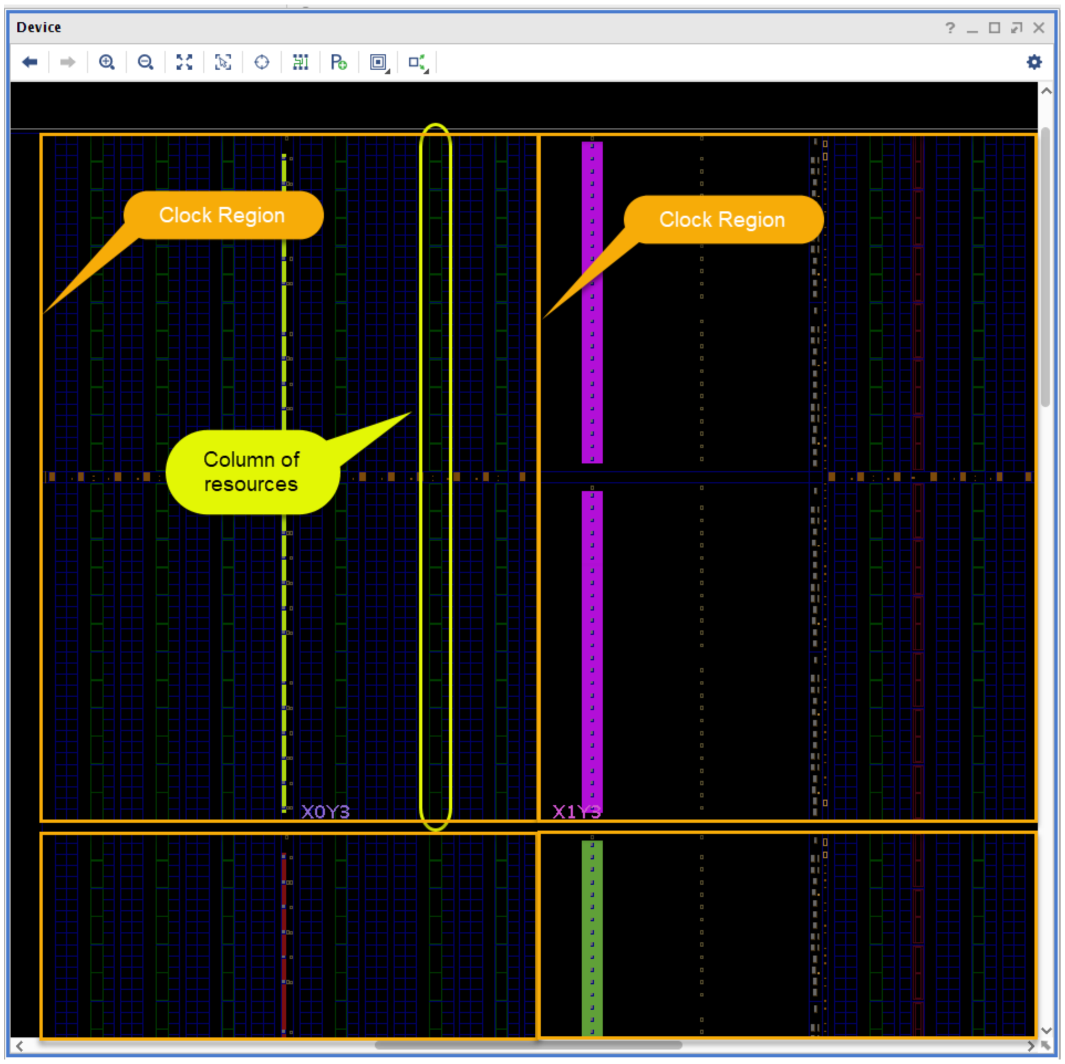
This step creates the actual logic that is used to implement your design.

## Floorplanning

Before an isolated design can be implemented, a floorplanning strategy must be manually entered. This placement restriction uses Pblocks, which are drawn on the device in the Device window. For logic that is outside an isolated hierarchy (such as top level logic), this logic can be in any Pblock.

FPGA arrays are made up of multiple clock regions with columns of logic resources within each clock region. Each logic resource has a fixed height which are stacked one on top of another to make up a column as shown in the following figure. Pblocks span these regions, but if a resource, such as a PCI™ block, which takes an entire clock region, is not enclosed by a single Pblock, it will not be included in the Pblock. Before creating each Pblock, map out where each Pblock will be located.

*Figure 34:* **FPGA Layout**

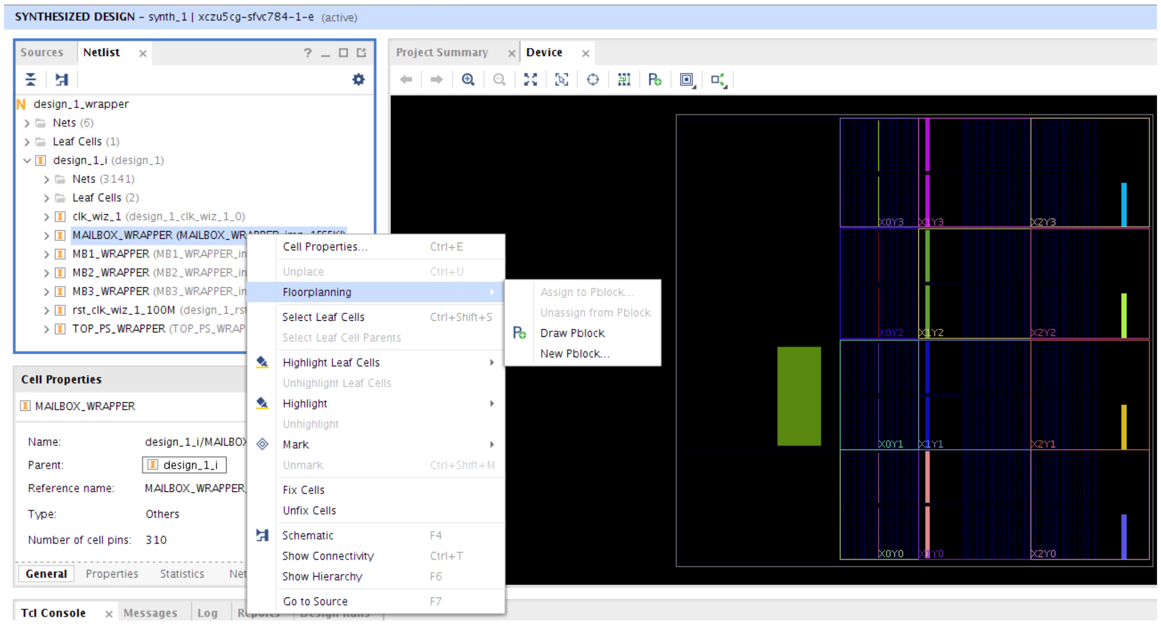


## Drawing Pblocks Using Vivado GUI

Floorplanning is accomplished by using Pblocks. Drawing Pblocks is a multi-step process which is outlined here. This is the recommended process, but not the only method. See Drawing Pblocks Using Tcl Commands.

On a synthesized design:

1. In the NETLIST window, right-click a module to be isolated. For this example, select a wrapper entry under design_1_i.
2. Select **Floorplanning** → **Draw Pblock** as shown in the following figure.



Cell Properties

MAILBOX_WRAPPER

Name: design_1_i/MAILBOX

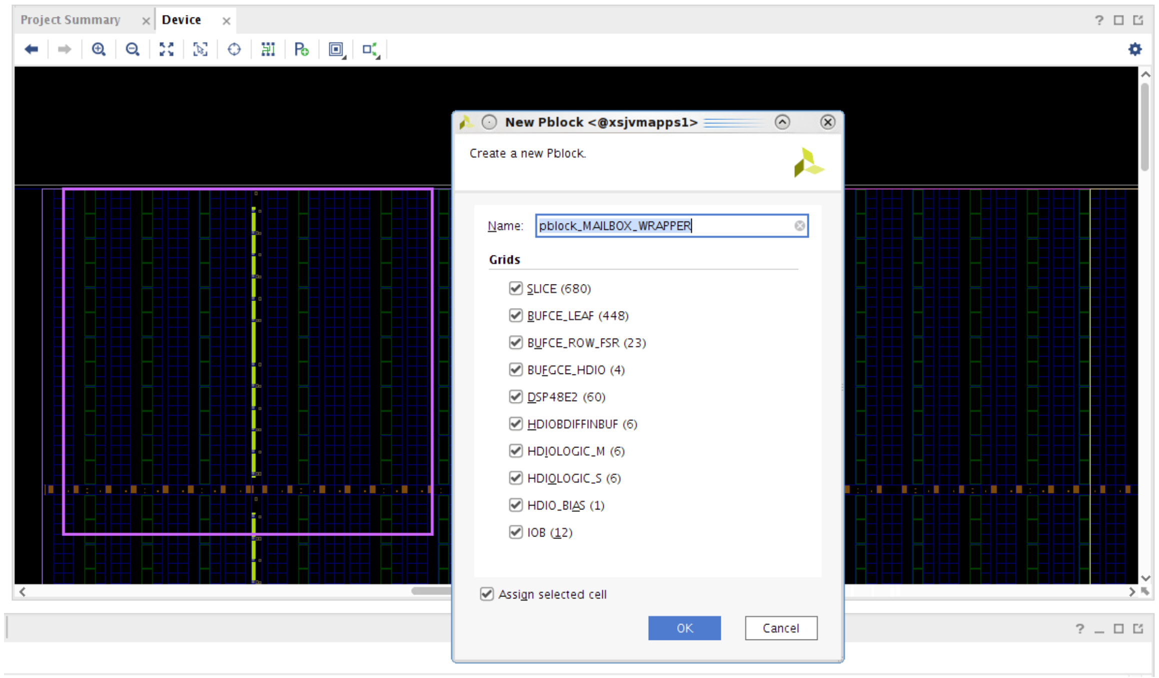
Parent: design_1_i

Reference name: MAILBOX_WRAPPER

Type: Others

Number of cell pins: 310

3. In the Device window, draw a rectangle that defines the initial Pblock as shown in the following figure.



Create a new Pblock.

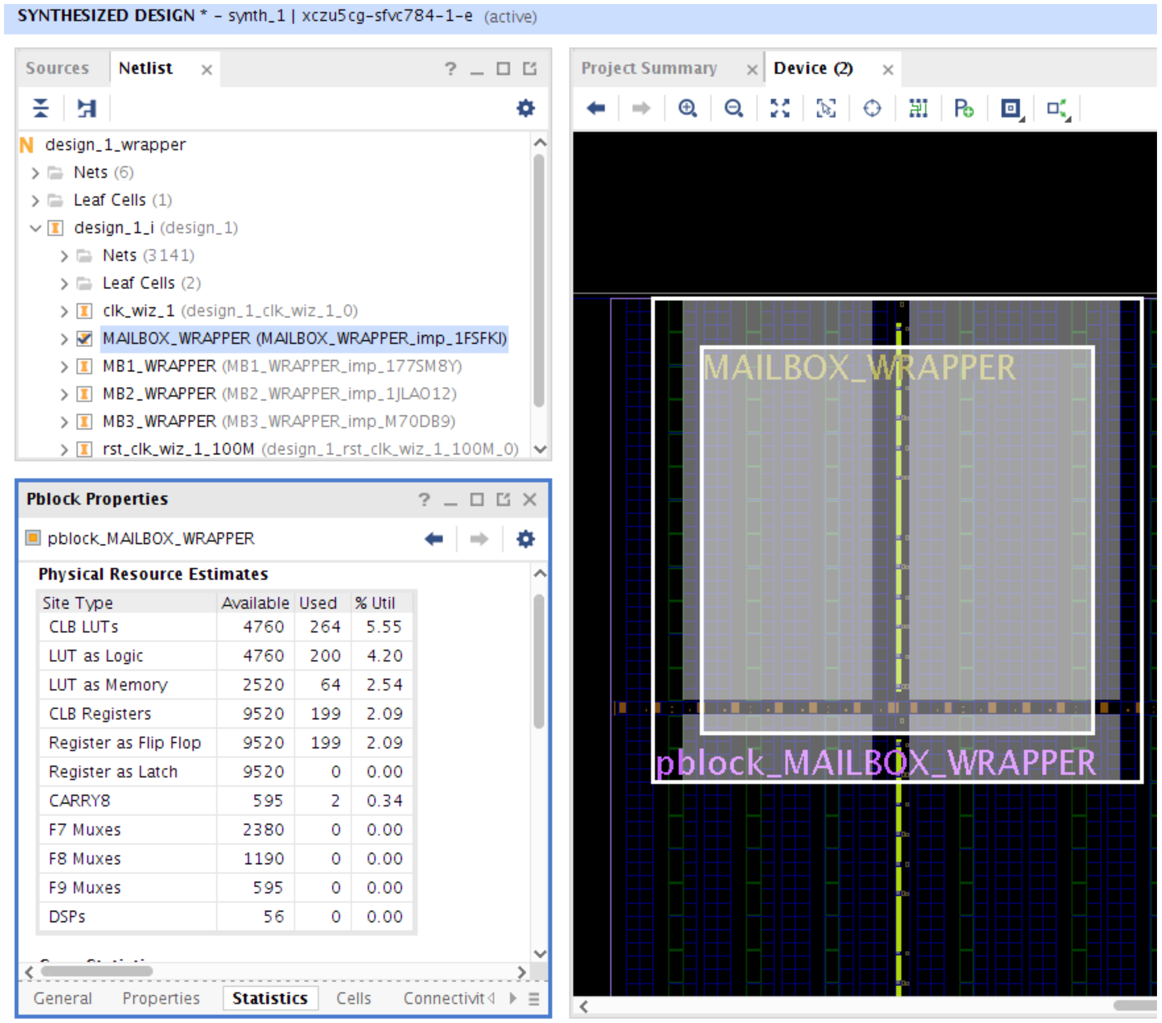
Name: pblock_MAILBOX_WRAPPER

Grids

- [✓] SLICE (680)
- [✓] BUFCE_LEAF (448)
- [✓] BUFCE_ROW_FSR (23)
- [✓] BUFGCE_HDIO (4)
- [✓] DSP48E2 (60)
- [✓] HDIOBDIFFINBUF (6)
- [✓] HDIOLOGIC_M (6)
- [✓] HDIOLOGIC_S (6)
- [✓] HDIO_BIAS (1)
- [✓] IOB (12)

[✓] Assign selected cell

Once a Pblock is created, the Statistics window in Pblock Properties is updated, which outlines the percentage the Pblock resources that satisfies the synthesized requirements of the isolated region hierarchy as shown in the following figure.

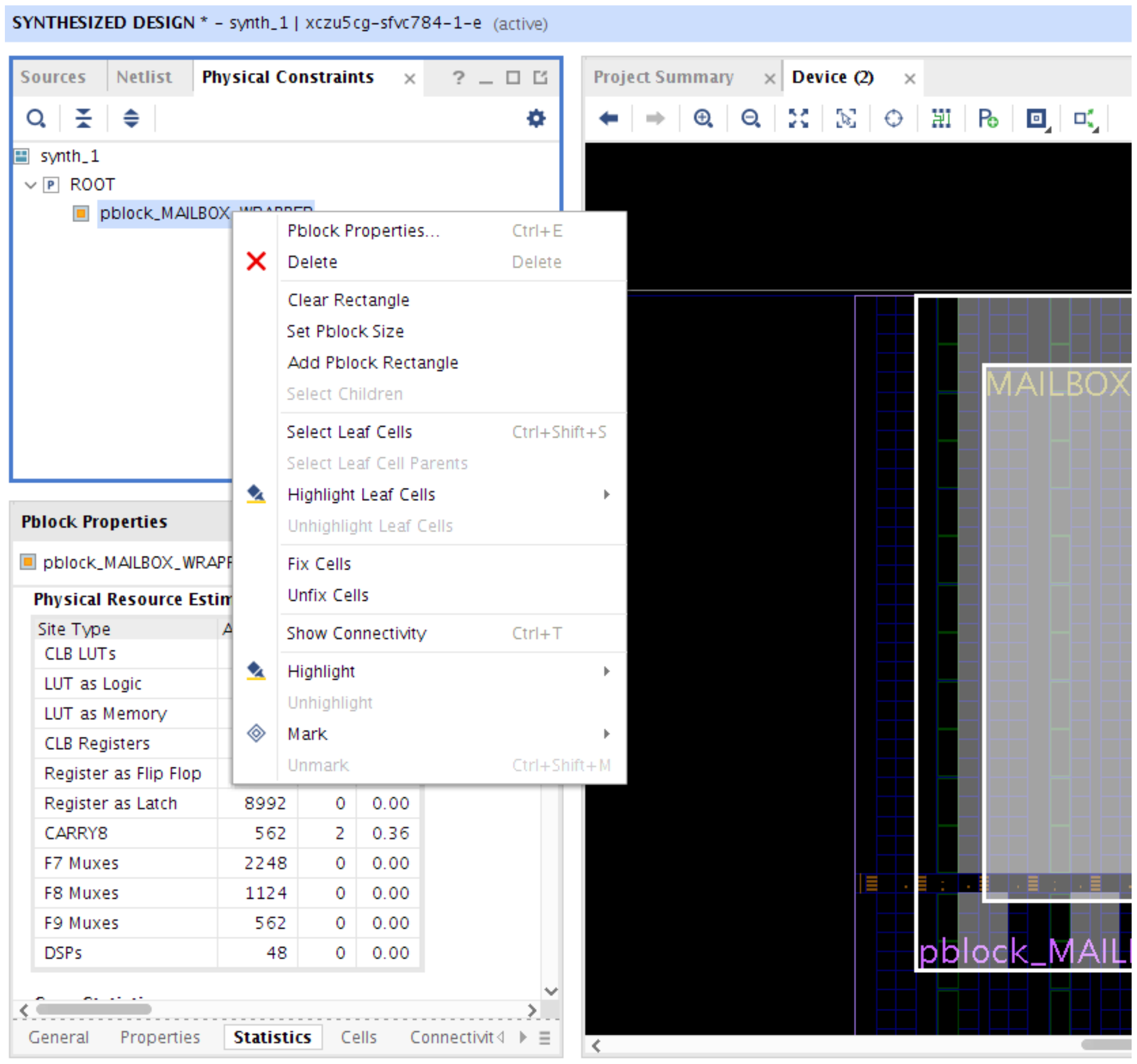

SYNTHESIZED DESIGN * - synth_1 | xczu5cg-sfvc784-1-e (active)
Sources
Netlist
Project Summary
Device (2)
design_1_wrapper
Nets (6)
Leaf Cells (1)
design_1_i (design_1)
Nets (3141)
Leaf Cells (2)
clk_wiz_1 (design_1_clk_wiz_1_0)
MAILBOX_WRAPPER (MAILBOX_WRAPPER_imp_1FSFKI)
MB1_WRAPPER (MB1_WRAPPER_imp_177SM8Y)
MB2_WRAPPER (MB2_WRAPPER_imp_1JLA012)
MB3_WRAPPER (MB3_WRAPPER_imp_M70DB9)
rst_clk_wiz_1_100M (design_1_rst_clk_wiz_1_100M_0)
Pblock Properties
pblock_MAILBOX_WRAPPER
Physical Resource Estimates
MAILBOX_WRAPPER
pblock_MAILBOX_WRAPPER
General
Properties
Statistics
Cells
Connectivit

| Site Type | Available | Used | % Util |
|---|---|---|---|
| CLB LUTs | 4760 | 264 | 5.55 |
| LUT as Logic | 4760 | 200 | 4.20 |
| LUT as Memory | 2520 | 64 | 2.54 |
| CLB Registers | 9520 | 199 | 2.09 |
| Register as Flip Flop | 9520 | 199 | 2.09 |
| Register as Latch | 9520 | 0 | 0.00 |
| CARRY8 | 595 | 2 | 0.34 |
| F7 Muxes | 2380 | 0 | 0.00 |
| F8 Muxes | 1190 | 0 | 0.00 |
| F9 Muxes | 595 | 0 | 0.00 |
| DSPs | 56 | 0 | 0.00 |

4. To add more resources to a Pblock, in the Device window, select the Pblock to highlight the region.

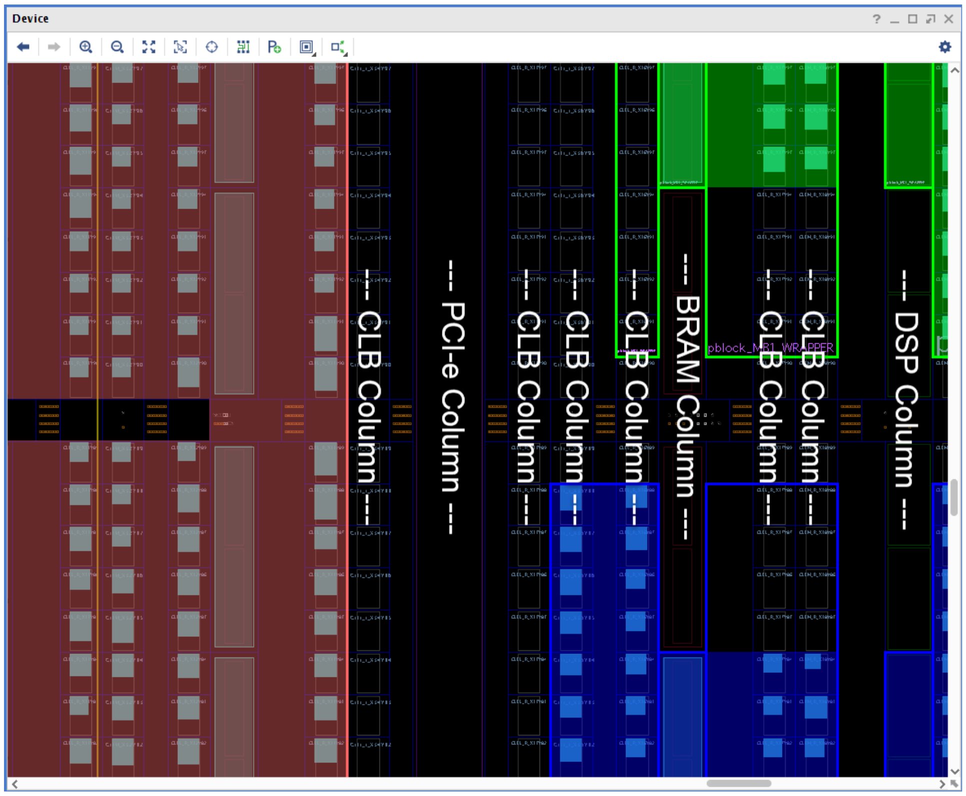
5. Right-click the highlighted region, and select **Add Pblock Rectangle**. As you grow the size of the Pblock, the Statistics window will update showing added resources. Pblocks can be any shape.

6. To delete the Pblock completely, in the Window drop down menu under Physical Constraints, right-click the Pblock entry, and click **Delete** as shown in the following figure.

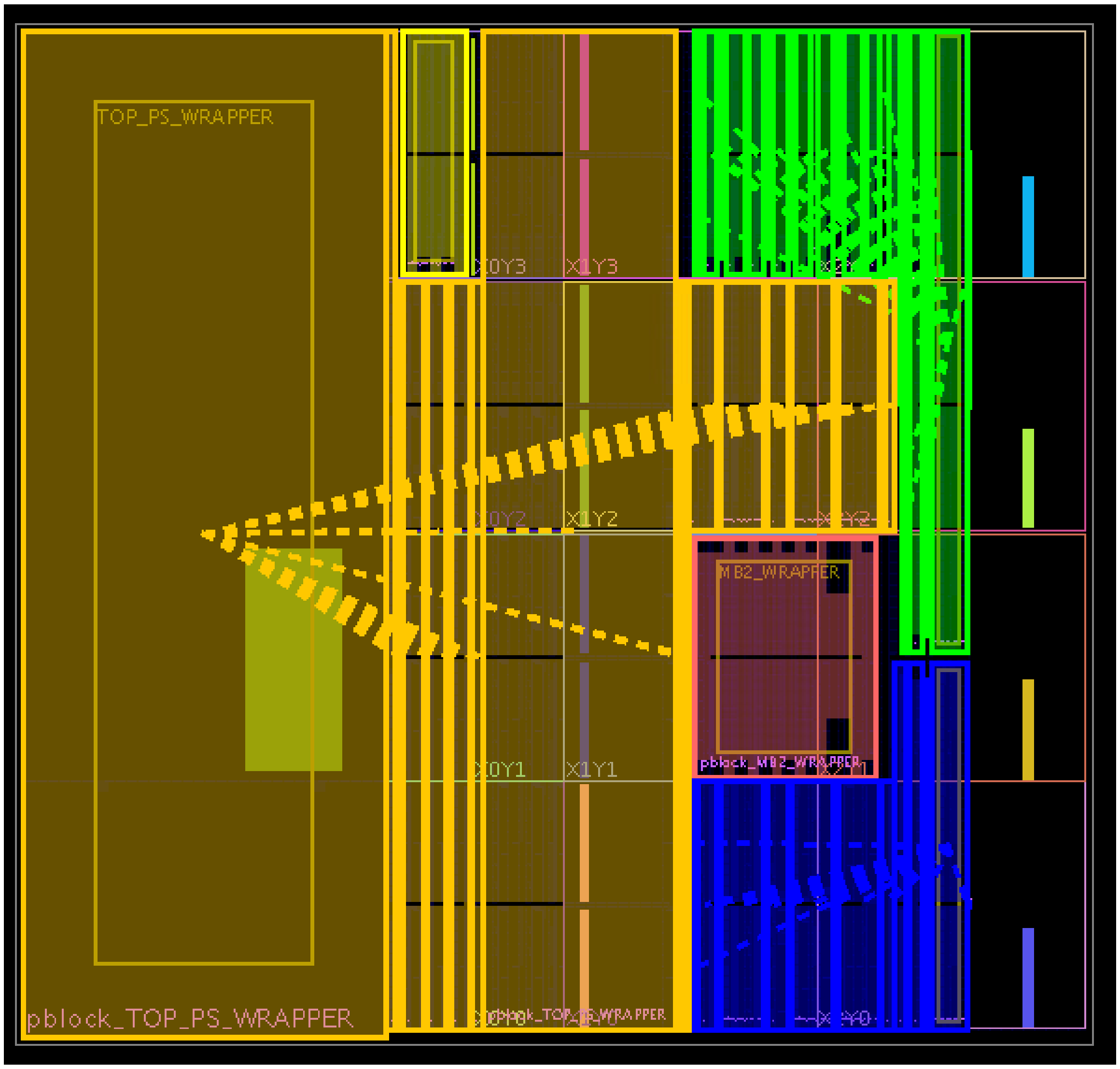


| Site Type | A | | |
|---|---|---|---|
| CLB LUTs | | | |
| LUT as Logic | | | |
| LUT as Memory | | | |
| CLB Registers | | | |
| Register as Flip Flop | | | |
| Register as Latch | 8992 | 0 | 0.00 |
| CARRY8 | 562 | 2 | 0.36 |
| F7 Muxes | 2248 | 0 | 0.00 |
| F8 Muxes | 1124 | 0 | 0.00 |
| F9 Muxes | 562 | 0 | 0.00 |
| DSPs | 48 | 0 | 0.00 |

When drawing Pblocks, you will see shaded resources that show you which resources are included in the Pblock and which are not. All resources not in a Pblock (non-shaded) are fences. In the following figure, the red shaded resources are allocated to one Pblock, the green shaded resources are allocated to a second Pblock, and the blue shaded resources are allocated to a third Pblock. The regions outlines are only guidance artifacts from creating the Pblock and do not define resources. All non-shaded resources are fences, which means those resources are not being used.



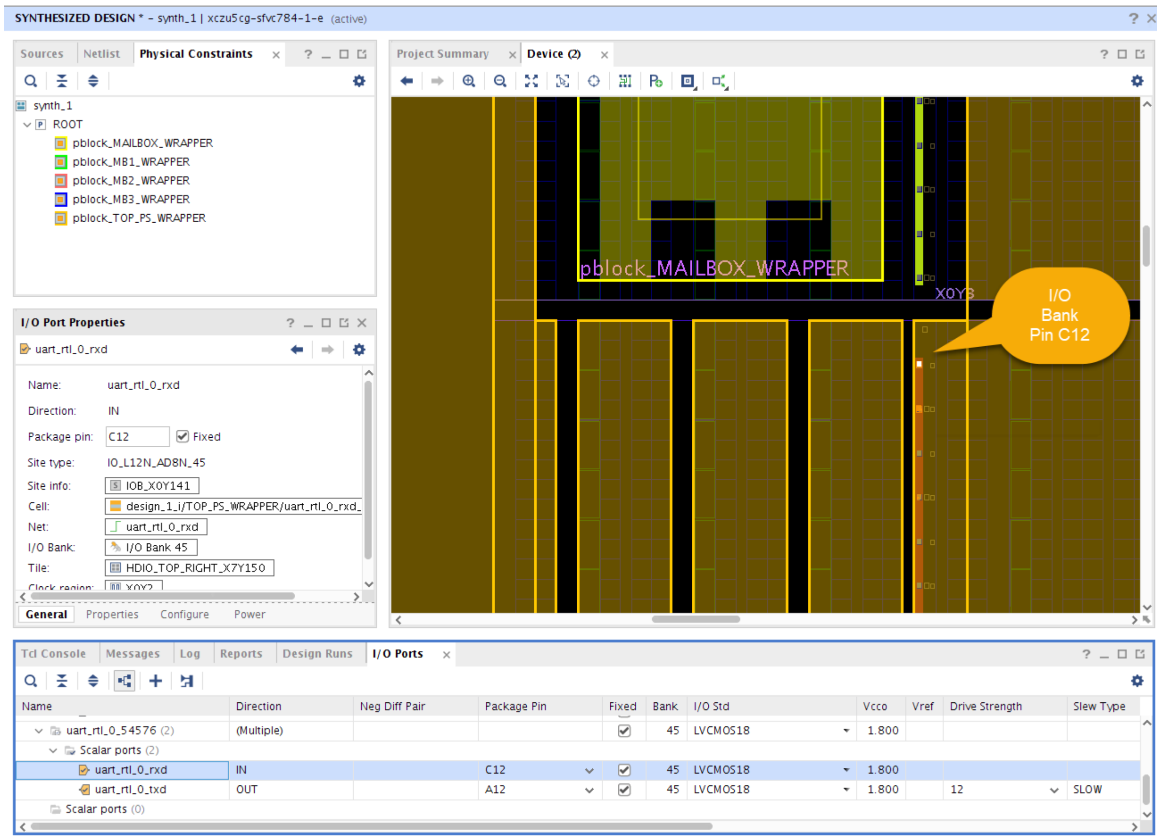
Pblocks can be any shape and size as shown in the following figure. This design enables green, red, and blue Pblocks to easily route to each other. The orange Pblock has all of the top level logic, and the yellow Pblock routes the orange Pblock.



7. To complete the design and make sure your Pblock includes the required I/O Bank, assign pins. In this example, the orange Pblock contains the design I/Os.

   In the drop-down menu, select **IO Ports** to display the I/O Ports window.

   ***Note***: In the following figure, the target I/O block is part of the orange Pblock. Ensure each Pin Assignment is fixed and has the correct I/O standard; otherwise, the final design rule check (DRCs) ran under Generate Bitstream will fail.



synth_1
ROOT
- pblock_MAILBOX_WRAPPER
- pblock_MB1_WRAPPER
- pblock_MB2_WRAPPER
- pblock_MB3_WRAPPER
- pblock_TOP_PS_WRAPPER

**I/O Port Properties**

uart_rtl_0_rxd

| | |
|---|---|
| Name: | uart_rtl_0_rxd |
| Direction: | IN |
| Package pin: | C12 ✓ Fixed |
| Site type: | IO_L12N_AD8N_45 |
| Site info: | IOB_X0Y141 |
| Cell: | design_1_i/TOP_PS_WRAPPER/uart_rtl_0_rxd_ |
| Net: | uart_rtl_0_rxd |
| I/O Bank: | I/O Bank 45 |
| Tile: | HDIO_TOP_RIGHT_X7Y150 |
| Clock region: | X0Y2 |

| Name | Direction | Neg Diff Pair | Package Pin | Fixed | Bank | I/O Std | Vcco | Vref | Drive Strength | Slew Type |
|---|---|---|---|---|---|---|---|---|---|---|
| uart_rtl_0_54576 (2) | (Multiple) | | | ✓ | 45 | LVCMOS18 | 1.800 | | | |
| Scalar ports (2) | | | | | | | | | | |
| uart_rtl_0_rxd | IN | | C12 | ✓ | 45 | LVCMOS18 | 1.800 | | | |
| uart_rtl_0_txd | OUT | | A12 | ✓ | 45 | LVCMOS18 | 1.800 | | 12 | SLOW |
| Scalar ports (0) | | | | | | | | | | |

## Drawing Pblocks Using Tcl Commands

When you draw Pblocks using the GUI as discussed above, the corresponding constraints gets saved to XDC files. In addition to using GUI, you can also use Tcl commands to create Pblocks by executing those commands in the Vivado Tcl Console or saving them in the XDC file. The following example uses one of the modules - `MAILBOX_WRAPPER` of XAPP1336 reference design to show how Tcl commands can be used as part of the XDC file to create an isolated Pblock.

**IMPORTANT!** *The discussion in this section of doing floorplanning using Tcl commands is an advanced topic. The same can be achieved by using Vivado GUI as explained in above section. This section can be used as reference if Pblocks need some fine adjustments or otherwise can be used for understanding how the tools work behind the hood. Xilinx highly recommends to use Vivado GUI for Floorplanning.*

`create_pblock` creates a Pblock that allows adding the logic instances inside it as shown below:

```
create_pblock pblock_MAILBOX_WRAPPER
```

`add_cells_to_pblock` command is then used to assign the isolated module's logic elements / instances to the newly created Pblock as shown below:

```
add_cells_to_pblock Pblock_MAILBOX_WRAPPER [get_cells[list design_1_i/
MAILBOX_WRAPPER]-clear_locs
```

Once isolated modules/cells are assigned to the Pblock, the isolated Pblock must be assigned to a specific range of logics/PUs in the FPGA. Isolated regions/Pblocks can be defined in terms of `SLICEs, RAMB18s, RAMB36s, IOBs, ILOGICs, OLOGICs, IDELAYs, ODELAYs, PLLs, MMCME2s, IN_FIFOs, OUT_FIFOs, BUFGCTRLs, GTXs, DSP48s`. Adding components (specified range of sites) to an isolated Pblock is done by using the Tcl command `resize_pblock` as shown below:

```
resize_pblock Pblock_MAILBOX_WRAPPER -add {SLICE_X2Y181:SLICE_X9Y239
BUFCE_LEAF_X16Y12:BUFCE_LEAF_X55Y15 BUFCE_ROW_FSR_X3Y3:BUFCE_ROW_FSR_X11Y3
DSP48E2_X0Y74:DSP48E2_X2Y95}
```

The generic format for the Tcl command for adding the ranges `resize_pblock` is:

```
resize_pblock -add {<comp name>_XaYb:<comp name>_XcYd
```

Where `<comp name>` = name of desired component, for example: SLICE, DSP48, and RAMB36, etc.

The component name can be identified by pointing at it in the Device view in the Vivado tool and reading it off the bottom right side of the screen. Coordinates can also be identified from the same location that the component name was identified.

`a, b, c, d` = Coordinates of the starting component and the ending components

A full listing of this syntax can be found in *Vivado Design Suite Tcl Command Reference Guide* (UG835).

***Note***: While creating Pblocks, ensure that the SNAPPING_MODE property is set to FINE_GRAINED. See Derived Range and Snapping Mode section.

***Note***: Even though the physical resources are being added using the sites, the end goal is to capture the Programmable Units in entirety and assign them to the Pblocks. See Pblocks and Programmable Units section.

***Note***: Nested Pblocks inside the isolated Pblocks is supported. Users can use nested Pblocks to meet timing closures. IDF supports only one level of nesting. Isolated Pblocks can have nested child Pblocks inside them, but the nested child Pblocks should not have child Pblocks.

Even if the design does not use DSP or block RAM PUs, they must be added to the isolated Pblock so that the routing resources contained by these PUs can be used. As a general rule, all available resources (except BUFGs and BUFHs i.e., global logics) should be assigned to the isolation Pblock unless there is specific need to exclude that resource. This is the default selection when generating the XDC file if you use the Vivado GUI tools as discussed above to create isolated Pblocks. If you are not familiar using Tcl commands to create Pblocks and assign isolated modules to them, you are encouraged to use the Vivado GUI tools for the same as explained in Drawing Pblocks for Floorplanning.

## Constraints

Some initial architecting and floorplanning of the FPGA/SoC, coupled with a list of constraints, is all that is required to achieve isolation of specific modules within a single FPGA or SoC. It is important to note that any logic that is not isolated is, by definition, unconstrained logic and can be placed or routed by the tools in any isolated Pblock. Due to this, it is highly recommended that only global logic remains unconstrained, and global logic should be minimized. The following example shows the constraints generated by the tools when an isolated module is floorplanned into a specific region of the device using Pblocks and Vivado GUI as discussed in the Floorplanning section.

As an example, isolation of the `Pblock_MAILBOX_WRAPPER` module of XAPP1336 is achieved by the following Tcl statements (commands):

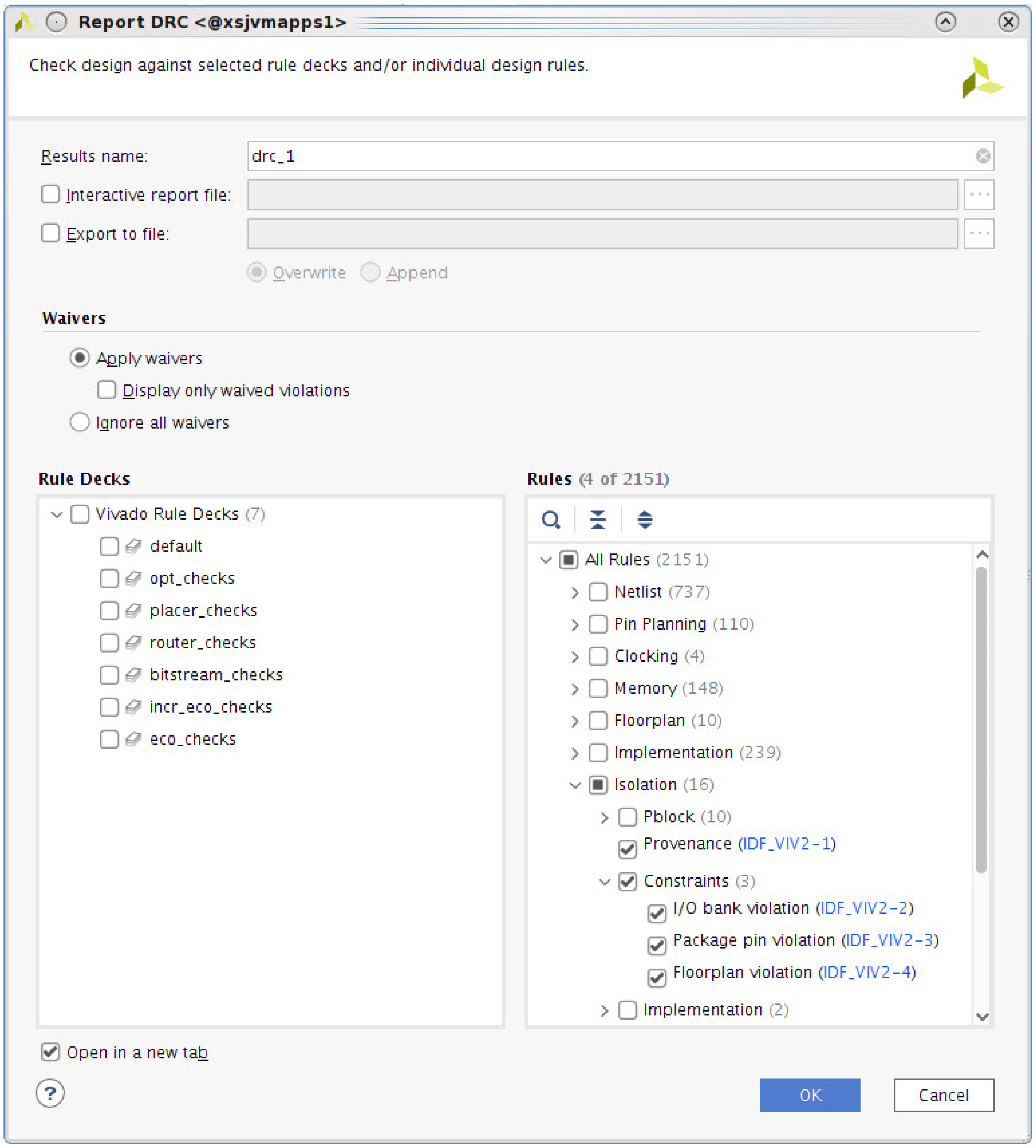
```
create_pblock Pblock_MAILBOX_WRAPPER
resize_pblock pblock_MAILBOX_WRAPPER -add {SLICE_X2Y181:SLICE_X9Y239
BUFCE_LEAF_X16Y12:BUFCE_LEAF_X55Y15 BUFCE_ROW_FSR_X3Y3:BUFCE_ROW_FSR_X11Y3
DSP48E2_X0Y74:DSP48E2_X2Y95}
add_cells_to_pblock pblock_MAILBOX_WRAPPER [get_cells [list design_1_i/
MAILBOX_WRAPPER]] -clear_locs
```

## Vivado IDF Verifier (VIV) Checks 1,2,3, and 4

The Vivado Isolation Verifier (VIV) verifies that an FPGA design partitioned into isolated Pblocks meets stringent standards for fail-safe design. VIV is a collection of six design rule checks (DRCs) intended to aid FPGA developers in producing and documenting fault-tolerant FPGA applications developed with the Xilinx Isolation Design Flow (IDF). Historically, VIV (VIV1) was a Tcl script that ran in the Vivado tool framework in the form of user defined DRCs. As part of adding support of IDF for UltraScale+ architecture, Xilinx has released VIV2 which has the same set of six DRCs, but they are now are a part of Vivado built-in system DRCs. For convenience, in this application note VIV2 will be referred to as VIV.

Vivado Isolation Verifier (VIV) 2.0 is available in Vivado® Design Suite 2018.3, and is enabled by executing the **set_param hd.enableIDFDRC true** command in Vivado's Tcl Console. Refer to *Vivado Isolation Verifier User Guide* (UG1291) for more detailed information on VIV2.

1. To run this tool, open the Report DRC window in the Reports drop-down menu as shown in the following figure.



Report DRC <@xsjvmapps1>

Check design against selected rule decks and/or individual design rules.

Results name: drc_1

☐ Interactive report file:

☐ Export to file:

[✓ Overwrite] Append

**Waivers**

[✓ Apply waivers]

☐ Display only waived violations

Ignore all waivers

**Rule Decks**

- ☐ Vivado Rule Decks (7)
  - ☐ default
  - ☐ opt_checks
  - ☐ placer_checks
  - ☐ router_checks
  - ☐ bitstream_checks
  - ☐ incr_eco_checks
  - ☐ eco_checks

**Rules** (4 of 2151)

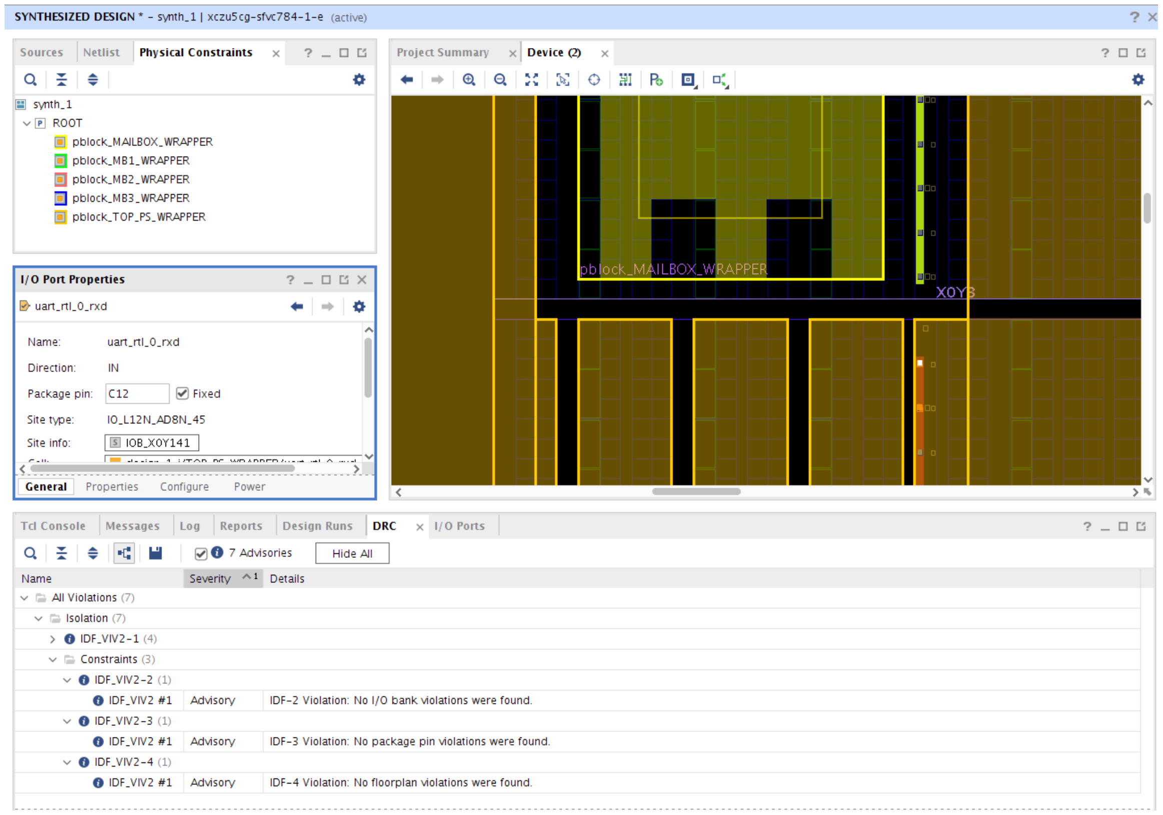
- ■ All Rules (2151)
  - ☐ Netlist (737)
  - ☐ Pin Planning (110)
  - ☐ Clocking (4)
  - ☐ Memory (148)
  - ☐ Floorplan (10)
  - ☐ Implementation (239)
  - ■ Isolation (16)
    - ☐ Pblock (10)
    - ✓ Provenance (IDF_VIV2-1)
    - ✓ Constraints (3)
      - ✓ I/O bank violation (IDF_VIV2-2)
      - ✓ Package pin violation (IDF_VIV2-3)
      - ✓ Floorplan violation (IDF_VIV2-4)
    - ☐ Implementation (2)

✓ Open in a new tab

OK Cancel

2. In the Rules window, under Isolation, select **Provenance** and **Constraints**, and click **OK**.

The result displays in the DRC window as shown in the following figure.



synth_1
ROOT
- pblock_MAILBOX_WRAPPER
- pblock_MB1_WRAPPER
- pblock_MB2_WRAPPER
- pblock_MB3_WRAPPER
- pblock_TOP_PS_WRAPPER

I/O Port Properties: uart_rtl_0_rxd

| | |
|---|---|
| Name: | uart_rtl_0_rxd |
| Direction: | IN |
| Package pin: | C12 [✓ Fixed] |
| Site type: | IO_L12N_AD8N_45 |
| Site info: | IOB_X0Y141 |

| Name | Severity | Details |
|---|---|---|
| All Violations (7) | | |
| Isolation (7) | | |
| IDF_VIV2-1 (4) | | |
| Constraints (3) | | |
| IDF_VIV2-2 (1) | | |
| IDF_VIV2 #1 | Advisory | IDF-2 Violation: No I/O bank violations were found. |
| IDF_VIV2-3 (1) | | |
| IDF_VIV2 #1 | Advisory | IDF-3 Violation: No package pin violations were found. |
| IDF_VIV2-4 (1) | | |
| IDF_VIV2 #1 | Advisory | IDF-4 Violation: No floorplan violations were found. |

Ensure there are no errors in the DRC report and then run Implementation.

## Constraint Check (VIV-Constraints)

VIV, on the floorplan, checks the following:

### IDF_VIV2-1 – Provenance:

IDF_VIV2-1 is an advisory DRC documenting the circumstances of the run. It also validates that the design has at least two modules marked as isolated (using the HD.ISOLATED property). Nets driven by cells marked HD.ISOLATED_EXEMPT are exempt from inter-region isolation rules and are listed in the IDF_VIV2-1 output. This DRC also checks that the SNAPPING_MODE property of each Isolated Pblock is set to FINE_GRAINED.

### IDF_VIV2-2 - I/O Bank Violation:

Pins from different isolation regions are not co-located in the same IOB bank.

***Note***: While VIV does fault such conditions, only specific security related applications require such bank isolation. Most applications allow sharing of IO banks. Bank sharing is dependent on the specific application and users need to take decision to allow bank sharing per use case basis.

**IDF_VIV2-3 - Package Pin Violation:**

Pins from different isolation groups are not physically adjacent, vertically or horizontally, at the die level.

Pins from different isolation groups are not physically adjacent at the package level. Adjacency is defined in eight compass directions: north, south, east, west, northeast, southeast, northwest, and southwest.

**IDF_VIV2-4 - Floorplan Violation:**

The Pblock constraints in the XDC file are defined by the user such that a minimum fence as listed in Table 1 exists between isolated Pblocks. This means Pblocks are drawn with valid fence between them.

## Implementation

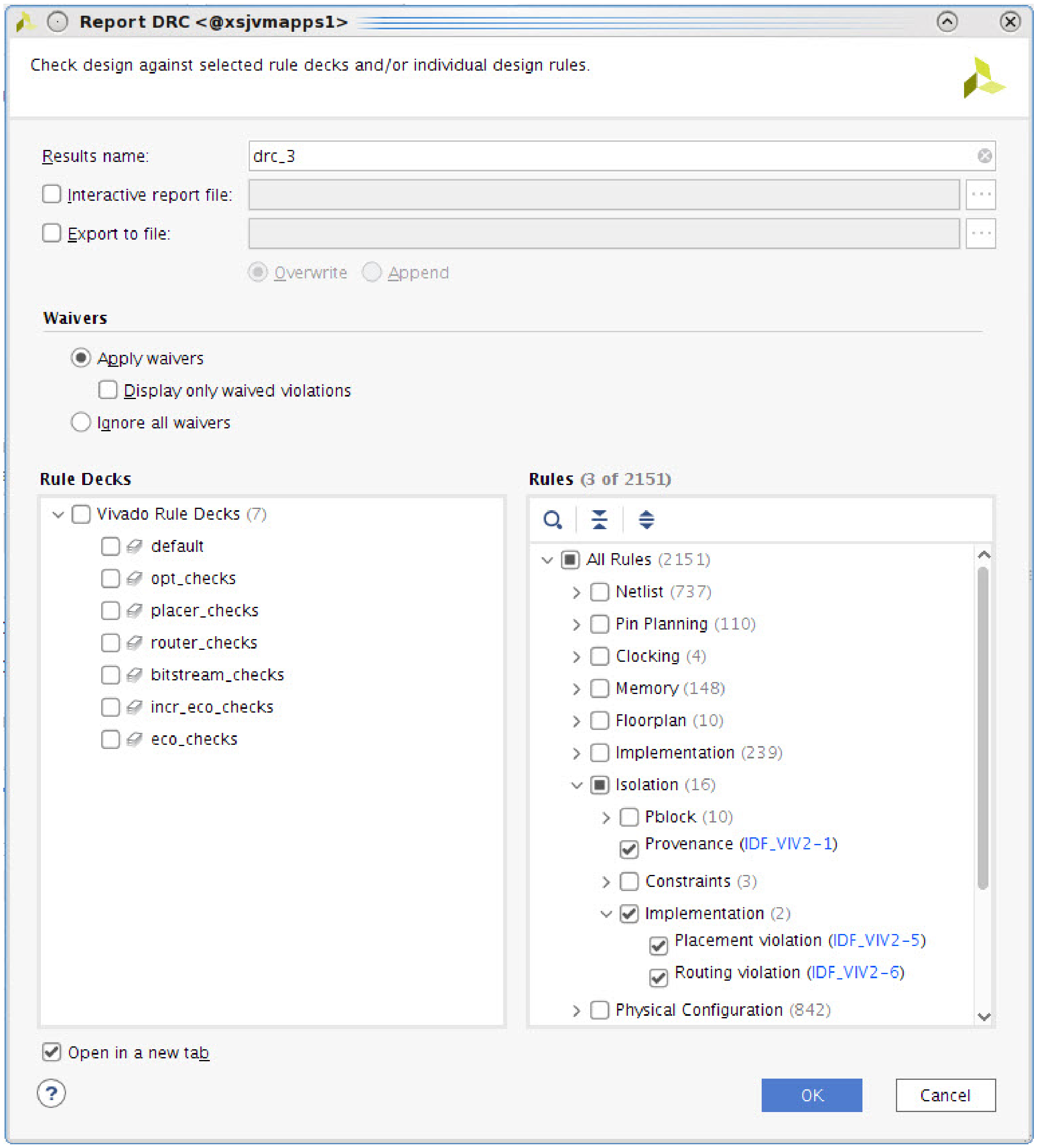
This step performs resource placement, which uses the Pblock isolation rules, followed by routing which uses only trusted routes that avoids using routing resources in the fences.

## Vivado Isolation Verifier (VIV) Checks 5 and 6

After implementation completes (placement and routing) without any errors, VIV is run again on the implemented design to validate that the required isolation is built into the design.

To run these DRCs, open the Report DRC window in the Reports drop-down menu, and select Implementation under Isolation, as shown in the following figure.

*Figure 35:* **Report DRC Window**



Report DRC <@xsjvmapps1>

Check design against selected rule decks and/or individual design rules.

Results name: drc_3

☐ Interactive report file:

☐ Export to file:

Overwrite  Append

**Waivers**

● Apply waivers

☐ Display only waived violations

○ Ignore all waivers

**Rule Decks**

- ☐ Vivado Rule Decks (7)
  - ☐ default
  - ☐ opt_checks
  - ☐ placer_checks
  - ☐ router_checks
  - ☐ bitstream_checks
  - ☐ incr_eco_checks
  - ☐ eco_checks

**Rules** (3 of 2151)

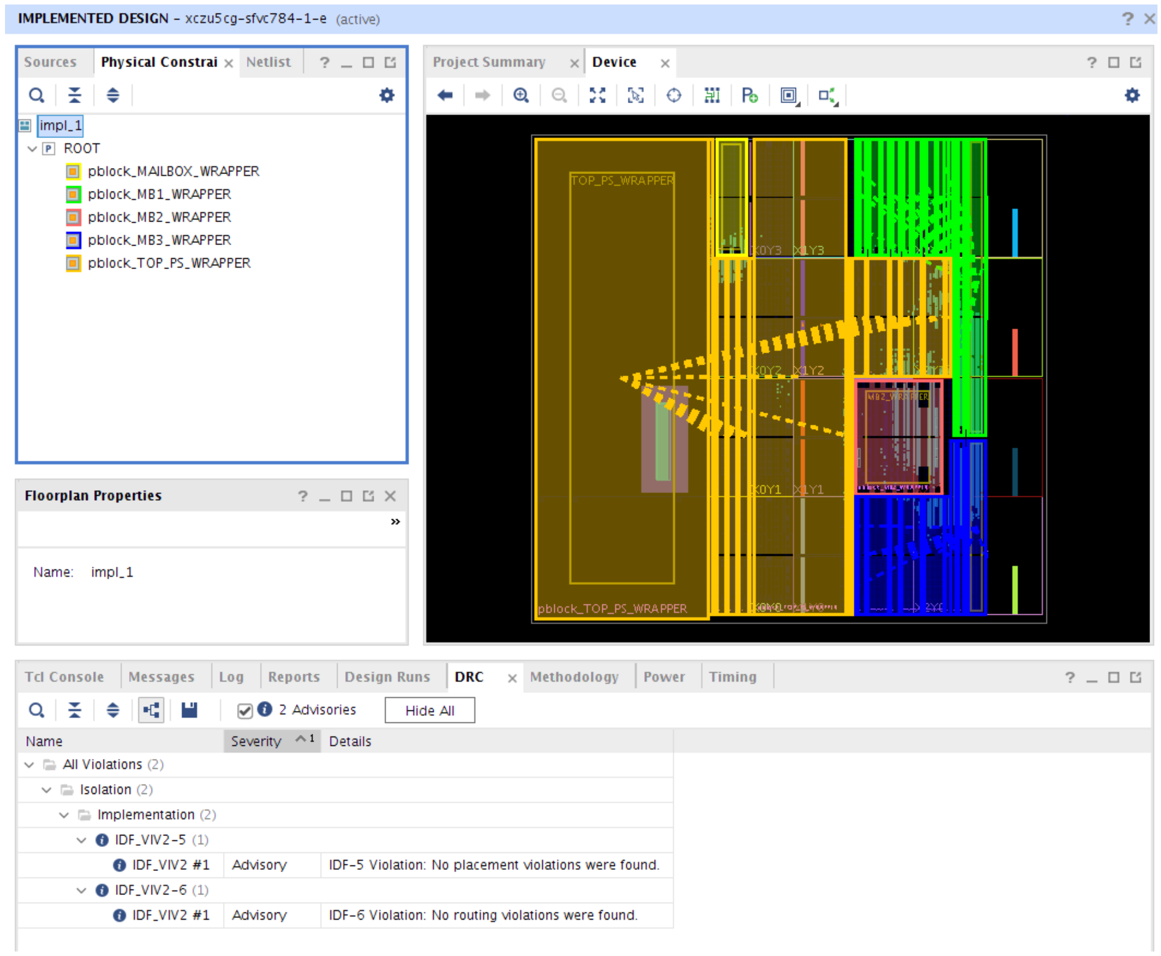
- ■ All Rules (2151)
  - ☐ Netlist (737)
  - ☐ Pin Planning (110)
  - ☐ Clocking (4)
  - ☐ Memory (148)
  - ☐ Floorplan (10)
  - ☐ Implementation (239)
  - ■ Isolation (16)
    - ☐ Pblock (10)
    - ✓ Provenance (IDF_VIV2-1)
    - ☐ Constraints (3)
    - ✓ Implementation (2)
      - ✓ Placement violation (IDF_VIV2-5)
      - ✓ Routing violation (IDF_VIV2-6)
  - ☐ Physical Configuration (842)

✓ Open in a new tab

OK  Cancel

The result is displayed in the DRC window as shown in the following figure.

*Figure 36:* **DRC Final Report**



impl_1
ROOT
- pblock_MAILBOX_WRAPPER
- pblock_MB1_WRAPPER
- pblock_MB2_WRAPPER
- pblock_MB3_WRAPPER
- pblock_TOP_PS_WRAPPER

Name: impl_1

| Name | Severity | Details |
|---|---|---|
| All Violations (2) | | |
| Isolation (2) | | |
| Implementation (2) | | |
| IDF_VIV2-5 (1) | | |
| IDF_VIV2 #1 | Advisory | IDF-5 Violation: No placement violations were found. |
| IDF_VIV2-6 (1) | | |
| IDF_VIV2 #1 | Advisory | IDF-6 Violation: No routing violations were found. |

Ensure there are no errors in the DRC report.

## Final Isolation Verification (VIV - Implementation)

VIV, on the implemented design, checks the following:

### IDF_VIV2-5 - Placement Violation

In contrast to IDF_VIV2-4 which checks floorplanned Pblock boundaries, IDF_VIV2-5 checks actual placement of logic on how it was implemented by the tools. Two checks are performed

- Firstly, a search for adjacent logic from distinct isolation modules is performed. In this context a placed logic is considered adjacent to another placed isolated logic if the separation between the logic placed PUs is not composed of a fence (unprogrammed) PU tiles.
- Secondly, a check is performed to determine that placed top-level logic does not form a potential path from one isolation module to another.

**IDF_VIV2-6 - Routing Violation**

Isolated routing must be separated by an adequate fence and trusted routing must satisfy the following:

- Inter-region routes have loads in exactly one isolation group
- No routing switches (PIPs) are used in the fence
- Inter-region routes cannot share a tile unless source regions match and load regions match
- An intra-region route cannot enter a fence tile or an isolated tile of another isolation group unless it is driven by a cell marked with the HD.ISOLATED_EXEMPT property

***Note***: Vivado tools automatically take care of the trusted routing and users do not need to do anything extra other than having valid fences in their designs.

# Design Guidance

## Concept of Ownership

This section is an extension of the ownership discussion in Mapping the Logical Ownership to the Physical Ownership.

## Logical Ownership

Logical ownership relates to HDL partition / isolation modules

- Each HDL file logically owns the logic instantiated within it.
- Only global logic (BUFG, MMCM etc.) is allowed at the top logical level of design.
- Only global logic can be at the top, but it can also instantiated in an isolated hierarchy if the HD.ISOLATED_EXEMPT property is used.

## Physical Ownership

Physical ownership relates to Isolated Regions (Pblocks)

- No user logic component can be used if it is not physically owned by an isolated Pblock.
    - Physical ownership is defined by the addition of the available components/PUs to the isolated Pblock definition in the XDC file/ Pblock drawn using Vivado GUI.
    - A Pblock is called isolated Pblock if it is mapped to an isolated module.
    - `resize_Pblock` constraint allows a range of slices placed in the Pblock to be used by the developer.
- Global logic, while not typically owned by any isolated logic, must be associated with any isolated Pblock, except for BUFGs and BUFHs. If the global logic is part of an isolated module and not a BUFG or BUFH, it must be associated with the isolated Pblock of that isolated module. Additionally, remember to exempt global logic using HD.ISOLATED_EXEMPT property.

## Trusted Routing Rules

Trusted routing is automatic when HD.ISOLATED is set on the isolated modules of a design. The design tools recognize the communication between isolated Pblocks and use the trusted routing resources. Thus, the tools do everything automatically for you. However, some rules must be adhered to if safe communication between isolated modules is to be guaranteed. The rules outlined in this section are a general guideline that the isolation designers must know with regards to the isolation routing concepts.

***Note***: The Vivado tools automatically follow each of the rules listed below. As a good design practice, it is recommended to understand these IDF rules and take the suggested precautions to avoid violating them, even though Vivado tools takes the necessary steps to ensure that they are followed. The changes done by the tools to incorporate trusted routing rules appear in the netlist after the synthesis phase.

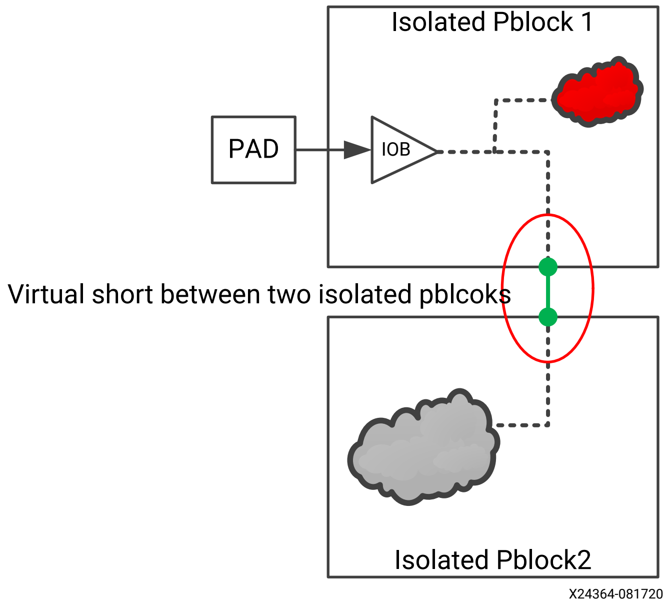
Following are the rules listed for trusted routing:

*Rule 1*: Feed-through signals are not allowed without buffering of some kind such as LUT or FF.

- If a signal is directly connected to both an input port and an output port, it must be buffered
- Direct instantiation of a buffer (LUT1, for example) is recommended. This isolates the wire segments in each of the isolated Pblocks with the LUT buffer, preventing a common (shorted) net throughout both regions

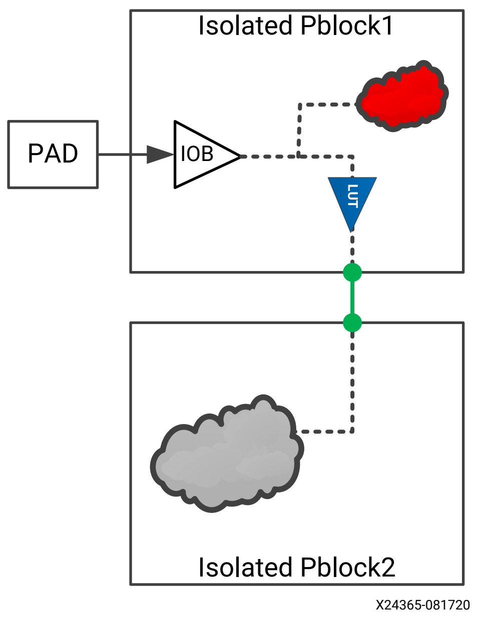
*Problem*: The following figure shows IDF rules violation because of the short between two isolated Pblocks.

*Figure 37:* **Short created by Feed-Trough Signal**



*Solution*: Feed-through signals need to be buffered. This can be achieved either through HDL coding to ensure that there is some unique driver on the output port, by direct instantiation of a LUT, or flip-flop buffer as shown in the following figure, or by letting the Vivado tools address the issue using separate wire segments.

*Figure 38:* **Elimination of Short by Instantiation of a LUT Buffer**

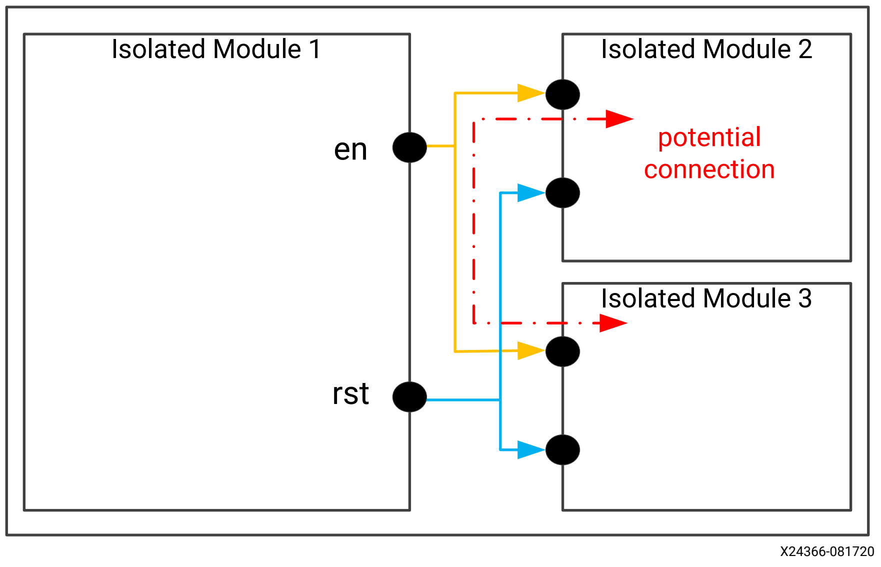


*Rule 2*: An isolated module's output port (driver) cannot connect to more than one isolated module's input port (load). Stated differently, port-to-port connections must be singular:

- Two different ports need to be created for such a connection.
- Each port must not violate Rule 3

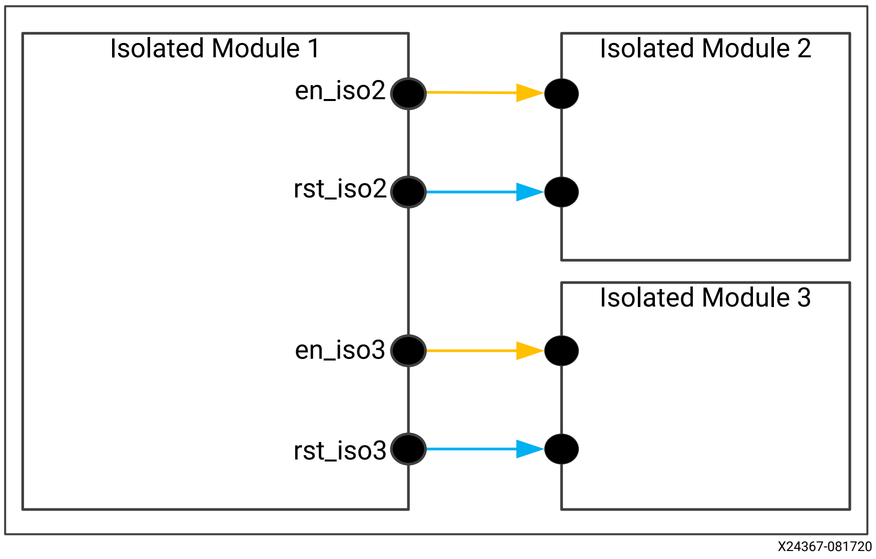
*Problem*: In the following figure, the orange and blue colored nets creates a potential connection between Isolated Module 2 and Isolated Module 3 which was never intended in the original design, and hence violates isolation between Isolated Modules 2 and 3.

*Figure 39:* **Multi-Port Connection Causing Connectivity between Isolated Module 2 and Isolated Module 3**



*Solution:* You must create multiple output ports as many as needed to drive multiple input ports of other isolated modules as shown in the following figure, or you can let the Vivado tools split the offending ports. If you choose to let Vivado do this for you, the HDL does not need to be modified to achieve this. The tools split the ports at the netlist level.

*Figure 40:* **Elimination of Unintended Connection Using Multiple Output Ports**

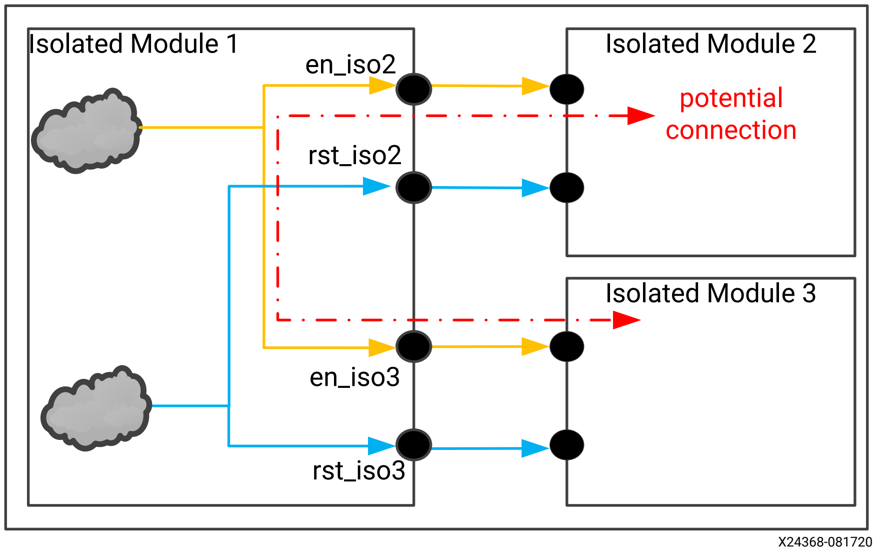


*Rule 3*: One signal cannot drive two different output ports of the same function:

- Each port must have a unique driver.
- Direct instantiation of a buffer (LUT1, for example) is necessary. This isolates regions with the LUT preventing a shorted net.

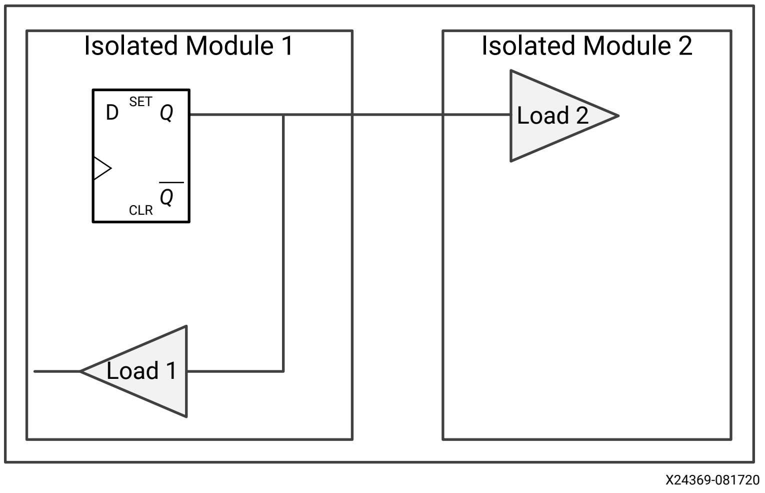
*Problem*: In the following figure, even though there are separate output ports on Isolated Module 1 going to two different isolated modules – 2 and 3, inside Isolated Module 1 it is the same orange or blue net going to two different isolated modules. Thus, Isolated Module 2 and 3 might get connected via the orange or blue nets.

*Figure 41:* **Unintended Connectivity between Isolated Module 2 and Isolated Module 3 Inside Isolated Module 1**



Another variant of this problem can be when driver of an isolated module drives itself as well as another isolated module as shown in the following figure. This also violates the IDF rules for Trusted Routing.

*Figure 42:* **Unintended Connectivity between Isolated Module 1 and Isolated Module 2**



*Solution*: Each port driver needs to be buffered. This can be achieved through HDL coding to ensure that there is some unique driver for each output port, by direct instantiation of a LUT buffer or flip-flip as shown below in the following figures, or by letting the Vivado tools address the issue using separate wire segments.

*Figure 43:* **Elimination of Unintended Connectivity Using LUT Buffers**

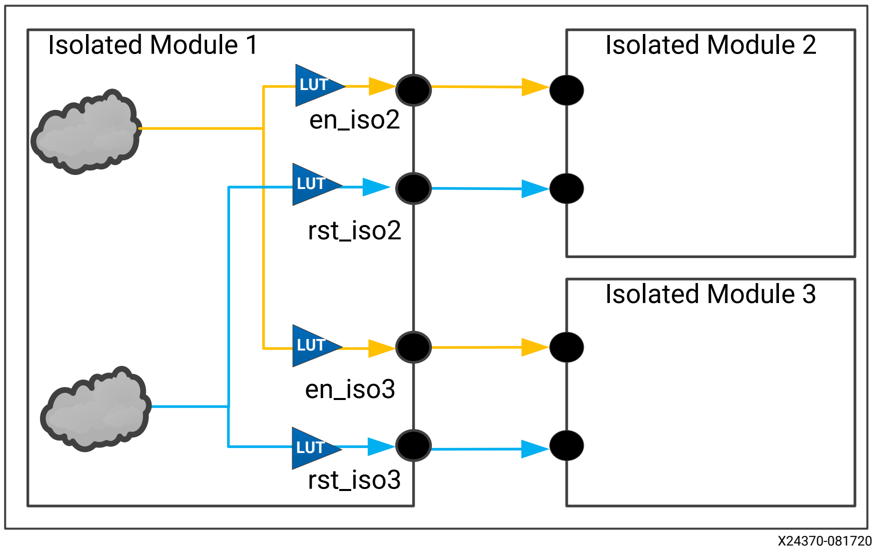
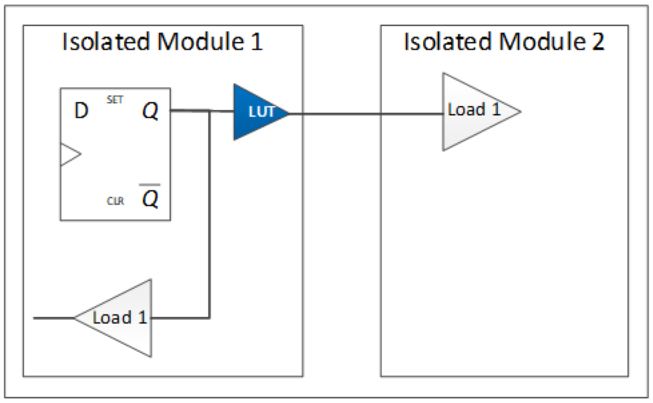


*Figure 44:* **Elimination of Unintended Connectivity Using LUT Buffers**



The developer can manually modify the HDL to meet the IDF rules and manually split the nets, as well as add LUTs as described below. Additionally, this automatic modification by Vivado tools can be disabled by setting the parameter HD.ISOLATED_DISABLE_NETSPLIT to 1 using the following syntax:

```
set_parameter HD.ISOLATED_DISABLE_NETSPLIT 1
```

However, it is highly recommended that you let the tools take care of the trusted routing.

# Hints and Guidelines

## Resources to Add in an Isolated Pblock

It is generally advised to add all the available resources i.e. Programmable Units (PUs) within an isolated Pblock (except those that are needed for the fence), even if the logic tile/PU is not used in the design, because excluding them also excludes using their respective routing resources. Failure to do so, while not an error, might produce designs that are very difficult to route. This is true even for the I/O clock buffers.

However, in the Vivado tool, BUFGs and BUFHs must not be included in the isolated Pblocks unless you want the global clock isolated (not advisable unless the design warrants it to be isolated). All other components must be included.

***Note***: Resources that are not associated with a Pblock cannot be used even if it is needed by top level logic. Resources not assigned to any Pblock are invisible to placement and routing tools. If global clock in an IP is owned by any isolated module it needs to be exempted from isolation using HD.ISOLATION_EXEMPT property.

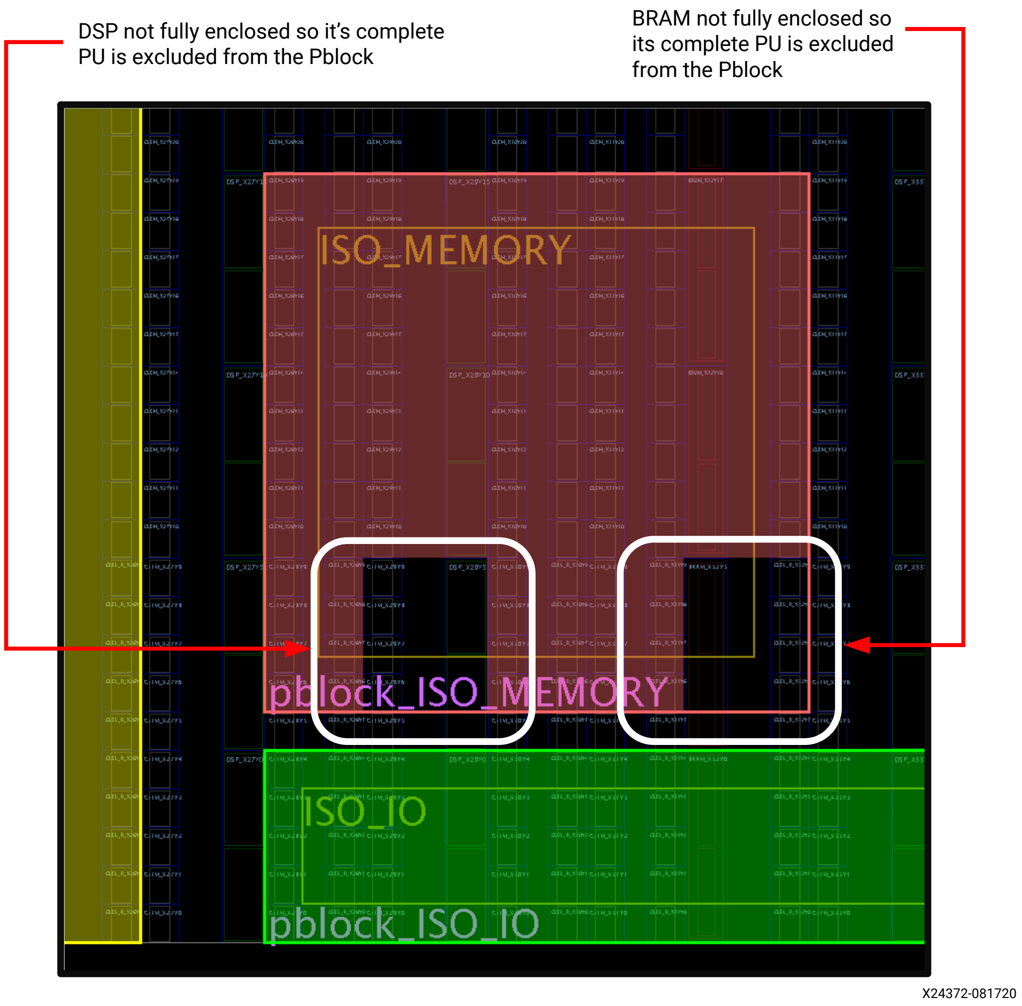
## Shading Pblocks

Looking at Pblocks might be easy but user floorplans are not always simple. IDF highly recommends taking advantage of the highlighting features of Vivado. The following Tcl script will highlight all the Pblocks in the design:

```
set pblocks [get_pblocks *];set ci 1;foreach pblock $pblocks
{highlight_objects -color_index [expr {1 + ($ci % 19)}] [get_pblocks
$pblock]; incr ci}
```

Shading the Pblocks tells the user what resources are included in it. Although shading can be seen when the Pblock is selected, highlighting it helps for better visibility. Additionally, it helps to differentiate between different Pblocks. In a highlighted Pblock, resources that have color are added to the Pblock and the regions that are black are not included. In the following figure, some of the PUs are not included for the red color highlighted Pblock. This is apparent from the highlighting, since entirety of the DSP and BRAM is not included in the original Pblock boundary. Hence, the corresponding PU tiles such as CLEs also did not get included.

*Figure 45:* **Shading of Pblocks for Better Visualization of Its Included Resources**


DSP not fully enclosed so it's complete PU is excluded from the Pblock
BRAM not fully enclosed so its complete PU is excluded from the Pblock
ISO_MEMORY
pblock_ISO_MEMORY
ISO_IO
pblock_ISO_IO
X24372-081720

## HDIO versus HPIO PUs

### HDIO

As listed in Table 1, HDIO has the following fence rule:

- Minimum fence width (vertical fence) is ½ HDIO
- Minimum fence height (horizontal fence) is ½ HDIO PU

HDIO PU is a special PU where ½ of it can be used as fence. That also means half of it can be included in a Pblock.

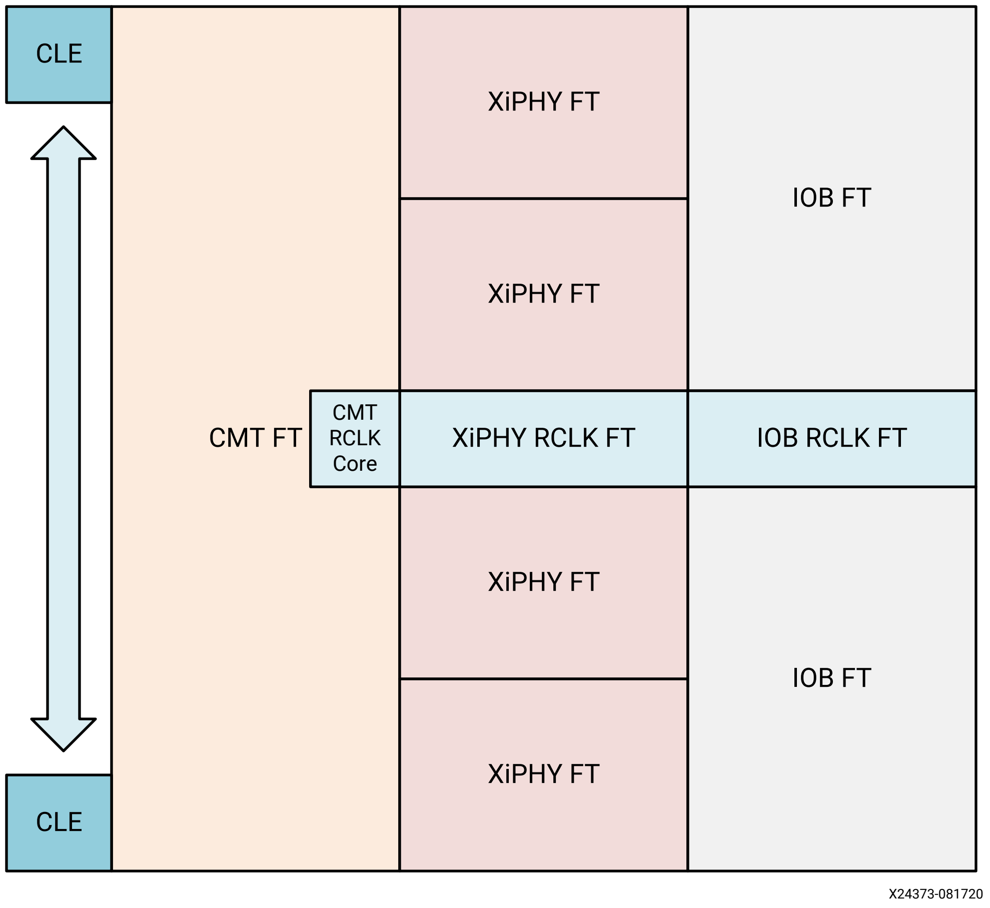
### HPIO

As listed in Table 1, HPIO has the following fence rule:

- Minimum fence width (vertical fence) is 1 HPIO PU
- Minimum fence height (horizontal fence) is 1 HPIO PU

A HPIO tile consists of 4 XiPHYs and a CMT tile along with the IOBs. And the HPIO PU, in addition to the HPIO tile components consists of CLEs sharing the interconnects with HPIO tile. This whole PU must be used as fence whenever isolation is needed. The following figure shows the HPIO PU with all the component tiles.

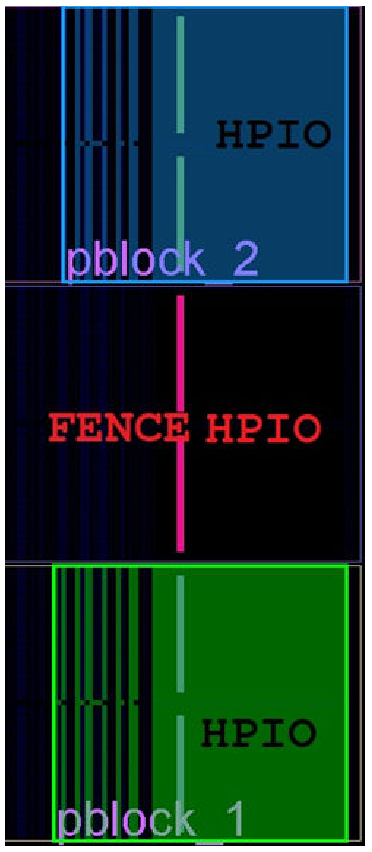
*Figure 46:* **HPIO PU with All the Sub-Components**



***Note***: Individual tiles of HPIO such as XiPHY and CMT cannot be used as fence, whenever needed the entire HPIO PU must be used for fence.

As with any other PUs, for IDF design, two HPIO PUs belonging to two different isolated Pblocks cannot be adjacent to each other. They will need a least one unprogrammed HPIO PU (not included in any Pblock) as fence between them. The following figure shows a valid fence between two HPIOs belonging to two different isolated Pblocks.

*Figure 47:* **Green and Blue Isolated Pblocks Separated by an HPIO PU Fence**



***Note***: While it is understood that the user will incur a huge penalty while using HPIO PU as fence by not being able to use 52 IO pins, this is needed to achieve isolation as the CMT tile of HPIO spans the complete PU and it shares interconnect with all the tiles in the HPIO PU.

## Guidelines for Controlling Global Clocking Logic

It needs to be ensured that no global logic is logically contained in modules that are to be isolated. This is due to the UltraScale+ architecture that prohibits the isolation of such components due to their large scope. To aid non-isolation of global logic, Vivado tools allow you to keep your global logic as is (without HDL modification) by giving you the option to target said logic with an attribute that exempts it from isolation. This is a key feature in Vivado tools, because there are many IPs that have global logic embedded in them (BUFGs and MMCMs or Debug Hubs for example). In such cases, you identify them with a search and add the HD.ISOLATED_EXEMPT attribute to them.

An example of exempting Global Clocks from Isolation:

```
set_property HD.ISOLATED_EXEMPT true [get_cells -hierarchical -filter
{PRIMITIVE_TYPE =~ CLK.BUFFER.*}]
```

***Note***: The filtering structure (that is, CLK.BUFFER.*) shown in the example is correct for Vivado Design Suite 2019.1, however, this could change in future versions. Verify the correct structure for the version of Vivado used for your design.

These filter statements are very generic (wild cards). More specific operations can be performed if there are some clocks that need exemption and some that do not.

***Note*:** You do not apply the HD.ISOLATED_EXEMPT property on a net, you apply it to the cell that drives the net to exempt the net from isolation.

**IMPORTANT!** *You cannot isolate clocks due to their global scope.*

## Vivado IP Integrator Differences – Creating Wrappers

Vivado IP integrator is fully compatible with the Isolation Design Flow. In some cases, it even makes it easier, for example, the ability to add hierarchy with a single click/command.

However, there is one feature to be mindful of when creating isolated designs in Vivado IP integrator. To preserve the IP being instantiated, IP integrator automatically adds the DONT_TOUCH property to all instances of IP. This is a good thing, because it adds the IP vendors' IP without modification. However, this impacts the feature of IDF where isolated nets can drive multiple isolated regions. In this case, the Vivado tool attempts to split that net to follow IDF rules but is prevented from doing so due to the DONT_TOUCH attribute. Thus, if an IP is to be marked as an isolated module, that IP must be wrapped with a level of hierarchy that does not have the DONT_TOUCH attribute. In such cases, follow the *Vivado Design Suite User Guide: Implementation* (UG904) to add a level of hierarchy to a module (a right-click operation).

## Gaps in the Floorplan

There are apparent gaps in the floorplan as seen by the Vivado tool Device window's Routing Resources view (refer to the following three (3) figures). These gaps are an artifact of how the software model is depicted in the Vivado IDE. Such gaps do not represent any form of isolation and cannot be used as a fence because they are not, in fact a PU/tile at all.

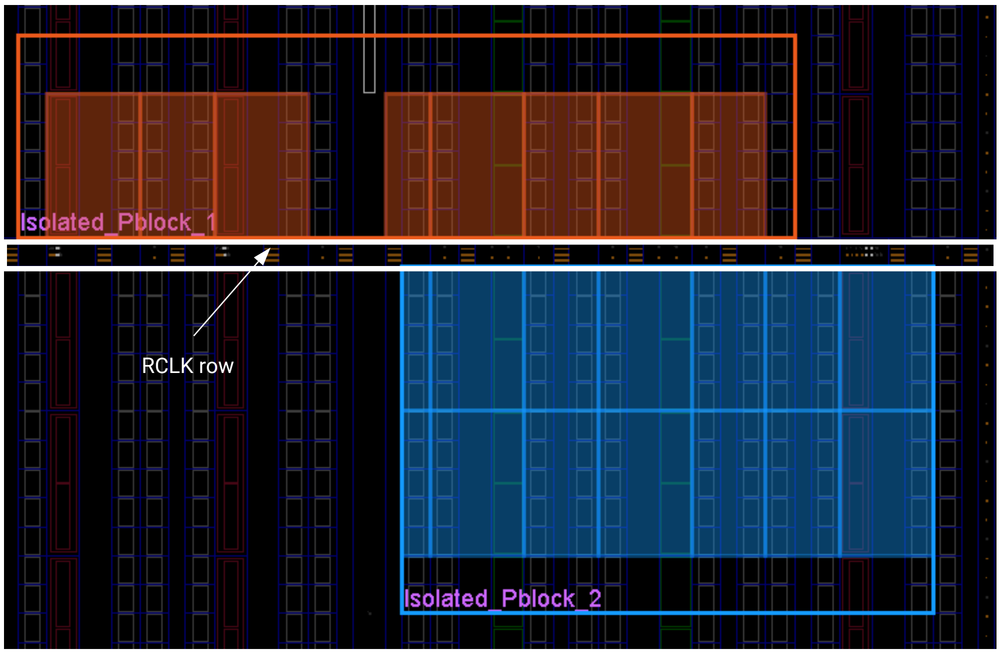
The following are the most common gaps seen in the floorplan:

- RCLK row (regional clock row)
- Configuration blocks near the center of the device floorplan
- Boundary of clock regions

### RCLK Row Gaps

The RCLK row, in the center of each clock region, appears as a gap in the FPGA. The following figure shows such a gap highlighted by a white rectangle. Do not mistake this gap as a valid fence.

*Figure 48:* **RCLK Row Highlighted by White Rectangle That Looks like Fence, but It Is Not**

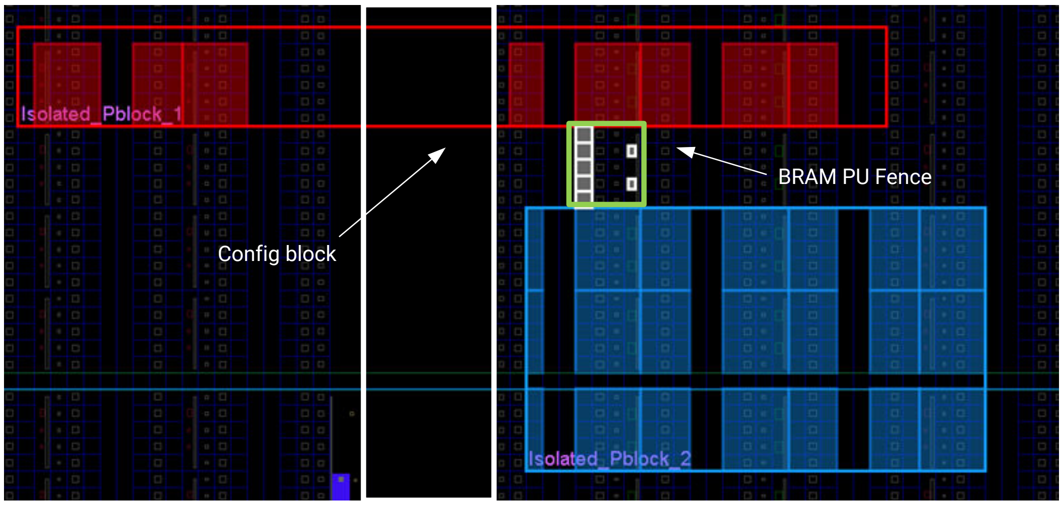


## Configuration Blocks Gap

There are some configuration blocks in the FPGA that looks like gaps but are not a fence. The following figure highlights such a gap with a white rectangle. These blocks cannot be used as a fence. However, a valid BRAM PU fence has been shown with green rectangle.

***Note***: The following figure shows the routing view in Vivado. It is highly recommended to always use device view in Vivado to look at fences and create Pblocks while using IDF. The routing view is good for determining PUs if needed.

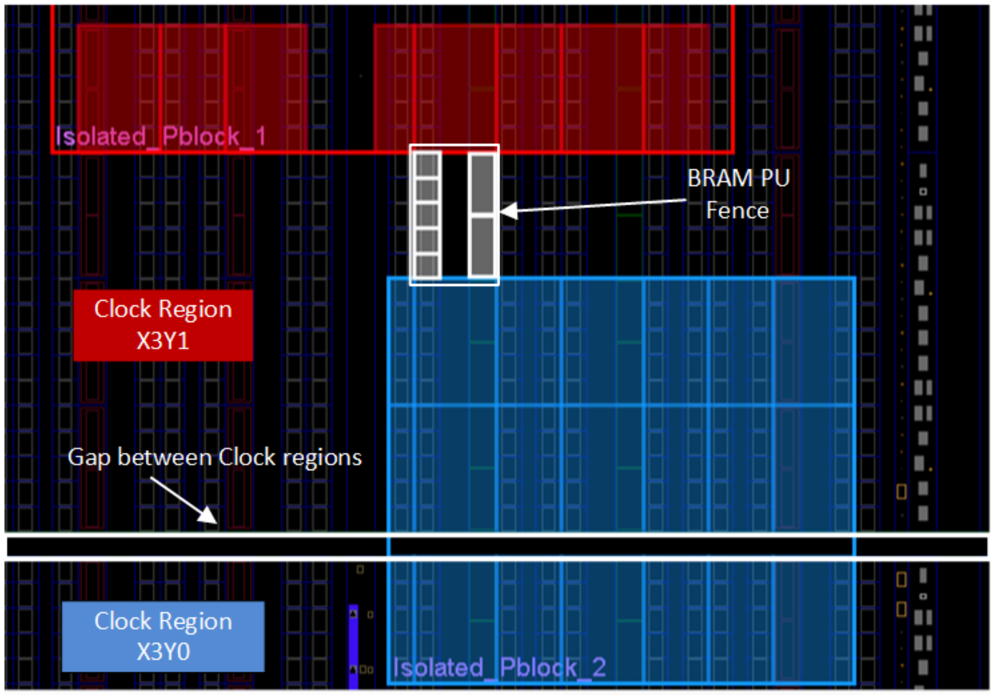
*Figure 49:* **Config Block Highlighted in White Rectangle Looks like a Fence, but It Is Not**



## Boundary of Clock Regions

Gaps can be seen at the boundary of the clock regions. These are not valid fences and do not provide any isolation. The following figure shows such a gap as that is seen in Vivado GUI.

*Figure 50:* **Gap between Two Clock Regions Highlighted in White Rectangle That Looks like a Fence, but It Is Not**



## Automatic Movement of IOB into Isolated Hierarchy

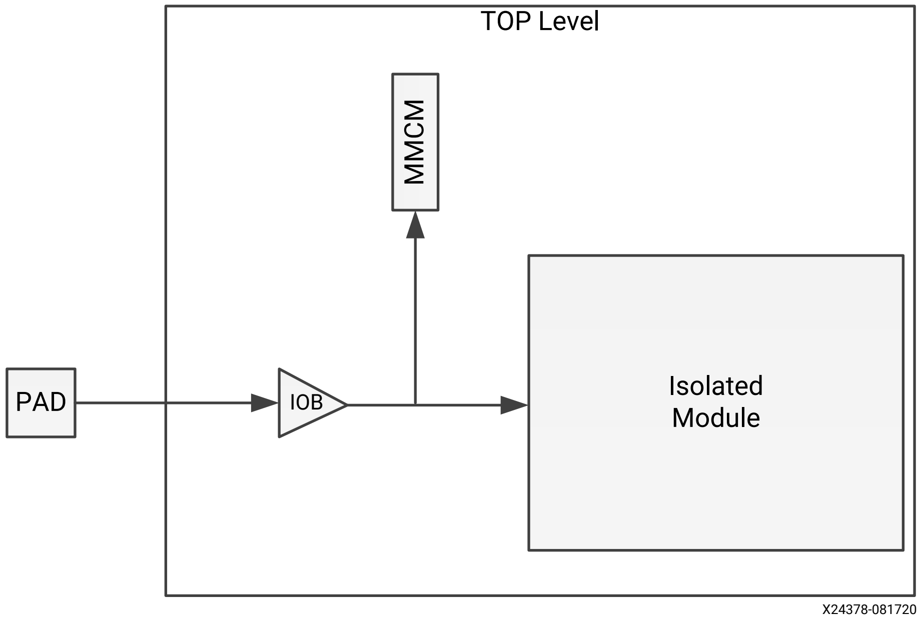
The automatic insertion of IOBs into the intended isolated hierarchy is a feature of the Vivado Design Suite. Before Vivado tools, you had to manually instantiate IOBs inside of the HDL for each isolated module. Vivado tools, however, do this for you provided you have not already instantiated the IOB themselves. At initial inference, all IOB are at top level. Wherever Vivado IDF can see a clear usage for the IOB, it automatically moves this IOB from the top level into the desired isolated module. This movement happens in the Vivado synthesis and/or the opt_design step after synthesis but before placement.

However, there are some limitations to this feature because not all possible cases allow for such movement. These cases are as follows:

### Case 1

Input with multiple destinations (that includes the top level). See the following figure.

*Figure 51:* **Case 1 Block**

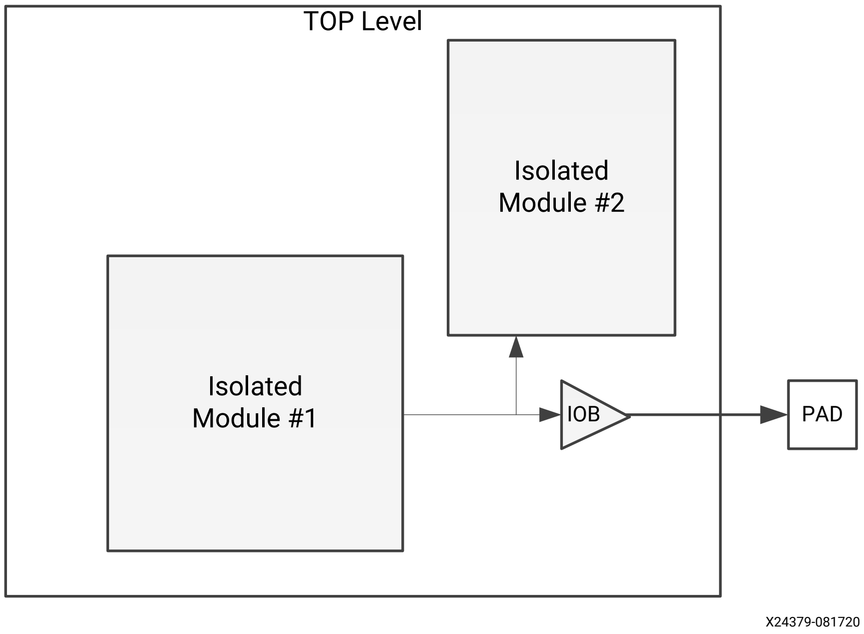


In this case, a "reset" input to the top level MMCM also connects to at least on isolated region.

### Case 2

Output whose input drives the IOB in question and at least one other region. See the following figure.

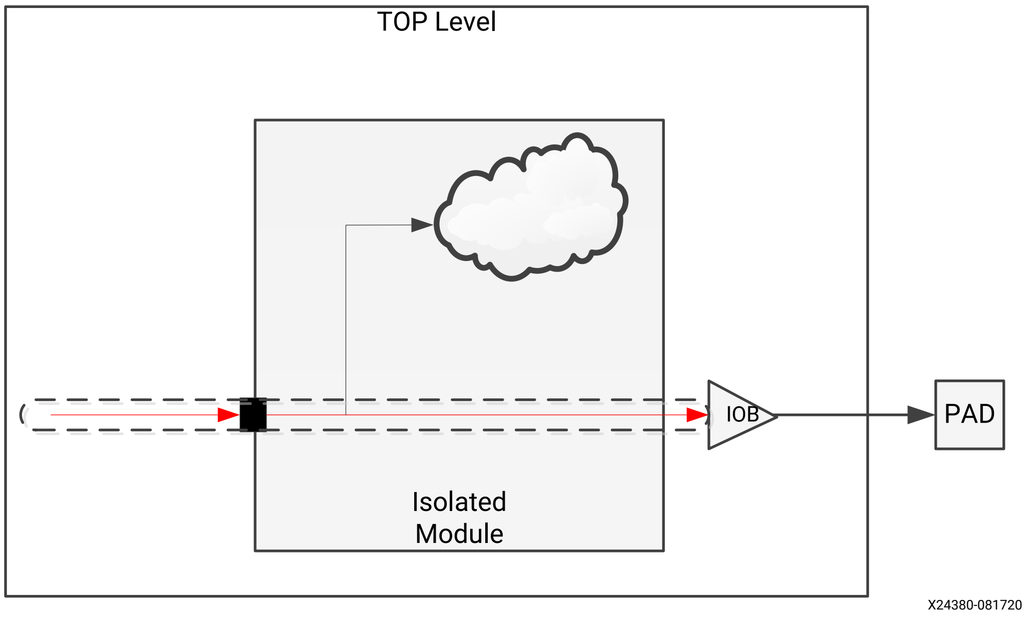
*Figure 52:* **Case 2 Block**



### Case 3

Output whose input comes directly from an input port of the isolated module in question. See the following figure.

*Figure 53:* **Case 3 Block**

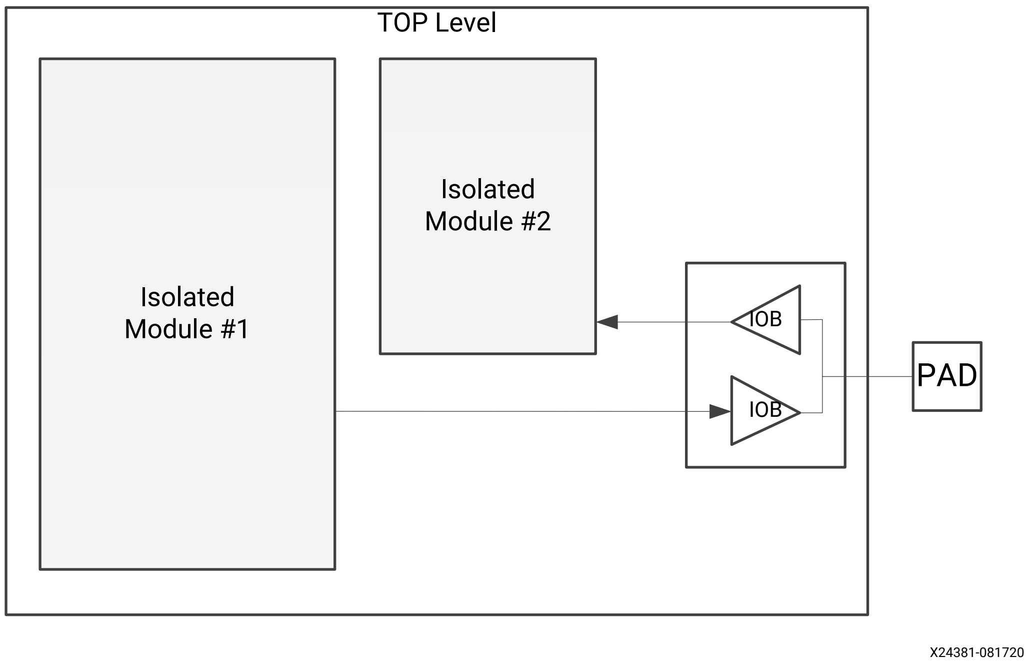


***Note***: It does not matter if that signal is used in the isolated module in question or not.

### Case 4

Bidirectional IOB where input and output do not go to the same isolated region. See the following figure.

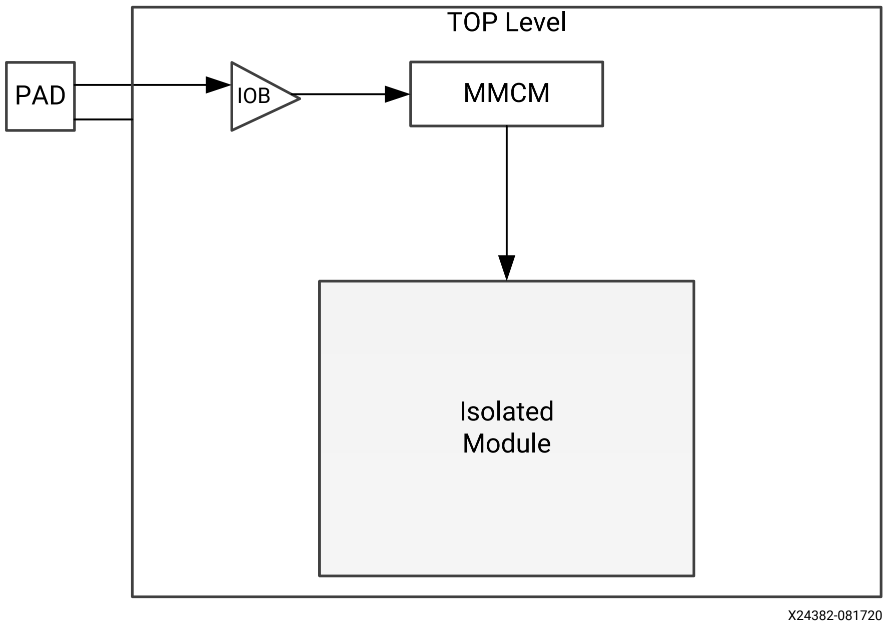
*Figure 54:* **Case 4 Block**



### Case 5

IOB with connections only to top level logic. See the following figure.

*Figure 55:* **Case 5 Block**



***Note***: Top level IOB driving only top-level logic will not be moved

### Case 6

IOB was manually instantiated by the user on their IP.

Directly instantiated IOB with the DONT_TOUCH property set on the buffer will not be moved. The DONT_TOUCH property can be inherited from the IP that instantiates the IOB or you can manually add the property.

# IDF Rules Checklist

The following checklist is a helpful set of items which you can use to verify that all the basic rules and guidelines for an IDF design are followed. The checklist can be used before, during, or after project design to help ensure the rules have been followed and to determine if overall good IDF design practices have been considered.

**IDF Checklist**

- The HD.ISOLATED property is applied to all isolated modules (except global clocking modules).
- All isolated modules have corresponding Pblocks associated with them.
- Pblocks must contain the required resource types in enough quantities to implement the intended function or functions.
- Map out Pblocks that need connectivity and those that need IOBs for better floorplanning strategy.
- Each isolated module must be in its own level of hierarchy.
    - Isolated modules cannot be nested
- A valid isolation fence (unused PU) must exist between Pblocks.
    - The fence is constructed indirectly by the space between isolated regions / Pblocks.
    - Refer to Table 1 for creating a valid fence.
    - Fence size is determined through schematic analysis and stated in rules and guidelines throughout this document.
    - Use the minimum fence width/height to maximize routing resources.
    - Bigger fence is not better. A bigger fence will not increase fault immunity but instead might impact timing due to a reduction in the availability of trusted routes.
- Isolated modules that communicate with each other must share a coincident Pblock edge separated by a valid fence otherwise routing might fail.
- Pblocks must enclose all available tiles in the device except the ones needs for minimum fence size (only the fence remains).
    - Do not leave large gaps because the gaps do not provide an advantage and can even result in routing difficulty.
    - Making the fence larger than necessary can make it difficult or impossible for inter-region signals (trusted routes) to cross the fence.
        - For example, IDF rules prevent routing touchdowns in the fence. A fence size of one PU (the minimum fence size allowed) prohibits the use of all routes (to cross the fence) that span only a single PU. Larger fences further limit the routing resources available.

- Only global clocking logic should not be isolated. All other logic should be associated with an isolated module and placed in the associated isolated Pblock.
- For global logic instantiated within an isolated module, the HD.ISOLATED_EXEMPT property must be set (TRUE) for the tools to route to or from that global logic.
- Feed-through signals are not allowed without buffering of some kind (LUT, FF, etc.).
  - Vivado does this automatically unless you choose to disable this feature
- Ports communicating between two isolated modules can only have one source and one destination.
  - Vivado does this automatically unless you choose to disable this feature
- IOBs must be instantiated or inferred inside isolated modules for proper isolation of the IOB.
  - Vivado does this automatically unless you choose to disable this feature.
- These are the limitations on Vivado automatically inserting IOBs
  - Input with multiple destinations, including top-level (i.e., reset to MMCM and isolated module)
  - Output whose input drives the IOB in question and at least one other region, including top-level
  - Output whose input comes directly from an input port of the isolated module in question
  - Bidirectional IOB where input and output do not go to the same isolated region
  - IOB with connections only to top level. Top level IOB driving only top-level logic will not be moved.
  - IOB was manually instantiated by you or the IP.
    - Directly instantiated IOB with the DONT_TOUCH property set on the buffer will not be moved. The DONT_TOUCH property can be inherited by the IP that instantiates the IOB or you can add it directly.
- Enable VIV by executing the Tcl command `set_param hd.enableIDFDRC true` in Vivado's Tcl console.
- Run Xilinx Vivado Isolation Verifier (VIV) on the elaborated design (constraint checking) to check for DRC violations for device and package pins (pin adjacency) and for IOB bank violations (if required).
- Run Xilinx VIV on the placed and routed (implemented) design to check for DRC violations for Isolation violations.

## Isolation Design Example

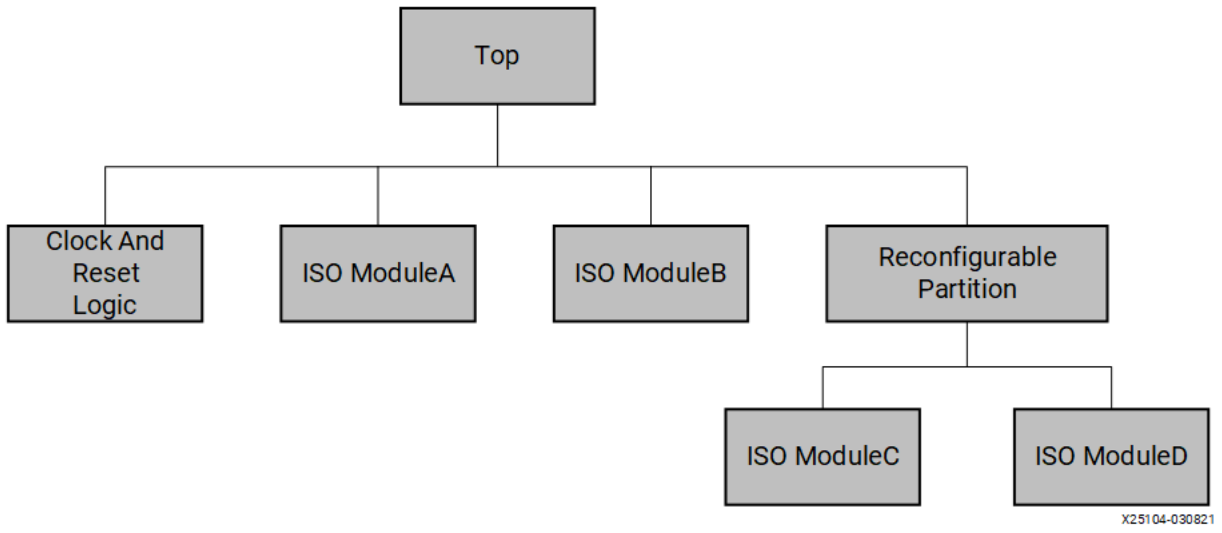
An isolation design example is provided in the *Isolation Design Example for Zynq Ultrascale+ MPSoC Application Note* (XAPP1336) and is used throughout this application note to describe the design details and tool flow. For detailed information, see Isolation Design Flow.

# IDF+DFX

## Overview of IDF+DFX

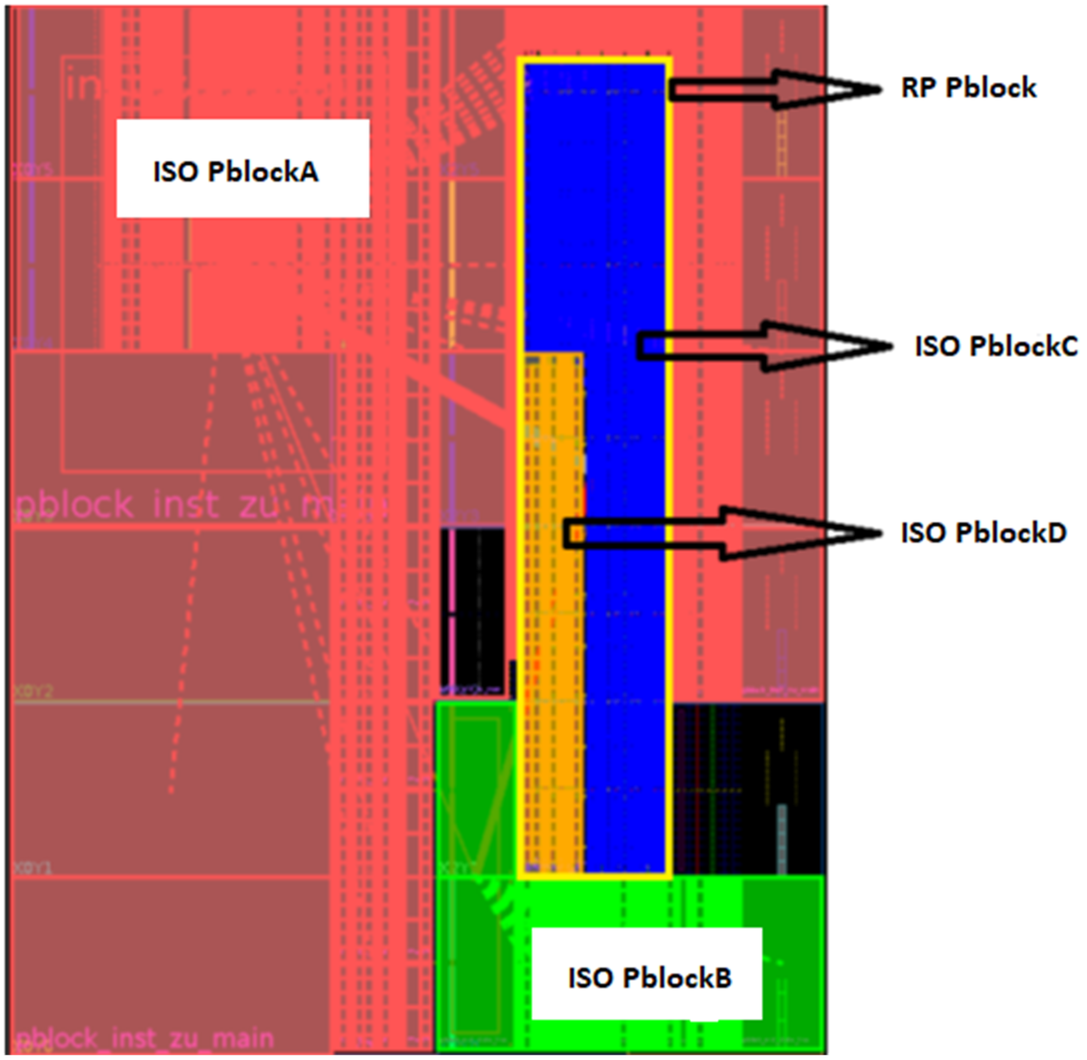
The Isolation Design Flow (IDF) and Dynamic Function eXchange (DFX) are two production solutions from Xilinx. They have been available for Zynq UltraScale+ devices from Vivado 2018.3 onwards. From Vivado 2020.2 onwards, Xilinx supports combined flow of IDF and DFX. The document is written with the assumption that the reader is familiar with both IDF and DFX methodologies. For details on these individual methodologies, the user may refer to Isolation Design Flow in this application note, *Vivado Isolation Verifier User Guide* (UG1291), *Isolation Design Example for Zynq Ultrascale+ MPSoC Application Note* (XAPP1336), *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909), and *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947). With this combined support, the user can create nested isolated modules (IM) inside reconfigurable partition (RP). Sample design hierarchy structure looks like the following example provided here.

*Figure 56:* **IDF+DFX Design Hierarchy**



These isolated and reconfigurable regions require a floorplan that defines the overall usage of the device. An example floorplan might look like this:

*Figure 57:* **Example Floorplan for IDF+DFX Design**



The reconfigurable partition can have one or more reconfigurable modules that will be dynamically exchanged by way of partial bitstream delivery, most likely via the processing system (PS). Users must develop their own solution for delivering partial bitstreams and handling any runtime tasks (logical decoupling of the dynamic region, updates to drivers associated with reconfigured peripherals).

Xilinx supports isolation design flow (IDF) with one or more reconfigurable partitions (RP). Users can have one or more isolated modules (IM) inside the RP region or outside the RP region (in Static region). The support for change of composition (hierarchy, floorplan) of the nested IMs inside the RP from one reconfigurable module (RM) to the next is limited. It is strongly recommended to keep the same floorplan for each configuration. The complete support for this will be added in future Vivado releases.

Use cases that are NOT supported include:

- Nesting an isolated region within an isolated region
- Nesting a reconfigurable partition within an isolated region
- Nesting of a reconfigurable partition within a reconfigurable partition and then an isolated partition under that.
- Identifying a single module as both reconfigurable and isolated

This last exception can be effectively supported by directly instantiating a level of hierarchy such that an isolated module is directly below a reconfigurable partition and their floorplans are identical.

## Enable IDF+DFX Flow

A combined flow of IDF+DFX is enabled by default for all Zynq UltraScale+ MPSoC devices but the user need to set a *param* to enable the appropriate set of design rule checks for IDF+DFX. Users need to enable IDF and DFX separately as mentioned in the Isolation Design Flow of this document and in the *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909). To enable IDF+DFX DRCs, set the following parameter.

```
set_param hd.enableIDFDRC 1
```

This parameter information is not stored in the database so must be re-enabled in each Vivado session. The parameter is most easily set in your `vivado_init.tcl`. For more details on how to run VIV DRCs, refer to Vivado IDF Verifier (VIV) Checks 1,2,3, and 4 and Vivado Isolation Verifier (VIV) Checks 5 and 6 of this document.

Both the project mode and the non-project Tcl scripted mode are supported. The DFX flow for IP Integrator, however, is still in early access. You may refer to the Isolation Design Flow of this document and *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909) for flow requirements and details for topics such as:

- The use of HD.ISOLATED and HD.RECONFIGURABLE to define the functionality of a given level of hierarchy
- Fence creation guidelines and rules for IO and clocks for the isolation design flow
- Logical decoupling and partial bitstream delivery for the dynamic function eXchange flow

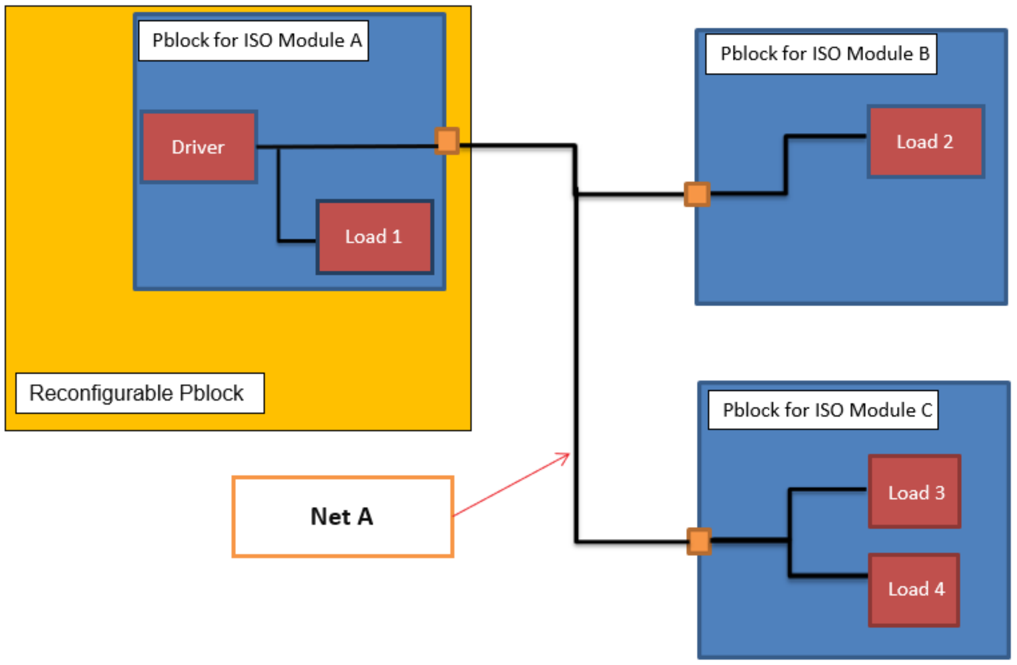
Routing expansion in IDF+DFX flow works same as DFX flow. The reconfigurable partition in the combined IDF+DFX solution is subject to more stringent regulations given the isolated modules that surround it. These additional rules must be considered when floorplanning the reconfigurable partition, and when designing each RM that will be placed in that RP:

- Expanded routing for RPs is enabled, but routing is contained per isolated module to follow IDF rules
- No embedded clocks are allowed within an isolated module within any reconfigurable module
- Each interface on the reconfigurable partition must connect to only one isolated module outside the RP*

***Note***: *Features planned for the future Vivado releases.

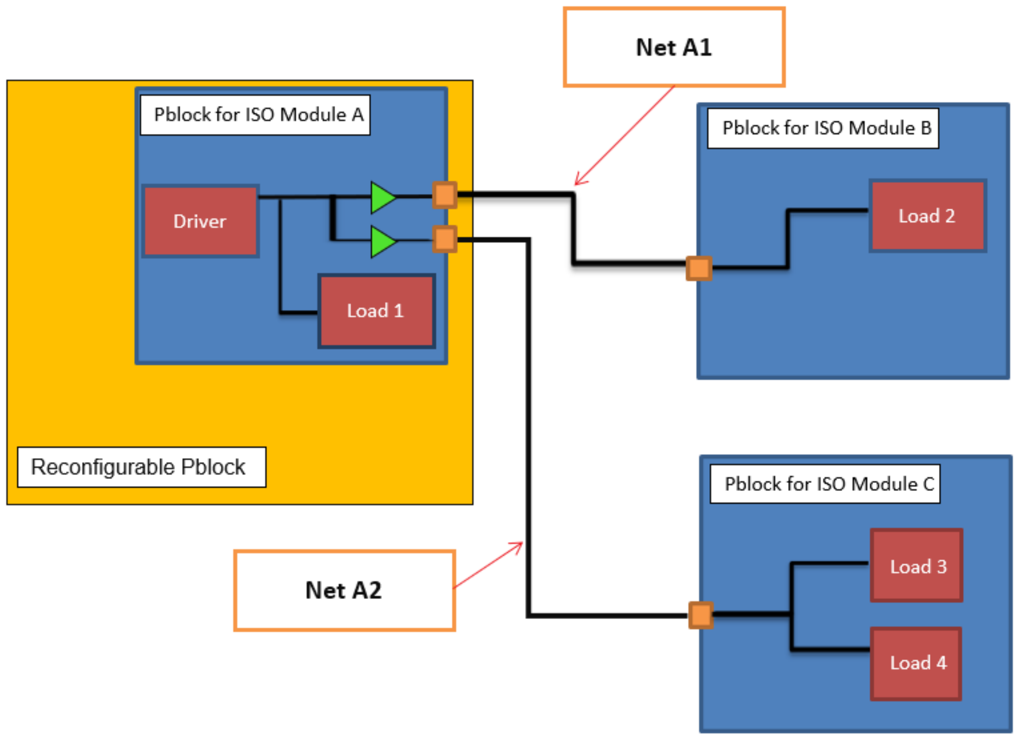
For this last item, the current requirement is that a trusted route starts in one IM and connects to only one other IM; no net splitting is supported. If a trusted route must go to more than one IM, the user must create additional drivers and ports that will each connect to a single IM. The following image shows one unsupported scenario in Vivado 2020.2. A similar scenario of a driver in a static IM driving loads in multiple IMs, at least one of which is within an reconfigurable partition (RP), would also be unsupported.

*Figure 58:* **Unsupported Connectivity That Requires Net Splitting**



While net splitting is not currently supported by the tools, it can be done manually. The support for automatic net splitting will be added in future Vivado releases.

*Figure 59:* **Workaround Using Replication of Drivers**



## Floorplan Rules and Guidance

Both isolated modules and reconfigurable partitions require Pblocks to identify resources to contain these respective parts of the design. You may follow requirements for the respective module type as documented in each user guide. The combined IDF+DFX solution adds more design rule checks (DRCs); run these interactively as you build the floorplan and watch for critical warnings as the implementation flow is run.

The most fundamental rule is that the Pblock ranges for any isolated module within a reconfigurable partition be completely contained within the RP Pblock. Standard fence rules apply regardless of where the isolated module Pblock exists. The fence can be either inside or outside of the RP Pblock.

This collection of RMs for the one reconfigurable partition in the design may have different internal floorplans for the isolated regions contained within, but the interface port list must remain consistent, per DFX rules. The two RMs shown in the following figures are legal and no extra guidance is needed to account for the different sizes of isolated modules within RM1 and RM2.

*Figure 60:* **Different Isolation Pblocks within the Reconfigurable Partition**

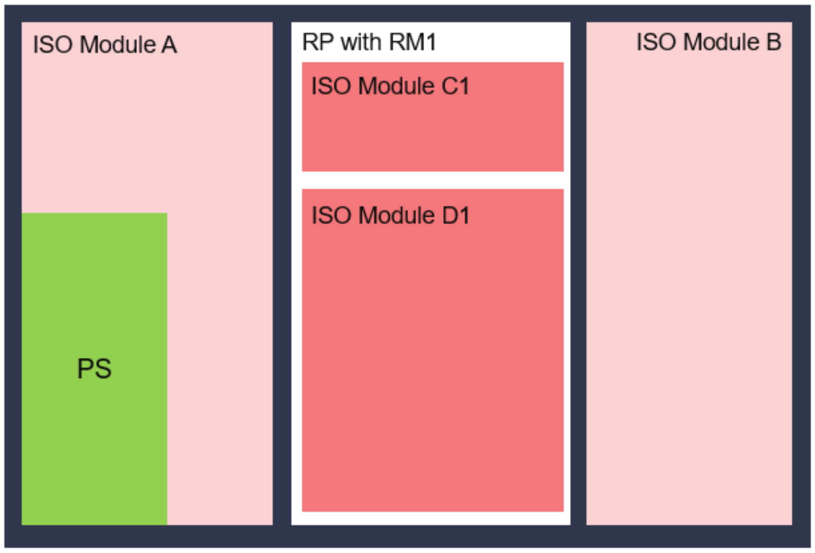
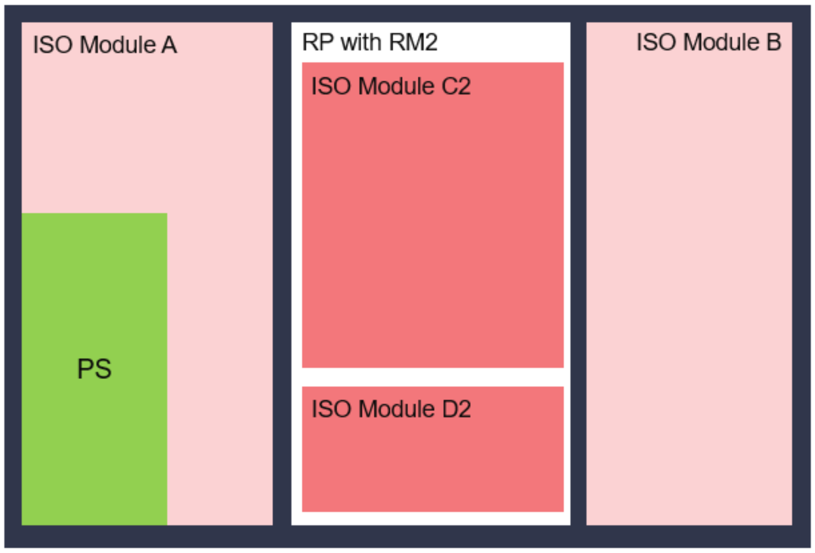


*Figure 61:* **Different Isolation Pblocks within the Reconfigurable Partition**



Nested Pblocks inside the isolated Pblocks are supported. Users can use nested Pblocks to meet timing closures. However, nested IDF Pblocks (isolated Pblock inside another isolated Pblock), nested DFX (DFX Pblock inside another DFX Pblock) and DFX Pblock inside the isolated Pblock, are not supported. The nested Pblocks inherit the properties of the parent. For example, `IS_SOFT` is false for the nested Pblocks. The support to change the properties will be added in future Vivado releases.

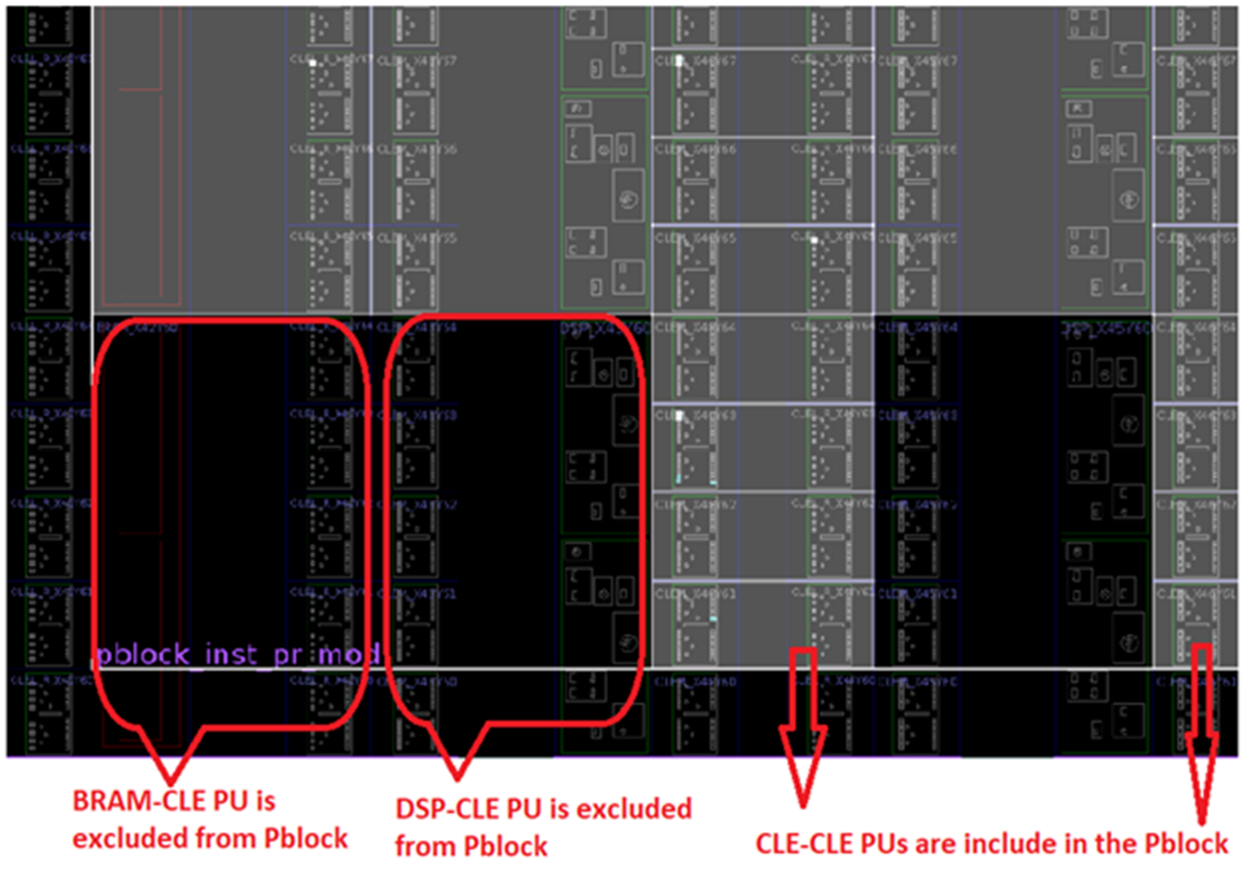
One more limitation for the nested Pblocks is that net splitting will fail if the driver or load is in the nested Pblock. The support for this will be added in future Vivado releases.

## Snapping Mode in IDF+DFX flow

Snapping mode of the Pblocks in IDF+DFX should be FINE_GRAINED whereas in individual flows the snapping modes are different. Snapping mode of Pblocks in DFX flow is ON and the snapping mode in IDF flow is FINE_GRAINED. The difference between the snapping modes ON and FINE_GRAINED are described in the following paragraph.

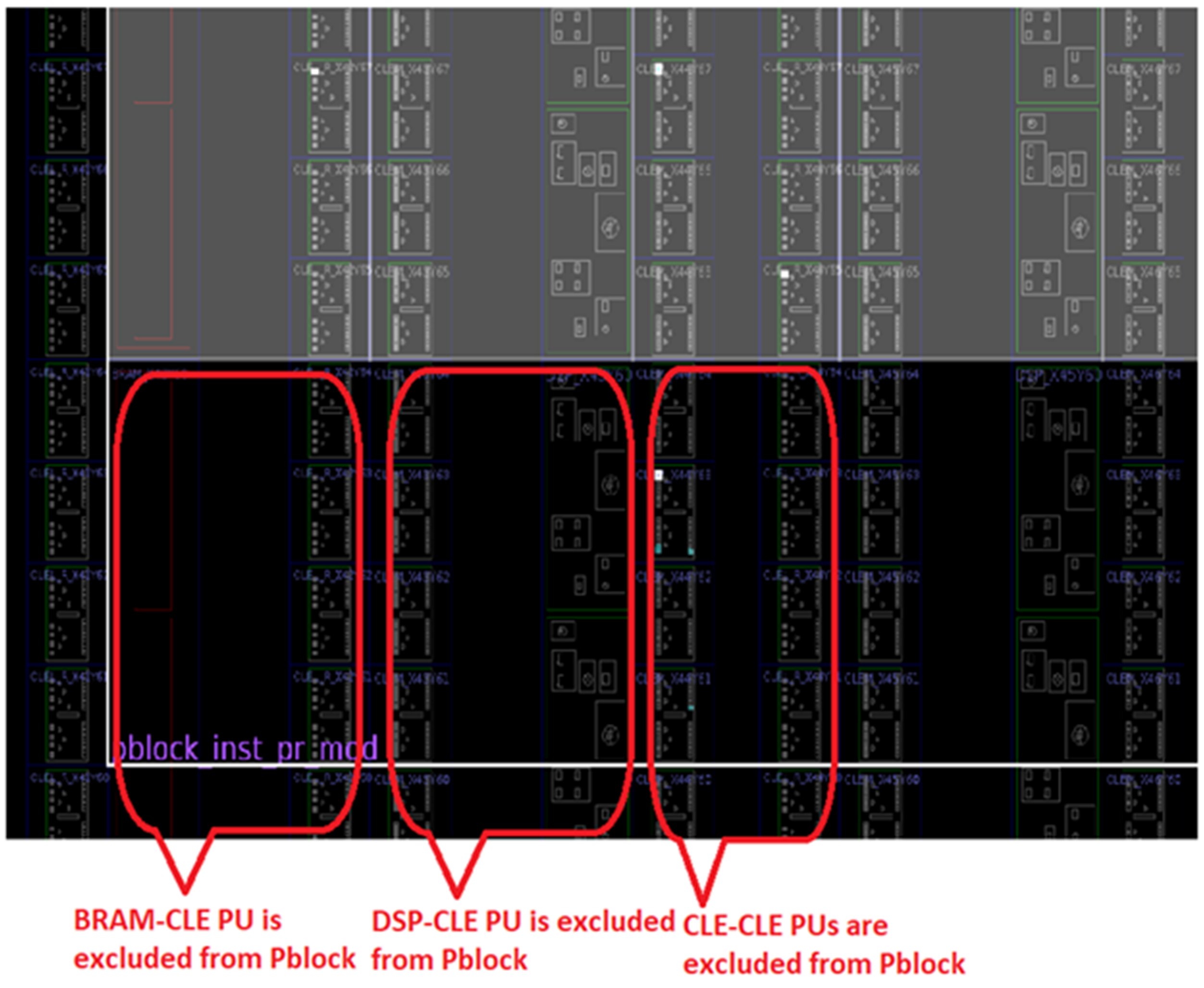
For snapping mode FINE_GRAINED, a PU is considered in derived ranges when it is completely enclosed within the Pblock. When a PU is partially included in the Pblock, then the whole PU is excluded from the Pblock's resources in the derived ranges. In the following figure, a Pblock is drawn such that only 80% of the BRAM-CLE and DSP-CLE PUs are included in the Pblock at the bottom boundary of the Pblock. With snapping mode FINE_GRAINED, the BRAM-CLE and DSP-CLE PUs are completely excluded from the Pblock, and CLE-CLE PUs which are in same row are included in the derived ranges of the Pblock. You may refer to Derived Range and Snapping Mode for detailed explanation.

*Figure 62:* **Pblock with Snapping Mode FINE_GRAINED**


pblock_inst_pr_mod
BRAM-CLE PU is excluded from Pblock
DSP-CLE PU is excluded from Pblock
CLE-CLE PUs are include in the Pblock

For snapping mode ON when a PU is partially included in the Pblock, then the whole PU is excluded from the derived ranges of the Pblock. This is the same as with the snapping mode FINE_GRAINED and additionally, CLE-CLE PUs in the same row are also excluded from the Pblock derived ranges. The following figure of a Pblock is drawn such that it displays only 80% of the BRAM-CLE and DSP-CLE PUs included in the Pblock, at the bottom boundary of the Pblock. With snapping mode ON, the BRAM-CLE and DSP-CLE PUs are completely excluded from the Pblock derived ranges and also the CLE-CLE PUs which are in same row are also excluded from the derived ranges of the Pblock. So, the Pblock boundary is always rectangle with snapping mode ON.

*Figure 63:* **Pblock with Snapping Mode ON**


pblock_inst_pr_mod
BRAM-CLE PU is excluded from Pblock
DSP-CLE PU is excluded from Pblock
CLE-CLE PUs are excluded from Pblock

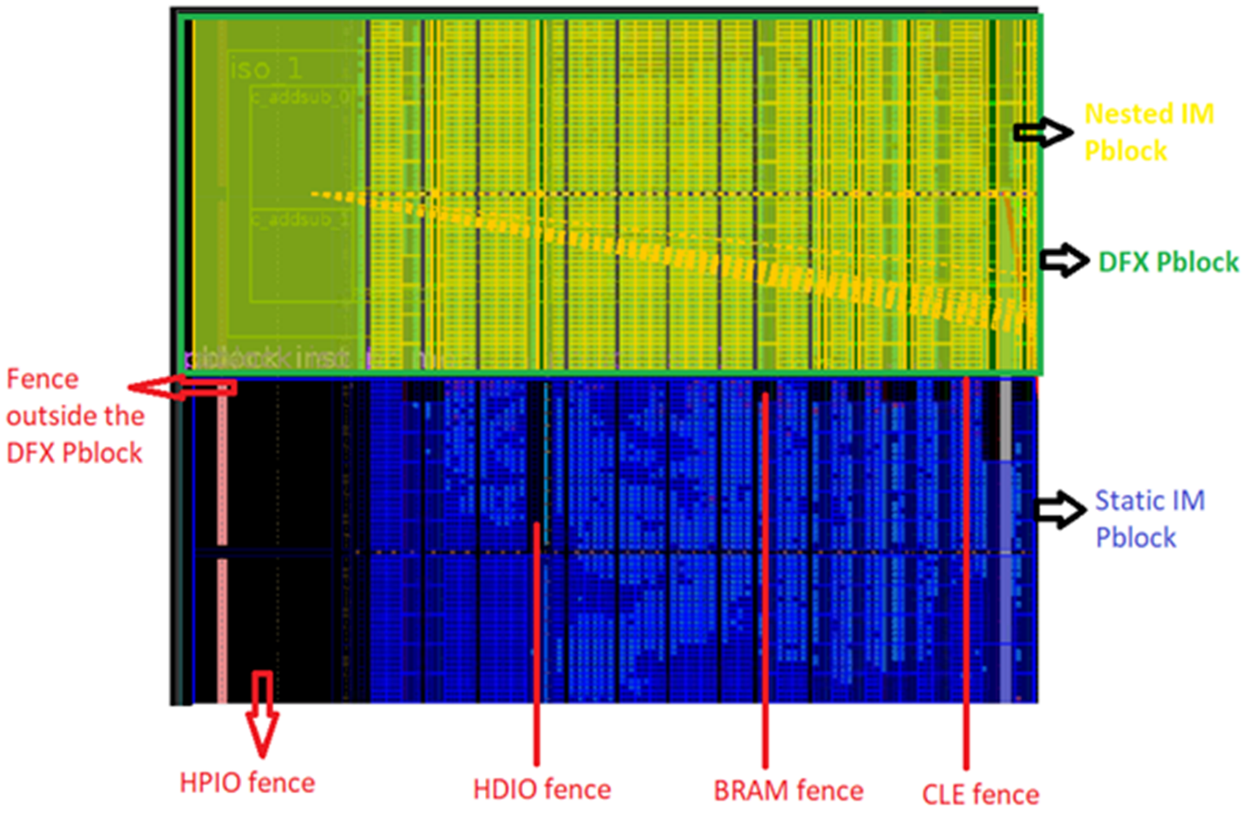
With the same boundary, Pblock with snapping mode FINE_GRAINED includes more resources into the derived ranges of the Pblock, than the Pblock with snapping mode ON.

## Fence Rules in IDF+DFX

Fence rules are the same as the normal IDF flow, you may refer to the Isolation Fence section for more details. Fence is needed only between IM Pblocks, irrespective of whether they are nested IM Pblocks or static IM Pblocks. A fence is not needed between the DFX Pblock and IM Pblock. A fence between the static IM Pblock and nested IM Pblock can be inside of the DFX Pblock or outside the DFX Pblock.

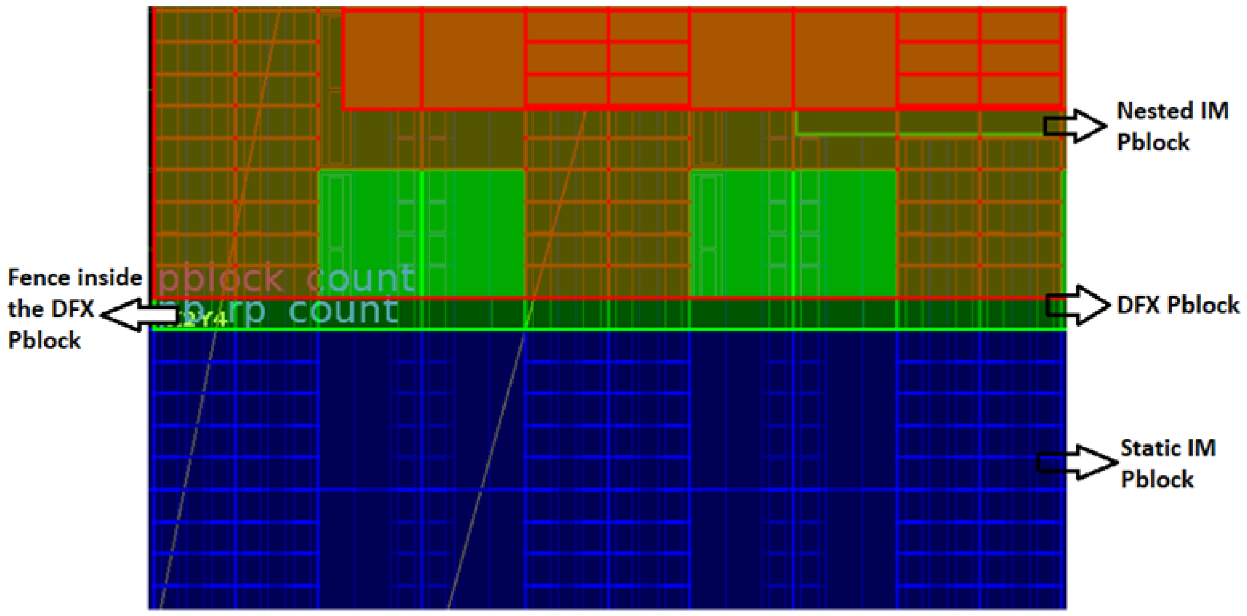
In the following figure, the DFX Pblock and the nested IM Pblock shares the same boundaries, and the fence between the nested IM Pblock and the static IM Pblock is outside of the DFX Pblock.

*Figure 64:* **Fence Outside of the DFX Pblock**



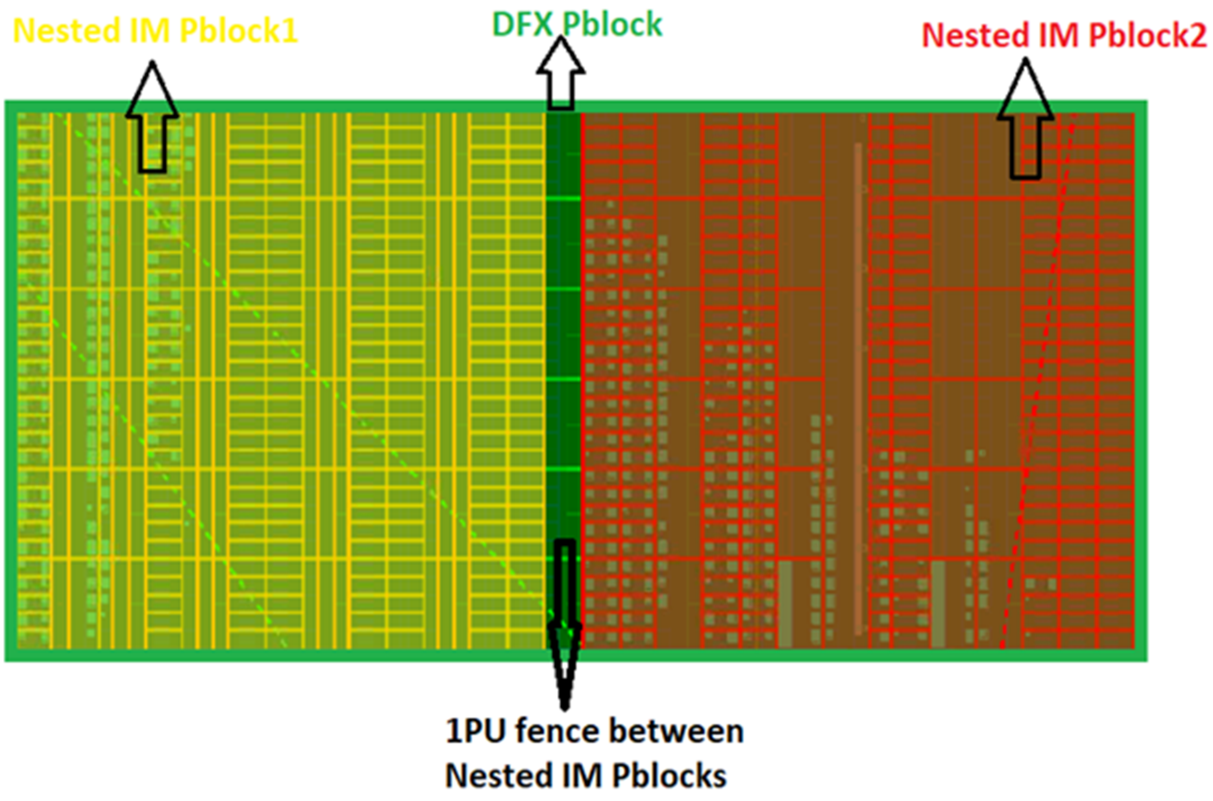
In the following figure, there is no gap between the DFX Pblock and the static IM Pblock, and the fence between the nested IM Pblock and the static Pblock is inside of the DFX Pblock.

*Figure 65:* **Fence Inside of the DFX Pblock**



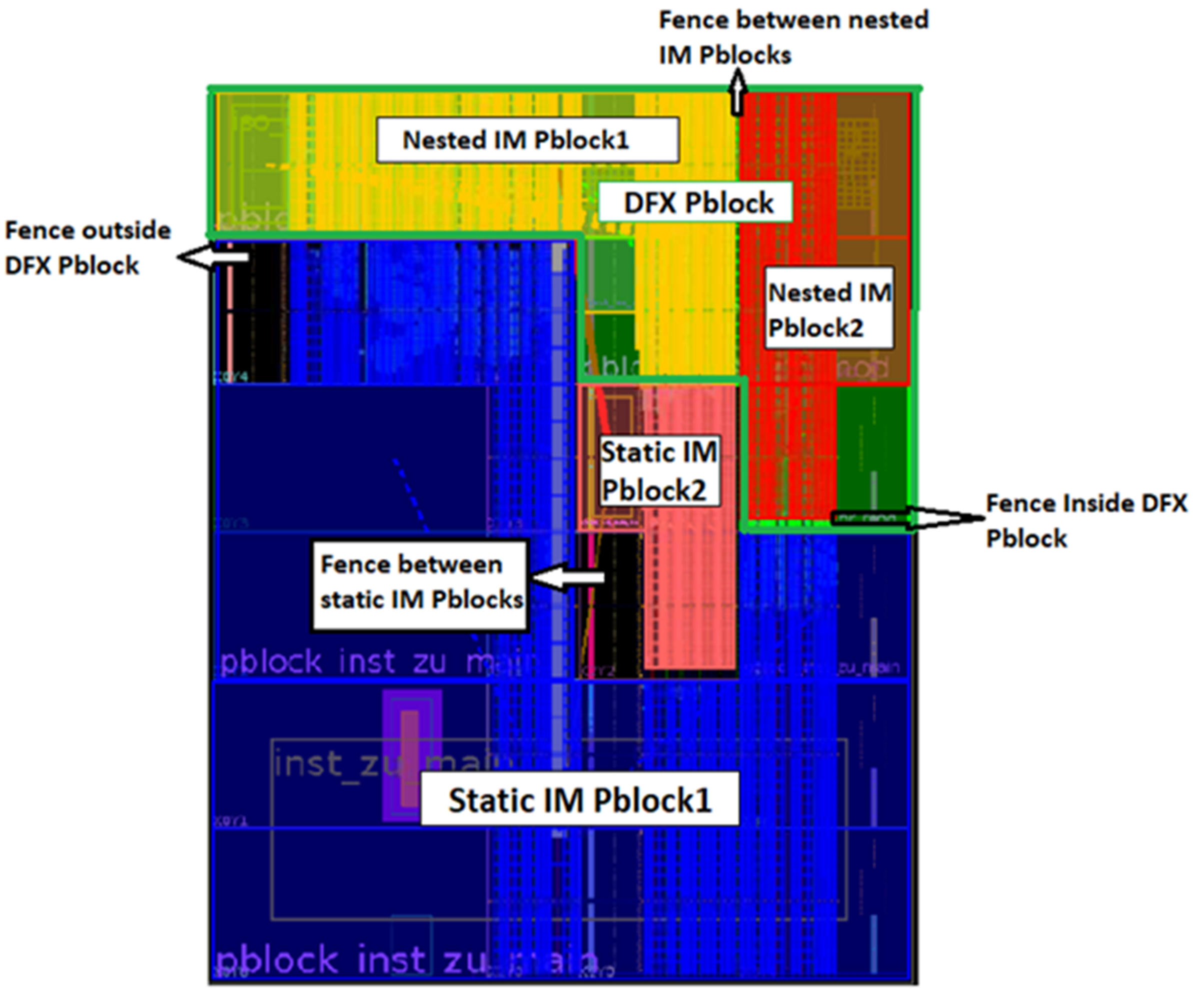
Nested IM Pblocks inside of the DFX Pblock also need a fence and follows standard IDF fence rules. The fence between the nested IM Pblocks is always inside of the DFX Pblock. In the following figure, nested IM Pblock1 and nested IM Pblock2 has an 1PU fence inside of the DFX Pblock.

*Figure 66:* **Fence Between the Nested IM Pblocks**



The following image is the complete example floorplan with two static IM Pblocks, and two nested IM Pblocks inside of a DFX Pblock.

*Figure 67:* **IDF+DFX Example Floorplan with Fences**



Similarly, the fence between the nested IM Pblocks of two DFX Pblocks can be inside either of the DFX Pblock or outside of the two DFX Pblocks.

## IDF + DFX Summary

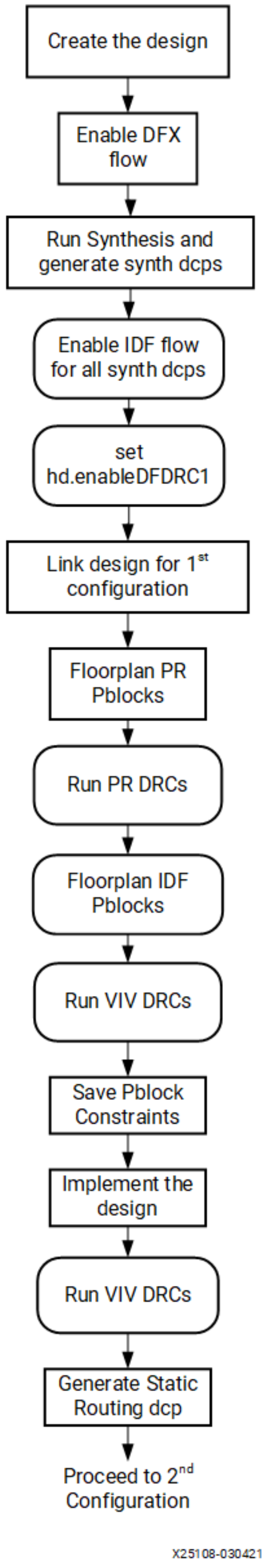
This section gives high level details on how to combine IDF and DFX flows in a single design. You may refer to *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947) for a detailed explanation on DFX design creation and *Isolation Design Example for Zynq Ultrascale+ MPSoC Application Note* (XAPP1336) for IDF design creation.

The following steps outline how to combine IDF and DFX flows in a single design.

- Follow the DFX flow by referring to *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947) till the synthesis and generate synth DCPs or static top-level design and for each DFX module.

- Enable IDF flow in all the DCPs. First, open the static dcp and set HD.ISOLATED to true on the modules and save the dcp. Similarly, you may open all the RM DCPs and set HD.ISOLATED to true and then save the DCPs.
- Enable IDF DRCs by setting hd.enableIDFDRC to true.
- Link the entire design for the first configuration using the link_design command. Some of the IDF optimizations related to net-splitting and IO buffer insertion takes place in the link_design phase. Ensure the IDF is enables on the synth DCPs before the link_design.
- Next, floorplan the design. The best way to do is step-by-step floorplanning. First, do the floorplanning for reconfigurable partitions. Follow the guidelines from *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909) and *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947) and draw the Pblocks for reconfigurable partitions. Run the Dynamic Function eXchange DRCs and ensure there are no DRC errors.
- Draw Pblocks for isolated modules including nested modules inside of the PR by following the IDF and IDF+DFX floorplanning guidelines mentioned in this document. You may refer to Drawing Pblocks Using Vivado GUI section for drawing Pblocks using Vivado GUI.
- Run VIV DRCs and ensure there are no warnings or errors. You may refer to *Vivado Isolation Verifier User Guide* (UG1291) on how to run the VIV DRCs.
- Save these Pblocks and associated properties. A script is provided with *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947) design files to save the Pblocks. It is advised to save static Pblocks and RP Pblocks in a single XDC file and nested IM Pblocks inside each RM in a separate file. You may use the script with **Pblocks option**.
- Implement the design using the standard DFX implementation flow.
- Run VIV DRCs on implemented design. You may refer to *Vivado Isolation Verifier User Guide* (UG1291) on how to run VIV DRCs. Ensure there are no warnings or errors.
- Generate black box dcp and static routing dcp by following the steps from *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947).

*Figure 68:* **IDF+DFX Design Flow Steps**

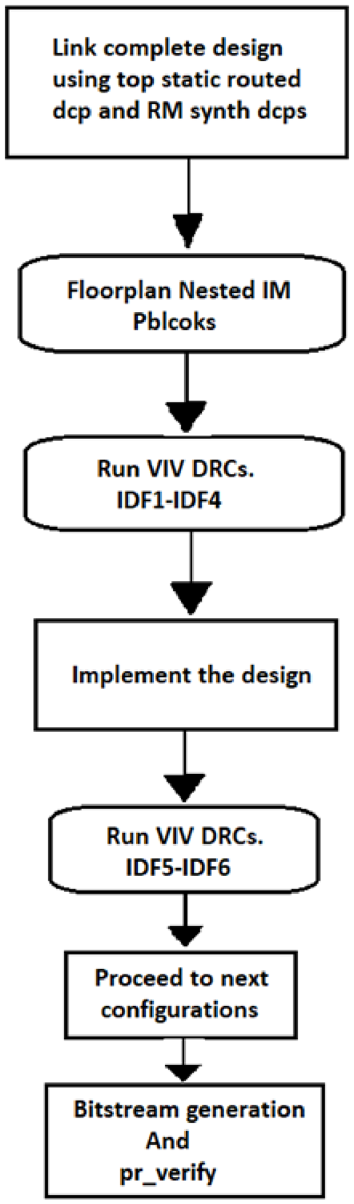


## Second Configuration

- Link the complete design for second configuration using the top static route dcp and RM synth dcps. You may refer to *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947) for further details.
- The static route dcp does not contain floorplanning information for nested IMs inside of the RP. It contains Pblock information for static IMs and PR region only.
- If there is no change in the hierarchy and floorplan inside the PR from the previous configuration, then use the already saved XDC file and load the floorplan for nested IMs. If there is a change in the hierarchy or floorplan, then create new floorplanning by following IDF and IDF+DFX guidelines. Refer to the Drawing Pblocks Using Vivado GUI section for drawing Pblocks using the Vivado GUI.
- Run VIV DRC checks 1, 2, 3, and 4. Refer to *Vivado Isolation Verifier User Guide* (UG1291) on how to select and run VIV DRCs. Do not run VIV implementation DRCs (check 5 and 6) at this point as the design is not implemented yet. Ensure there are no DRC warnings or error.
- Save the Pblocks for nested IMs. You can use the script provided with *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947).
- Implement the design by using the standard DFX implementation flow.
- Run all VIV DRC checks on the implemented design. Ensure there are no warnings or errors.

You may similarly implement all the configurations. For pr_verify, bitstream generation, and for loading the FPGA, follow the standard DFX flow.

*Figure 69:* **IDF+DFX Design Flow Steps Second Configuration**


Link complete design using top static routed dcp and RM synth dcps
Floorplan Nested IM Pblcoks
Run VIV DRCs. IDF1-IDF4
Implement the design
Run VIV DRCs. IDF5-IDF6
Proceed to next configurations
Bitstream generation And pr_verify

### Supported/Unsupported Features

This section lists the current 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 Zynq UltraScale+ MPSoC
- One or more reconfigurable partitions alongside of one or more isolated regions

### Unsupported Features

Some features are not yet implemented but could be planned for future releases, where noted.

- UltraScale™ and Versal™ architectures, Virtex®, and Kintex® UltraScale+ devices.
- Zynq UltraScale+ RFSoC devices
- Nesting a reconfigurable partition within an isolated module
- Net splitting, as documented above

### Known Limitations

- Clocks cannot be isolated from the other clocks.

## Conclusion

The focus of this document is to help design engineering professionals understand the new features and limitations of implementing the Xilinx® Isolation Design Flow (IDF) technology with UltraScale+™ architecture and Zynq® UltraScale+™ MPSoC programmable logic (PL). It helps implement the IDF technology as part of your overall layers of protection strategy used in functional safety and security design architectures. Although the impact of using IDF is not discussed in this document, you are encouraged to discuss such solutions with your Xilinx Field Application Engineer or other Xilinx resources. This document also discusses about IDF+DFX design flow. The Isolation Design Flow (IDF) and Dynamic Function eXchange (DFX) are two production solutions from Xilinx.

## References

1. *UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs* (UG949)
2. *Vivado Isolation Verifier User Guide* (UG1291)
3. *UltraScale Architecture Configuration User Guide* (UG570)
4. *Vivado Design Suite User Guide: Hierarchical Design* (UG905)
5. *Soft Error Mitigation Controller LogiCORE IP Product Guide* (PG036)

6. *UltraScale Architecture Memory Resources User Guide* (UG573)
7. *Triple Modular Redundancy (TMR) LogiCORE IP Product Guide* (PG268)
8. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
9. *UltraScale Architecture Clocking Resources User Guide* (UG572)
10. *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909)
11. *Vivado Design Suite Tutorial: Dynamic Function eXchange* (UG947)
12. *Isolation Design Example for Zynq Ultrascale+ MPSoC Application Note* (XAPP1336)
13. *Calculating Zynq-7000 Failure Rates for Functional Safety Applications* (XAPP1279)
14. *Calculating Artix-7 Failure Rates for Functional Safety Applications* (XAPP1310)
15. *Calculating Spartan-7 Failure Rates for Functional Safety Applications* (XAPP1325)

***Note*:** References 11 through 13 are available from the Functional Safety lounge.

## Revision History

The following table shows the revision history for this document.

| Section | Revision Summary |
|---|---|
| **03/09/2021 Version 2.1** | |
| IDF+DFX | Added several new sections and images for more clarity on combined IDF+DFX flow. |
| **10/09/2020 Version 2.0** | |
| Isolation Properties, Isolation Modules, and Communication between Isolated Modules | Added new sections for more clarity on the HD.ISOLATED and HD.ISOLATED_EXEMPTION properties. |
| Derived Range and Snapping Mode | Added new sections for more clarity on how placement and routing resources are added in Pblocks. |
| Mapping the Logical Ownership to the Physical Ownership | Added a new section for more clarity on link between logical and physical view for IDF flow. |
| Off-Chip Communication – Input / Output Buffer Control | Added a new section for more clarity. |
| Isolation Fence Fence Around Processor Subsystem for Zynq UltraScale+ Devices and Fence for SSIT Devices | Renamed Fence to Isolation Fence and added new content to clarify few key concepts with reference to Fence. |
| Floorplanning: Drawing Pblocks Using Tcl Commands and Constraints | Added a new section for more clarity. |
| Design Guidance, Hints and Guidelines, and IDF Rules Checklist | Added new sections to provide better understanding of IDF and its rules. |

| Section | Revision Summary |
|---|---|
| • Derived Range and Snapping Mode<br>• Derived Range and Snapping Mode<br>• Mapping the Logical Ownership to the Physical Ownership<br>• Isolation Fence<br>• Fence Around Processor Subsystem for Zynq UltraScale+ Devices<br>• Fence for SSIT Devices<br>• Trusted Routing Rules<br>• Shading Pblocks<br>• HDIO versus HPIO PUs<br>• RCLK Row Gaps<br>• Configuration Blocks Gap<br>• Boundary of Clock Regions<br>• Automatic Movement of IOB into Isolated Hierarchy | Added new content to the existing sections and images for clarity and all-around enhancement to the IDF concepts in this application note. |
| **04/15/2019 Version 1.1** | |
| Summary and Conclusion | Added coverage for the Zynq UltraScale+ MPSoC |
| **02/15/2019 Version 1.0** | |
| Initial release. | N/A |

## Please Read: Important Legal Notices

### AUTOMOTIVE APPLICATIONS DISCLAIMER

### Copyright