See all versions
of this document

Vivado Design Suite Tutorial

Dynamic Function eXchange

UG947 (v2026.1) July 29, 2026

AMD

Table of Contents

Introduction **7**
Navigating Content by Design Process 8
Hardware and Software Requirements 8
Tutorial Design Description 8

Lab 1: 7 Series Basic DFX Flow **12**
Overview 12
Step 1: Extract the Tutorial Design Files 12
Step 2: Examining the Scripts 12
Step 3: Synthesizing the Design 14
Step 4: Assembling and Implementing the Design 15
Step 5: Building the Design Floorplan 16
Step 6: Implementing the First Configuration 22
Step 7: Implementing the Second Configuration 27
Step 8: Examining the Results with Highlighting Scripts 29
Step 9: Generating Bitstreams 31
Step 10: Partially Reconfiguring the FPGA 33
Lab 1 Conclusion 34

Lab 2: UltraScale and UltraScale+ Basic DFX Flow **35**
Overview 35
Step 1: Extract the Tutorial Design Files 35
Step 2: Examining the Scripts 35
Step 3: Synthesizing the Design 37
Step 4: Assembling and Implementing the Design 38
Step 5: Build the Design Floorplan 39
Step 6: Implementing the First Configuration 41
Step 7: Implementing the Second Configuration 45
Step 8: Examine the Results with Highlighting Utilities 47
Step 9: Generating the Bitstreams 49
Step 10: Partially Reconfiguring the FPGA 51
Lab 2 Conclusion 52

Lab 3: DFX RTL Project Flow 54
Overview 54
Step 1: Extract the Tutorial Design Files 54
Step 2: Load Initial Design Sources 55
Step 3: Completing the Design with the Dynamic Function eXchange Wizard 59
Step 4: Synthesizing and Implementing the Current Design 63
Step 5: Adding an Additional Reconfigurable Module and Corresponding Configuration 67
Step 6: Creating and Implementing a Greybox Module 70
Step 7: Modifying a Design Source or Options 73
Lab 3 Conclusion 74

Lab 4: Vivado Debug and the DFX Project Flow 75
Overview 75
Step 1: Extract the Tutorial Design Files 75
Step 2: Loading Initial Design Sources 76
Step 3: Setting Up the Design for DFX 78
Step 4: Using the DFX Wizard to Complete the Rest of the Design 81
Step 5: Adding IP in the Reconfigurable Module 85
Step 6: Synthesizing the Design and Creating a Floorplan 87
Step 7: Running the PR Configuration Analysis Report 91
Step 8: Implementing the Design 92
Step 9: Adding an Additional Reconfigurable Module and Corresponding Configuration 96
Step 10: Generating Bitstreams 100
Step 11: Connecting to the Board and Programming the FPGA 101
Lab 4 Conclusion 108

Lab 5: DFX Controller IP for 7 Series Devices 109
Overview 109
Step 1: Extract the Tutorial Design Files 109
Step 2: Customizing the Dynamic Function eXchange (DFX) Controller IP 110
Step 3: Compiling the Design 117
Step 4: Setting Up the Board 118
Step 5: Operating the Sample Design 119
Step 6: Querying the DFX Controller in the FPGA 122
Step 7: Modifying the DFX Controller in the FPGA 124
Lab 5 Conclusion 125

Lab 6: DFX Controller IP for UltraScale Devices **127**
Overview 127
Step 1: Extract the Tutorial Design Files 127
Step 2: Customizing the Dynamic Function eXchange (DFX) Controller IP 128
Step 3: Compiling the Design 135
Step 4: Setting up the Board 136
Step 5: Operating the Sample Design 137
Step 6: Querying the DFX Controller in the FPGA 139
Step 7: Modifying the DFX Controller in the FPGA 141
Lab 6 Conclusion 142

Lab 7: DFX Controller IP for UltraScale+ Devices **143**
Overview 143
Step 1: Extract the Tutorial Design Files 143
Step 2: Processing the Tutorial Design 143
Step 3: Running the Tutorial Design 152
Lab 7 Conclusion 159

Lab 8: Nested Dynamic Function eXchange **160**
Overview 160
Step 1: Extracting the Tutorial Design Files 160
Step 2: Examining the Scripts 160
Step 3: Synthesizing the Design 162
Step 4: Assembling and Implementing the Design 163
Step 5: Test the Design in Hardware 173
Lab 8 Conclusion 175

Lab 9: Abstract Shell for Dynamic Function eXchange **177**
Overview 177
Step 1: Extracting the Tutorial Design Files 177
Step 2: Processing the Tutorial Design 178
Step 3: Create Abstract Shells 179
Step 4: Implement New RM within Abstract Shells 183
Step 5: Validate the Design in Hardware 186
Complete Hardware Validation 188
Lab 9 Conclusion 188

Lab 10: DFX BDC Project Flow in IP Integrator for Zynq UltraScale+ 189
Overview 189
Flow Summary 189
DFX Project Tutorial within IP Integrator 190
Vivado Hardware Design Flow 191
Step 1: Create a Flat Design in Vivado IP Integrator 191
Step 2: Create Levels of Hierarchy in the Block Design 192
Step 3: Create a Block Design Container 194
Step 4: Enable Dynamic Function eXchange 196
Step 5: Add a New Reconfigurable Module 199
Step 6: Confirm Apertures for All Reconfigurable Modules 203
Step 7: Create a Wrapper and Generate the Targets for the Top BD 204
Step 8: Use the DFX Wizard to Define Configurations 206
Step 9: Add Design Constraints for the Reconfigurable Partition 208
Step 10: Implement the Configurations and Generate Bitstreams 209
Step 11: Export the Hardware Platform for Each Configuration 210
Lab 10 Conclusion 211
Flow Summary 211

Lab 11: DFX BDC Project Flow in IP Integrator for Versal 213
Overview 213
Flow Summary 213
DFX Project Tutorial within IP Integrator 214
Vivado Hardware Design Flow 215
Step 1: Create a Flat Design in Vivado IP Integrator 215
Step 2: Create Levels of Hierarchy in the Block Design 216
Step 3: Create a Block Design Container 218
Step 4: Enable Dynamic Function eXchange 220
Step 5: Add a New Reconfigurable Module 224
Step 6: Confirm Apertures for All Reconfigurable Modules 228
Step 7: Create a Wrapper and Generate the Targets for the Top BD 230
Step 8: Use the DFX Wizard to Define Configurations 232
Step 9: Add Design Constraints for the Reconfigurable Partition 234
Step 10: Implement the Configurations and Generate Bitstreams 235
Step 11: Export the Hardware Platform for Each Configuration 236
Lab 11 Conclusion 237

Lab 12: Abstract Shell Project Mode **238**
Overview 238
Step 1: Extract the Tutorial Design Files 238
Step 2: Compile the Design in the Vivado Tools 239
Step 3: Set Up DFX Design Runs 243
Step 4: Examine the Results 251
Step 5: Hardware Validation 254
Lab 12 Conclusion 259

Appendix A: Additional Resources and Legal Notices **261**
Finding Additional Documentation 261
Support Resources 262
References 262
Revision History 262
Please Read: Important Legal Notices 263

Introduction

This tutorial covers the Dynamic Function eXchange (DFX) software support in the AMD Vivado™ Design Suite.

Lab 1: 7 Series Basic DFX Flow and Lab 2: UltraScale and UltraScale+ Basic DFX Flow step through basic information about the current DFX design flow, example Tcl scripts, and results within the Vivado Integrated Design Environment (IDE). You run scripts for part of the lab and work interactively with the design for other parts. You can also script the entire flow and a completed script is included with the design files. These labs focus specifically on the software flow from RTL to bitstream, demonstrating how to process a DFX design. Lab 2 also applies to AMD UltraScale+™ devices.

Lab 3: DFX RTL Project Flow guides you through the project flow within the Vivado IDE, from establishing the design using the DFX Wizard to synthesis, implementation runs, and then iterating through the design. Lab 4: Vivado Debug and the DFX Project Flow also walks you through the project flow, but includes adding IP, debug cores, and debugging through the Vivado Hardware Manager.

Lab 5: DFX Controller IP for 7 Series Devices, Lab 6: DFX Controller IP for UltraScale Devices, and Lab 7: DFX Controller IP for UltraScale+ Devices are designed to show the fundamental details and capabilities of the DFX Controller IP in the Vivado Design Suite. Managing partial bitstreams is one of the new design requirements introduced by DFX: designers plan for when partial bitstreams are required, where they are stored, how they are delivered to the configuration engine, and how the static design behaves before, during, and after the delivery of a new partial bitstream. The DFX Controller IP is designed to help users solve these challenges.

Lab 8: Nested Dynamic Function eXchange shows the flow and methodologies for nesting a reconfigurable partition inside a reconfigurable partition. This extension to the DFX solution further extends the flexibility of dynamic reconfiguration in any UltraScale or UltraScale+ device.

Lab 9: Abstract Shell for Dynamic Function eXchange shows how users can improve Vivado compile time by stripping away the bulk of the static logic and parallelizing runs when using UltraScale+ devices in non-project mode. This feature also safeguards static design information when applying DFX in multi-user environments.

Lab 10: DFX BDC Project Flow in IP Integrator for Zynq UltraScale+ and Lab 11: DFX BDC Project Flow in IP Integrator for Versal show the methodologies and capabilities of building and processing DFX designs within IP integrator. Block Design Containers allow users to build Reconfigurable Modules as block designs inserted in a top-level block design. These labs show examples targeting AMD Zynq™ AMD UltraScale+™ and AMD Versal™ devices, but all architectures are supported.

Lab 12: Abstract Shell Project Mode shows how the Abstract Shell feature has been integrated into project mode within the IDE to reduce compile time for child runs in UltraScale+ and Versal device DFX designs. This lab also shows the infrastructure for debug within static and dynamic regions for Versal device DFX designs.

In addition to the tutorial examples in this document, many more can be found on the GitHub repository. For examples showing new features and capabilities for Versal devices, see the Vivado Design Suite DFX Tutorials for Versal Devices. For architecture-independent design flow examples using Block Design Containers through IP integrator, see the Vivado Design Suite General DFX Tutorials section of the repository. For additional targeted examples, see the Vivado Design Suite DFX Tutorials for UltraScale+ Devices.

Navigating Content by Design Process

- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the AMD Vivado™ timing, resource use, and power closure. Also involves developing the hardware platform for system integration.
- **Board System Design:** Designing a PCB through schematics and board layout. Also involves power, thermal, and signal integrity considerations.

Hardware and Software Requirements

This tutorial requires that the 2025.2 Vivado Design Suite software release or later is installed.

The labs in this tutorial document target eight different AMD development platforms. Unless specifically noted, production silicon and production boards are required to match the instructions in each lab. For operating systems support, see the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973) for a complete list and description of the system and software requirements.

Tutorial Design Description

Designs for the tutorial labs are available as a zipped archive. Each lab in this tutorial has its own folder within the zip file. To access the tutorial design files:

1. Download the reference design files.
2. Extract the zip file contents to any write-accessible location.

Lab 1: 7 Series Basic DFX Flow

The sample design used throughout this tutorial is called `led_shift_count_7s`. The design targets the following AMD development platforms:

- KC705 (xc7k325t)
- VC707 (xc7vx485t)
- VC709 (xc7vx690t)
- AC701 (xc7a200t)

This design is very small, which helps minimize data size and allows you to run the tutorial quickly, with minimal hardware requirements.

Lab 2: AMD UltraScale™ and AMD UltraScale+™ Basic DFX

The sample design used throughout this tutorial is called `led_shift_count_us`. The design targets the following AMD development platforms:

- KCU105 (xcku040)
- VCU108 (xcvu095)
- KCU116 (xcku5p)
- VCU118 (xcvu9p)

Lab 3: DFX RTL Project Flow

The sample design used throughout this tutorial is called `dfx_project`. It is a modified version of the `led_shift_count` design used in Lab 1: 7 Series Basic DFX Flow, modified to include two shift instances instead of one counter and one shifter. This change helps illustrate that a partition definition applies to all instances of a partition type. The design targets the following AMD development platforms:

- KC705 (xc7k325t)
- VC707 (xc7vx485t)
- VC709 (xc7vx690t)
- KCU105 (xcku040)
- KCU116 (xcku5p)
- VCU108 (xcvu095)
- VCU118 (xcvu9p)

Lab 4: Vivado Debug and the DFX Project Flow

The sample design used is called `dfx_project_debug`. The design targets the following AMD development platforms:

- KCU105 (xcku040)
- VCU108 (xcvu095)
- KCU116 (xcku5p)
- VCU118 (xcvu9p)

Lab 5: DFX Controller IP for 7 Series Devices

The sample design used throughout this tutorial is called `dfxc_7s` and is based on the design used in Lab 1: 7 Series Basic DFX Flow. The design targets the following AMD development platforms:

- KC705 (xc7k325t)
- VC707 (xc7vx485t)
- VC709 (xc7vx690t)

Lab 6: DFX Controller IP for UltraScale Devices

The sample design used throughout this tutorial is called `dfxc_us`. The design targets an xcvu095 device for use on the VCU108 demonstration board, Rev 1.0, and is based on the design used in Lab 2: UltraScale and UltraScale+ Basic DFX Flow .

Lab 7: DFX Controller IP for UltraScale+ Devices

The sample design used throughout this tutorial is called `dfxc_usp` and is based on the design used in Lab 6: DFX Controller IP for UltraScale Devices, but adds a MicroBlaze™ manager for organizing DFX events. The design targets the VCU118 demonstration board.

Lab 8: Nested Dynamic Function eXchange

The sample design in this tutorial is another variation on the shift-count design, where you can configure the shifter or counter for all eight LEDs, or reconfigure at a lower granularity, changing only four of the LEDs. This design targets the same UltraScale and UltraScale+ development platforms as Lab 4: Vivado Debug and the DFX Project Flow: KCU105, VCU108, KCU116, and VCU118.

Lab 9: Abstract Shell for Dynamic Function eXchange

The sample design in this tutorial is called `abstract_shell` and targets the same UltraScale+ development platform as Lab 7: DFX Controller IP for UltraScale+ Devices: the VCU118. This lab shows how Vivado compile time can be reduced by abstracting away the bulk of the static design for child runs in non-project mode.

Lab 10: DFX BDC Project Flow in IP Integrator for Zynq UltraScale+

The sample design in this tutorial is a simple block design example that targets the ZCU102 (xczu9eg). Two AXI GPIO are inserted, one in the static region and one in the dynamic region. Block Design Containers are leveraged to manage creation of reconfigurable partitions, with each reconfigurable module inserted as a new block design.

Lab 11: DFX BDC Project Flow in IP Integrator for Versal

The sample design in this tutorial is a variation on Lab 10: DFX BDC Project Flow in IP Integrator for Zynq UltraScale+, this time targeting the VCK190 (xcvc1902). Differences in the design are based on architectural differences: the NoC is introduced and INI ports connect to the dynamic region, and the DFX decoupler is removed as its functionality is embedded in the NoC.

Lab 12: Abstract Shell Project Mode

The sample design in this tutorial is a Versal DFX design targeting the VCK190 (xcvc1902). IP integrator and block design containers are used to build the design.

Lab 1

7 Series Basic DFX Flow

Overview

This lab introduces the basic Dynamic Function eXchange (DFX) flow for 7 series devices. First, use a script to individually synthesize the static module and each reconfigurable design module variant. Then in the IDE, constrain the location of the reconfigurable modules (RM) using Pblocks and implement the initial configuration of the design. Next, implement alternate configurations by locking the static portion of the design, updating the reconfigurable modules with a variant, and re-running implementation. Finally, verify that each implemented RM is compatible with the static portion of the design and, if compatible, generate bitstreams.

Step 1: Extract the Tutorial Design Files

1. Download the reference design files.
2. Extract the zip file contents to any write-accessible location.
3. In the extracted files, navigate to `\led_shift_count_7s`.

Step 2: Examining the Scripts

Start by reviewing the scripts provided in the design archive. The files `run_dfx.tcl` and `advanced_settings.tcl` are located at the root level. The `run_dfx.tcl` script contains the minimum required settings to run Dynamic Function eXchange. The `advanced_settings.tcl` contains default flow settings and should only be modified by experienced users.

The Main Script

In `\led_shift_count_7s`, open `run_dfx.tcl` in a text editor. This is the master script where you define the design parameters, design sources, and design structure. This is the only file you have to modify to compile a complete dynamic function eXchange design. Find more details regarding `run_dfx.tcl`, `advanced_settings.tcl`, and the underlying scripts in the `README.txt` located in the `Tcl_HD` subdirectory.

Note the following details in this `run_dfx.tcl`:

- Under Define target demo board, you can select one of many demonstration boards supported for this design.
- Under flow control, you can control which phases of synthesis and implementation are run. In the tutorial, only synthesis is run by the script; implementation, verification, and bitstream generation are run interactively. To run these additional steps via the script, set the flow variables (for example, `run.prImpl` to `1`).
- The output directories and input directories set the file structure expected for design sources and results files. You must reflect any changes to your file structure here.
- The top definition and RP module definitions sections let you reference all source files for each part of your design. Top definition covers all sources needed for the static design, including constraints and IP. The RP module definitions section does the same for reconfigurable partitions (RP). Identify each RP and list all reconfigurable module (RM) variants for each RP.
  - This design has two reconfigurable partitions (`inst_shift` and `inst_count`), and each RP has two module variants.
- The configuration definition sections define the sets of static and reconfigurable modules that make up a configuration.
  - This design has two configurations defined within the master script: `config_shift_right_count_up_implement` and `config_shift_left_count_down_import`.
  - You can create more configurations by adding RMs or by combining existing RMs.

The Supporting Scripts

Underneath the `Tcl_HD` subdirectory, several supporting Tcl scripts exist. The scripts are called by `run_dfx.tcl`, and they manage specific details for the dynamic function eXchange flow. Provided below are some details about a few of the key DFX scripts.



CAUTION! *Do not modify the supporting Tcl scripts.*

- **`step.tcl`:** Manages the current status of the design by monitoring checkpoints.
- **`synthesize.tcl`:** Manages all the details regarding the synthesis phase.

- `implement.tcl`: Manages all the details regarding the module implementation phase.
- `dfx_utils.tcl`: Manages all the details regarding the top-level implementation of a DFX design.
- `run.tcl`: Launches the actual runs for synthesis and implementation.
- `log_utils.tcl`: Handles report file creation at key points during the flow.

Remaining scripts provide details within these scripts (such as other `*_utils.tcl` scripts) or manage other hierarchical design flows (such as `hd_utils.tcl`).

Step 3: Synthesizing the Design

The `run_dfx.tcl` script automates the synthesis phase of this tutorial. Five iterations of synthesis are called, one for the static top-level design and one for each of four reconfigurable modules.

1. Open the AMD Vivado™ Tcl shell:
   - On Windows, select the Vivado desktop icon or select **Start → All Programs → Xilinx Design Tools → Vivado 2026.1 → Vivado 2026.1 Tcl Shell**.
   - On Linux, type `vivado -mode tcl`.
2. In the shell, navigate to `\led_shift_count_7s`.
3. If you are using a target demonstration board other than the KC705, modify the xboard variable in `run_dfx.tcl`. Valid alternatives are the VC707, VC709, and AC701 boards.
4. Run the `run_dfx.tcl` script by entering:

   ```
   source run_dfx.tcl -notrace
   ```

   After all five passes through Vivado Synthesis have completed, the Vivado Tcl shell remains open. You can find log and report files for each module, alongside the final checkpoints, under each named folder in the Synth subdirectory.

   TIP: *In `\led_shift_count_7s`, multiple log files are created:*
   - `run.log` shows the summary as posted in the Tcl shell window
   - `command.log` echoes all the individual steps run by the script
   - `critical.log` reports all critical warnings produced during the run

   Note: The `command.log` file is itself a Tcl run script. This file can be modified if desired and sources as an input to reproduce the same results as an alternative to the more complex and parameterized Tcl_HD scripts.

Step 4: Assembling and Implementing the Design

Now that the synthesized checkpoints for each module, plus top, are available, you can assemble the design.

Run all flow steps from the Tcl Console, but you can use features within the IDE (such as the floorplanning tool) for interactive events.

TIP: *Copy and paste commands directly from the tutorial to avoid redundant effort and typos in the Vivado IDE. Copy and paste only one full command at a time. Some commands are long and span multiple lines.*

1. Open the Vivado IDE. You can open the IDE from the open Tcl shell by typing `start_gui` or by launching Vivado with the command `vivado -mode gui`.
2. Navigate to `\led_shift_count_7s`, if you are not already there. The `pwd` command can confirm this.
3. Set variables that help with copying commands from this document into the Tcl Console. Select the part and board you are targeting for this lab, and apply them in Vivado:

```
set part "xc7k325t-ffg900-2"
set board "kc705"
set part "xc7vx485t-ffg1761-2"
set board "vc707"
set part "xc7vx690t-ffg1761-2"
set board "vc709"
set part "xc7a200t-fbg676-2"
set board "ac701"
```

4. Create an in-memory design by issuing the following command in the Tcl Console:

```
create_project -in_memory -part $part
```

5. Load the static design by issuing the following command:

```
add_files ./Synth/Static/top_synth.dcp
```

6. Load the top-level design constraints by issuing these commands:

```
add_files ./Sources/xdc/top_io_$board.xdc
set_property USED_IN {implementation} [get_files ./Sources/xdc/
top_io_$board.xdc]
```

   Selecting the top_io_$board version of the available xdc file loads the pin location and clocking constraints, but does not include floorplan information. The top_$board version includes pin location, clocking, and floorplanning constraints.

7. Load the first two synthesis checkpoints for the shift and count functions by issuing these commands:

```
add_files ./Synth/shift_right/shift_synth.dcp
set_property SCOPED_TO_CELLS {inst_shift} [get_files ./Synth/shift_right/
shift_synth.dcp]
add_files ./Synth/count_up/count_synth.dcp
set_property SCOPED_TO_CELLS {inst_count} [get_files ./Synth/count_up/
count_synth.dcp]
```

   The SCOPED_TO_CELLS property ensures that the proper assignment is made to the target cell. See the *Vivado Design Suite User Guide: Using Constraints* (UG903) for more information.

8. Link the entire design together using the `link_design` command:

```
link_design -mode default -reconfig_partitions {inst_shift inst_count}
-part $part -top top
```

   At this point a full configuration is loaded, including static and reconfigurable logic. Note that the Flow Navigator pane is not present while you are working in non-project mode.

> **TIP:** *Place the IDE in floorplanning mode by selecting **Layout → Floorplanning**. Make sure the Device window is visible.*

9. Save the assembled design state for this initial configuration:

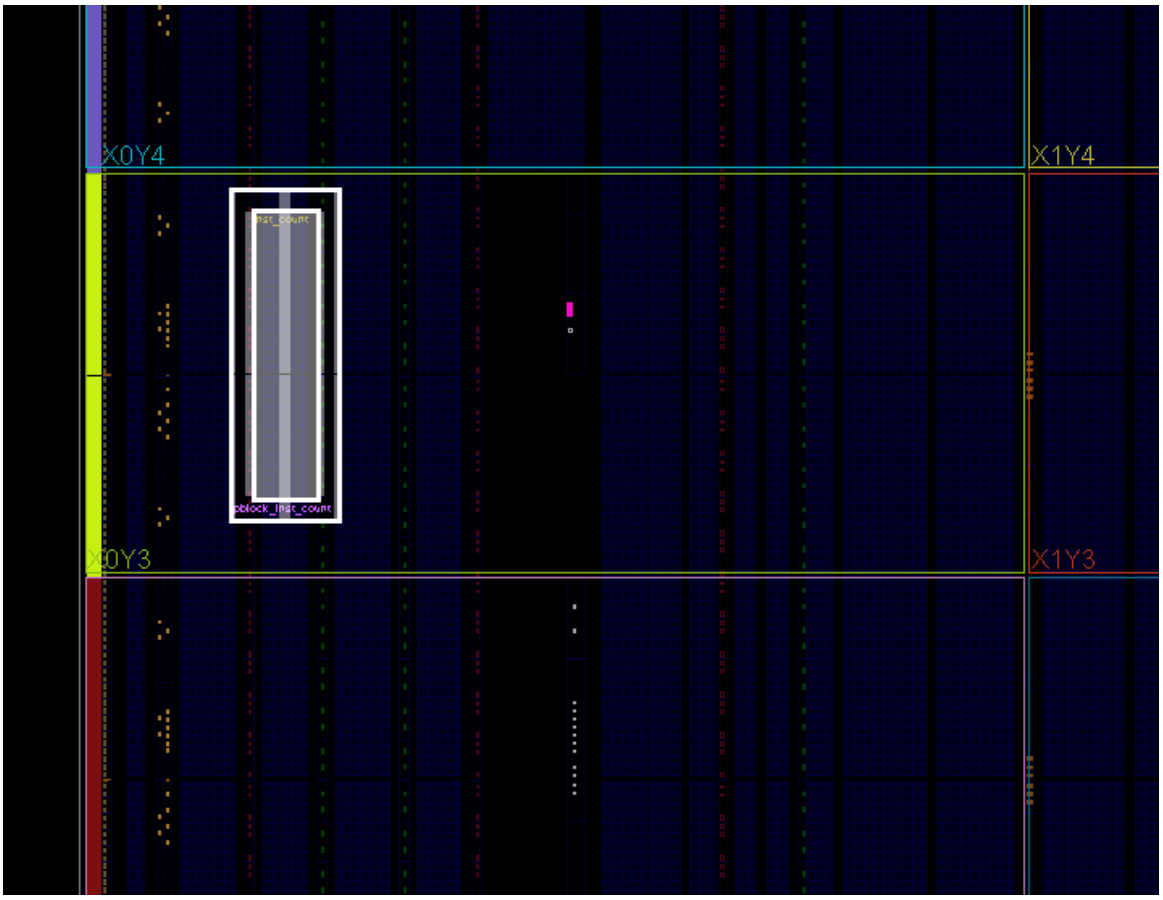
```
write_checkpoint -force ./Checkpoint/top_link_right_up.dcp
```

Step 5: Building the Design Floorplan

Next, create a floorplan to define the regions that are to be partially reconfigured.

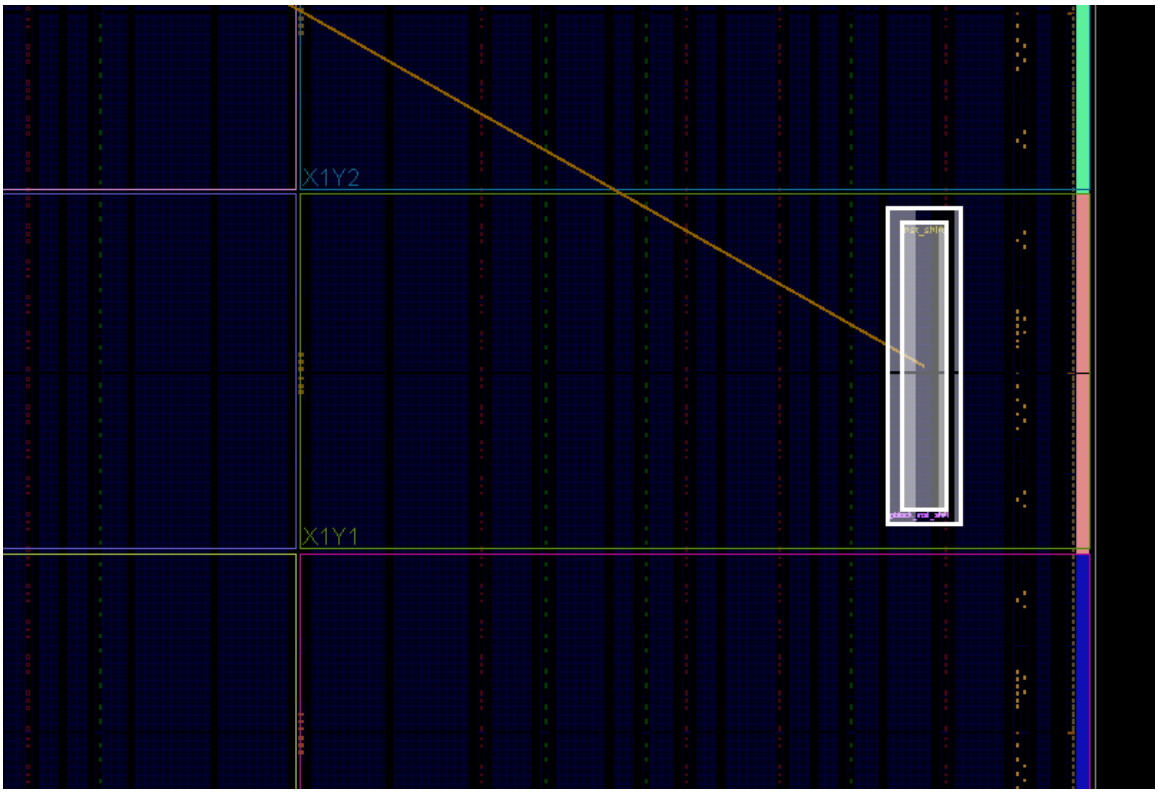
1. Select the inst_count instance in the Netlist window. Right-click and select **Floorplanning → Draw & Create Pblock**, or select the **Draw Pblock** toolbar button, and draw a tall narrow box on the left side of the X0Y3 clock region. The exact size and shape do not matter at this point, but keep the box within the clock region.

   Make sure that the Pblock is selected in the Device window before continuing.

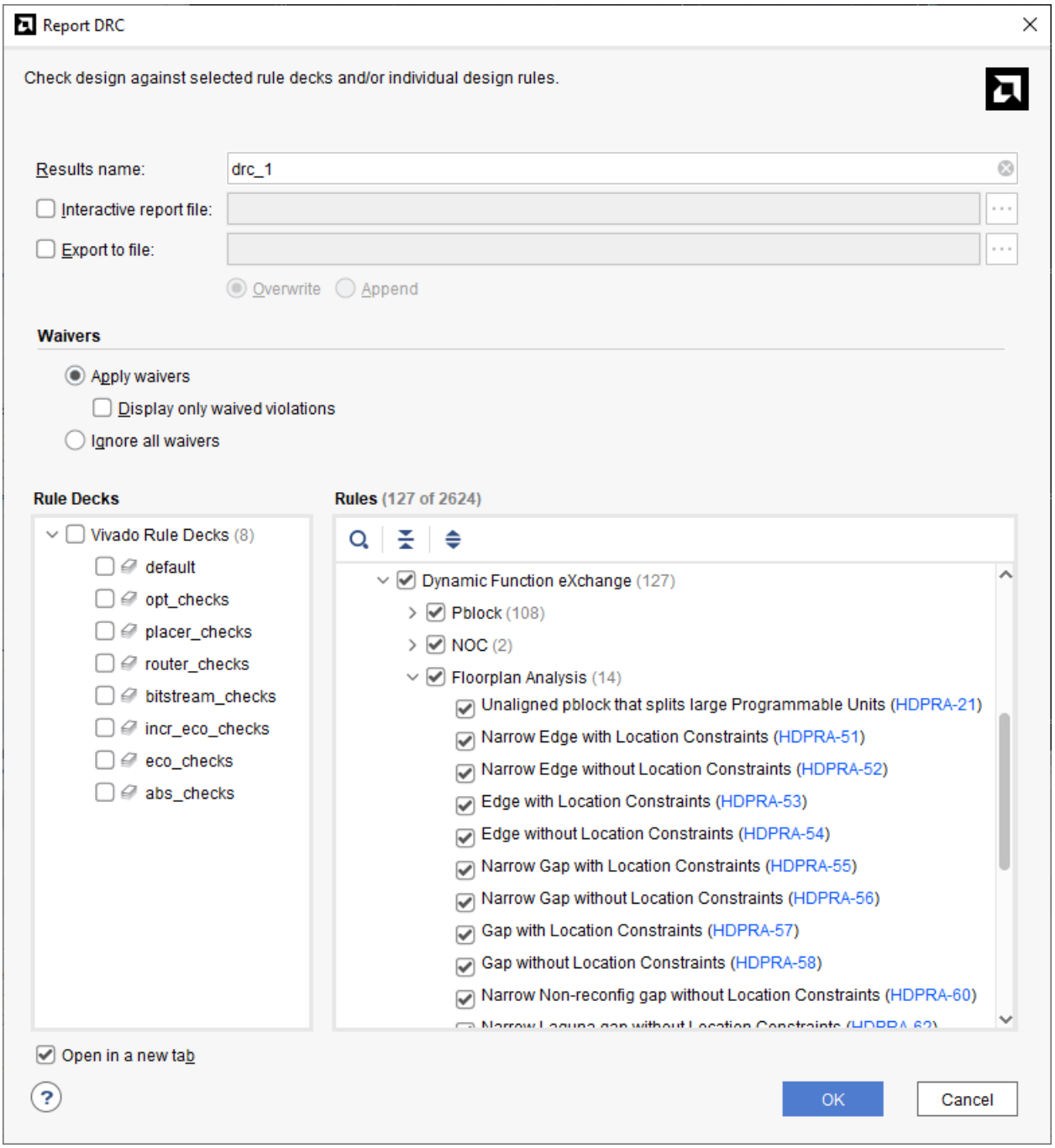


Although this Reconfigurable Module only requires CLB resources, it also includes RAMB18, RAMB36, or DSP48 resources if the box encompasses those types. This allows the routing resources for these block types to be included in the reconfigurable region. The General view of the Pblock Properties window can be used to add these if needed. The Statistics view shows the resource requirements of the currently loaded Reconfigurable Module.

2. In the Properties view, select the checkbox for RESET_AFTER_RECONFIG to use the dedicated initialization of the logic in this module after reconfiguration completes.
3. Repeat steps 1 and 2 for the inst_shift instance, this time targeting the right side of clock region X1Y1. This Reconfigurable Module includes block RAM instances, so the resource type must be included. If omitted, the RAMB details in the Statistics view are shown in red.



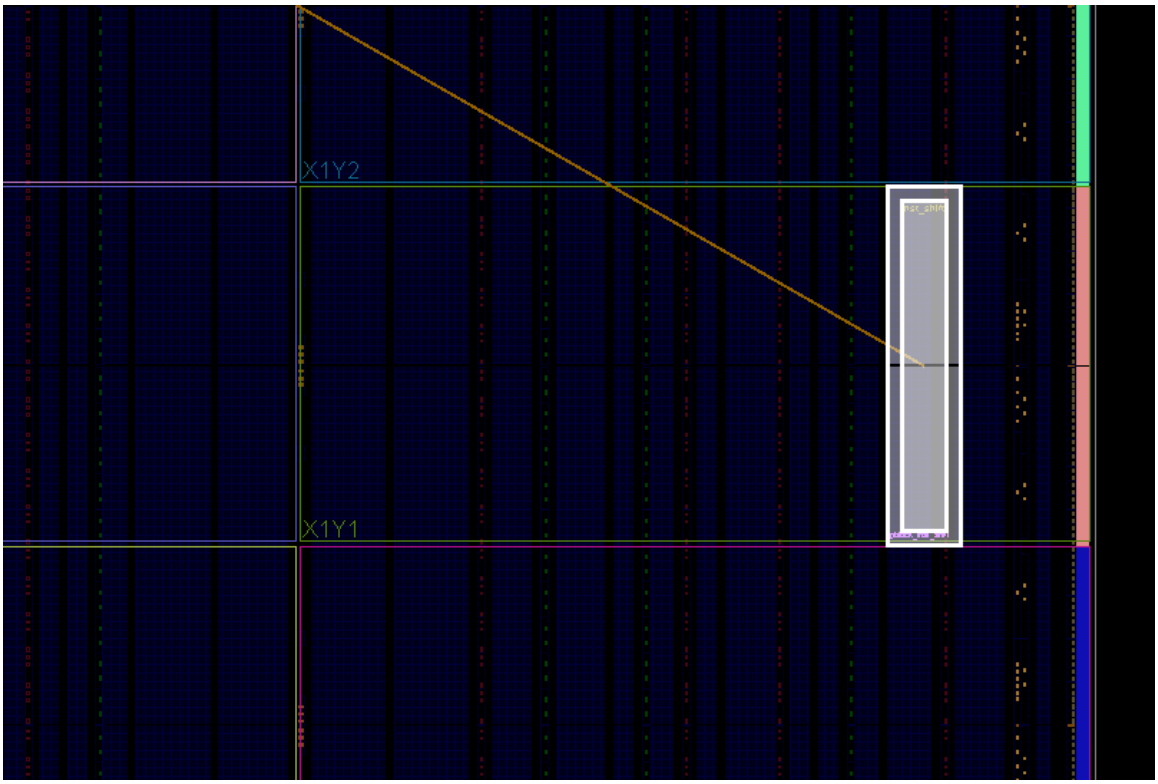
4. Run DFX Design Rule Checks by selecting **Reports → Report DRC**. You can uncheck All Rules and then check Dynamic Function eXchange to focus this report strictly on DFX DRCs.



One or two DRCs are reported at this point, and there are two ways of resolving them. For this lab, use one method for inst_shift and the other for inst_count.

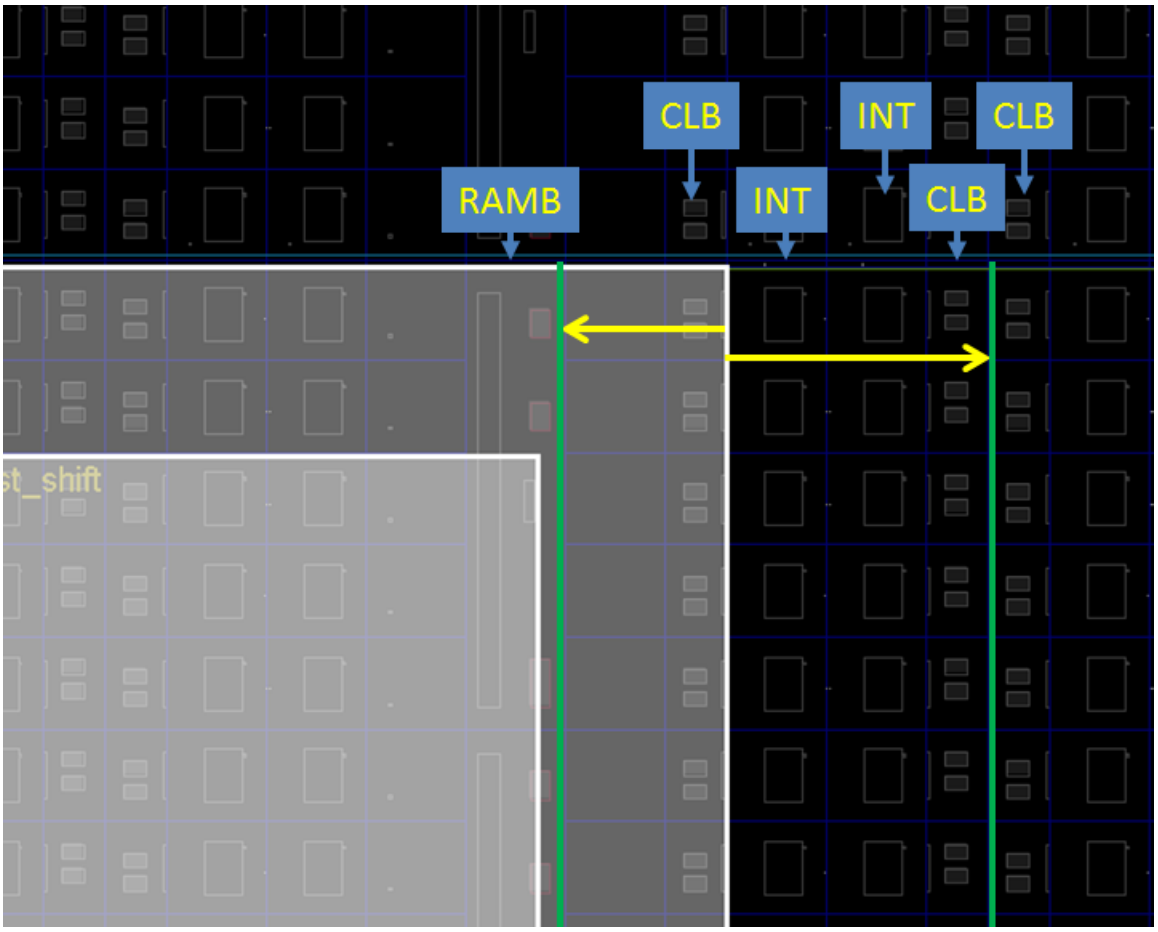
The first DRC is an error, HDPR-10, reporting that RESET_AFTER_RECONFIG requires Pblock frame alignment.

5. To resolve the first DRC error, make sure that the height of the Pblock aligns with the clock region boundaries. Using the Pblock for inst_shift, stretch the top and bottom edges to match the clock region boundaries of X1Y1 as shown in the following figure. See that the shading of the Pblock is now more uniform.



The other possible DRC is a warning, HDPR-26, reporting that a left or right edge of a reconfigurable Pblock terminates on an improper boundary. Left or right edges must not split interconnect (INT) columns. For more information on this requirement, see the *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909).

6. To manually avoid this DRC warning, zoom into the upper or lower corner on the reported edge of inst_shift (or inst_count, if inst_shift did not report an issue) to see where the violation occurred. Move this edge left or right one column, as shown by the yellow arrows in the following figure so it lands between two resource types (CLB-CLB or CLB-RAMB, for example) instead landing between CLB-INT or RAMB-INT.



7. Run the PR DRCs again to confirm that the errors and warnings that you have addressed have been resolved for the inst_shift instance.

   An alternative to manually adjusting the size and shape of reconfigurable Pblocks is to use the SNAPPING_MODE feature. This feature automatically adjusts edges to align with legal boundaries. It will make the Pblock taller, aligning with clock region boundaries, if the RESET_AFTER_RECONFIG feature is selected. It makes the Pblock narrower, adjusting left and/or right edges as needed. Note that the number and type of resources available are altered if SNAPPING_MODE makes changes to the Pblock.

8. Select the Pblock for inst_count in the Device window, and in the Properties view of the Pblock Properties window, change the value of SNAPPING_MODE from OFF to ROUTING (or ON).

   Note: The original Pblock does not change, but the shading behind it does. The adjustments to the Pblock needed for it to conform to DFX rules are done automatically, without modifying your source constraints.

9. Run the DFX DRCs once again to confirm that all errors have been resolved. Advisory messages might still be reported, especially if the Pblock is located near the edge of the device.

10. Save these Pblocks and associated properties:

```
write_xdc ./Sources/xdc/top_all.xdc
```

This exports all the current constraints in the design, including those imported earlier from `top_io_$board.xdc`. These constraints can be managed in their own XDC file or managed within a run script (as is typically done with `HD.RECONFIGURABLE`).

Alternatively, the Pblock constraints themselves can be extracted and managed separately. A Tcl proc is available to help perform this task.

a. Source the proc which is found in one of the Tcl utility files:

```
source ./Tcl_HD/hd_utils.tcl
```

b. Use the `export_pblocks` proc to write out this constraint information:

```
export_pblocks -file ./Sources/xdc/pblocks.xdc
```

This writes the Pblock constraint information for both Pblocks in the design. Use the `-pblocks` option to select only one if desired.

Now that the floorplan is established, the next step is implementing the design.

Step 6: Implementing the First Configuration

In this step, you place and route the design and prepare the static portion of the design for reuse with new Reconfigurable Modules.

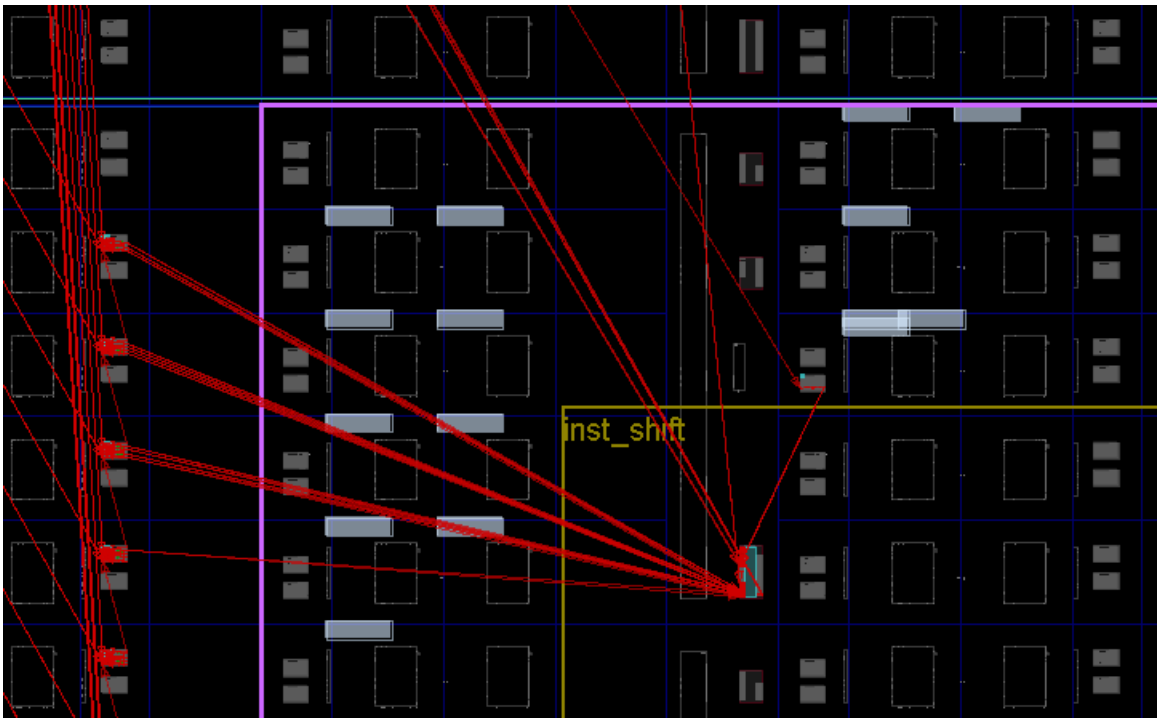
Implementing the Design

1. Optimize, place, and route the design by issuing the following commands:

```
opt_design
place_design
route_design
```

After both `place_design` and `route_design`, examine the state of the design in the Device view as shown in the figure below. One thing to note after `place_design` is the introduction of Partition Pins. These are the physical interface points between static and reconfigurable logic. They are anchor points within an interconnect tile through which each I/O of the Reconfigurable Module must route. They appear as white boxes in the placed design view.

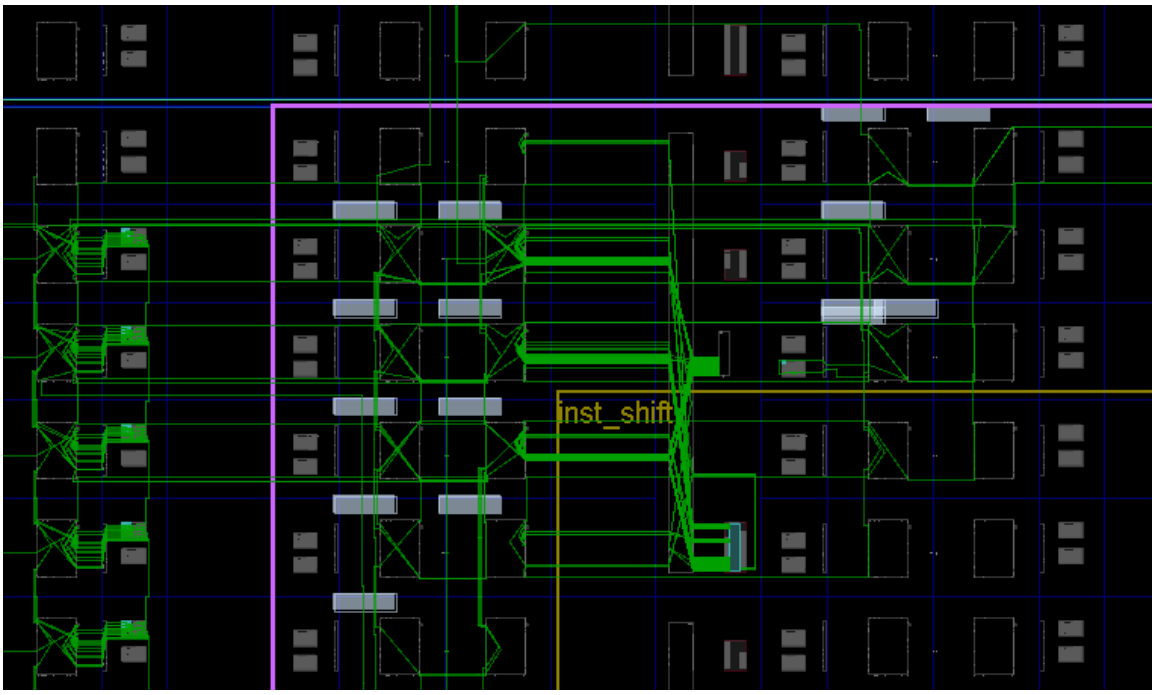
For `pblock_shift`, they appear in the top of that Pblock, as the connections to static are just outside the Pblock in that area of the device. For `pblock_count`, they appear outside the user-defined region, as SNAPPING_MODE vertically collects more frames to be added to the Reconfigurable Partition.



2. To find these partition pins in the GUI easily:
   a. Select the Reconfigurable Module (for example, inst_shift) in the Netlist pane.
   b. Select the Cell Pins tab in the Cell Properties pane.
3. Select any pin to highlight it, or press Ctrl+A to select them all. The Tcl equivalent of the latter is:

```
select_objects [get_pins inst_shift/*]
```

4. Use the Routing Resources toolbar button  to toggle between abstracted and actual routing information, and to change the visibility of the routing resources themselves. All nets in the design are fully routed at this point.



Saving the Results

1. Save the full design checkpoint and create report files by issuing these commands:

```
write_checkpoint -force Implement/Config_shift_right_count_up_implement/
top_route_design.dcp

report_utilization -file Implement/Config_shift_right_count_up_implement/
top_utilization.rpt

report_timing_summary -file Implement/
Config_shift_right_count_up_implement/top_timing_summary.rpt
```

2. [Optional] Save checkpoints for each of the Reconfigurable Modules by issuing these two commands:

```
write_checkpoint -force -cell inst_shift Checkpoint/
shift_right_route_design.dcp

write_checkpoint -force -cell inst_count Checkpoint/
count_up_route_design.dcp
```



TIP: *When running `run_dfx.tcl` to process the entire design in batch mode; design checkpoints, log files, and report files are created at each step of the flow.*

At this point, you have created a fully implemented Dynamic Function eXchange design from which you can generate full and partial bitstreams. The static portion of this configuration is used for all subsequent configurations. To isolate the static design, remove the current Reconfigurable Modules. Make sure routing resources are enabled, and zoom in to an interconnect tile with partition pins.

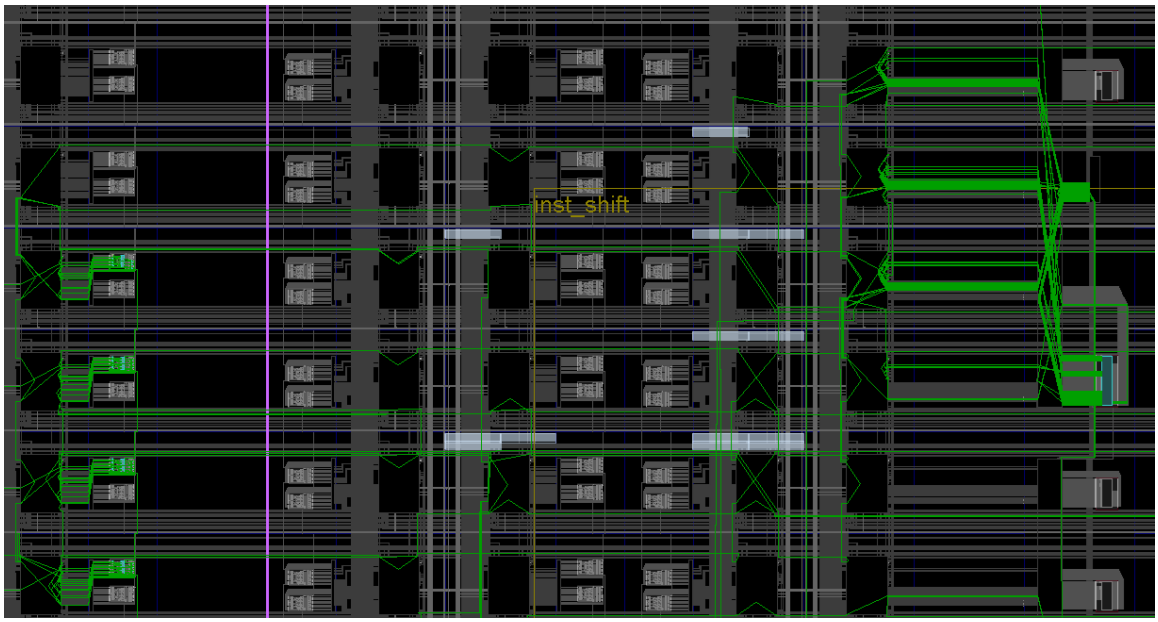
3. Clear out Reconfigurable Module logic by issuing the following commands:

```
update_design -cell inst_shift -black_box
update_design -cell inst_count -black_box
```

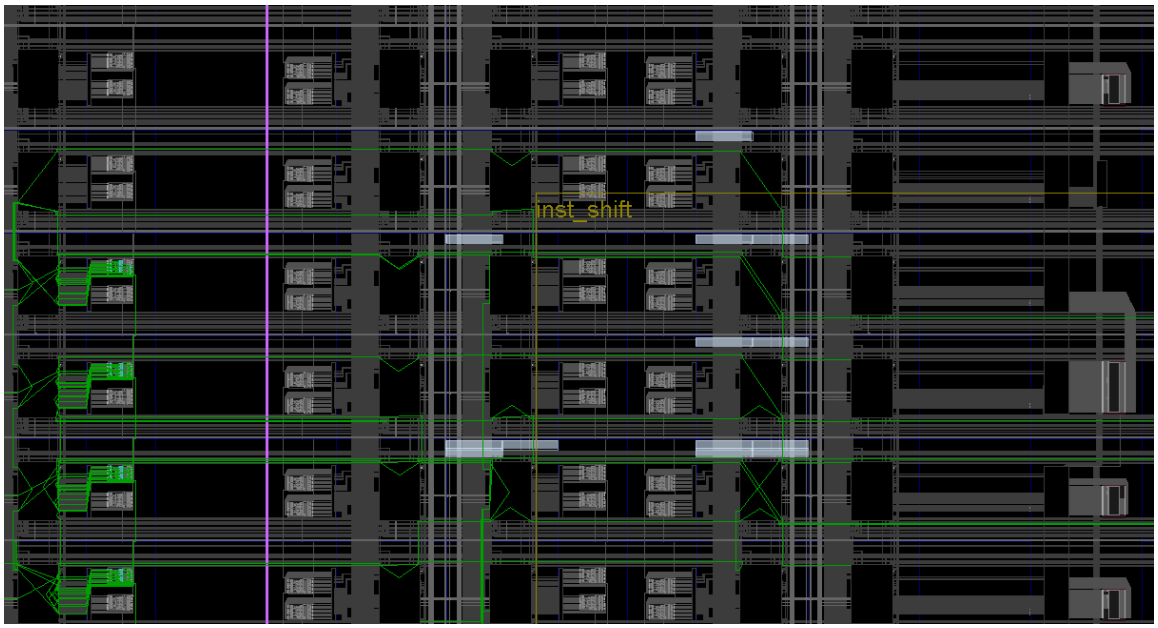
Issuing these commands results in many design changes as shown in the figure below:

- The number of Fully Routed nets (green) decreases.
- inst_shift and inst_count now appear in the Netlist view as empty.

The following figure shows the `inst_shift` module before `update_design -black_box`.



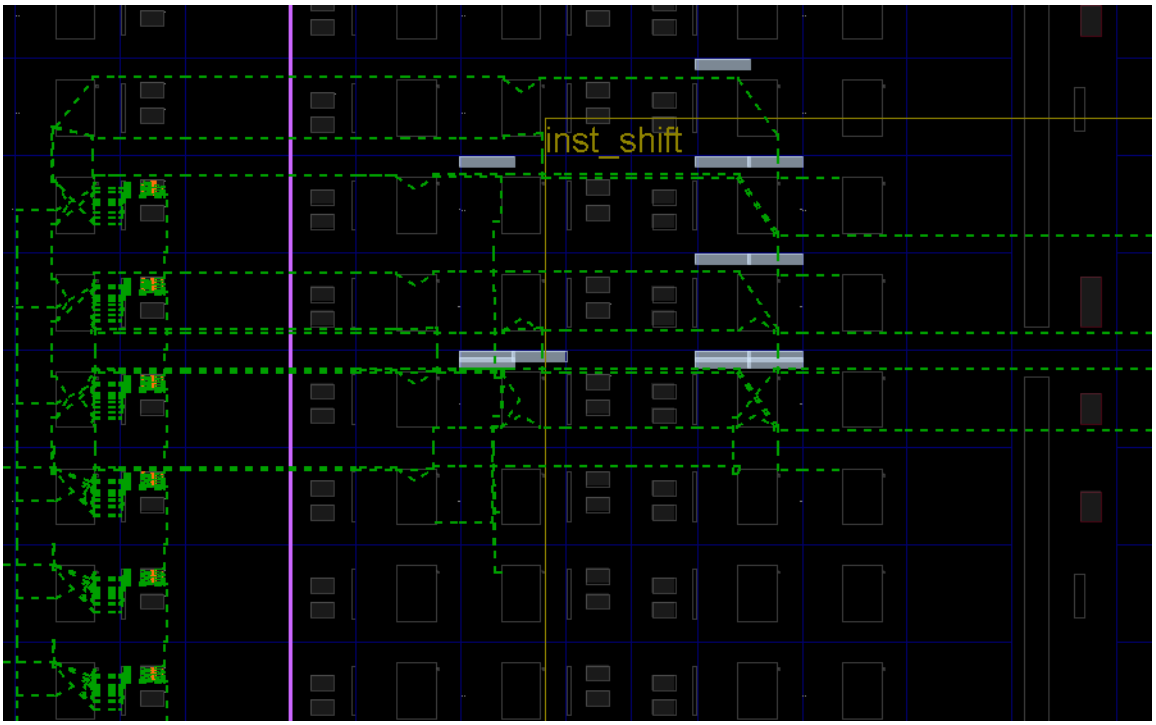
The following figure show the `inst_shift` module after `update_design -black_box`.



4. Issue the following command to lock down all placement and routing:

```
lock_design -level routing
```

Because no cell was identified in the lock_design command, the entire design in memory (currently consisting of the static design with black boxes) is affected. All routed nets are now displayed as locked, as indicated by dashed lines in the figure below. All placed components changed from blue to orange to show they are also locked.



5. Issue the following command to write out the remaining static-only checkpoint:

```
write_checkpoint -force Checkpoint/static_route_design.dcp
```

   This static-only checkpoint is used for future configurations.

6. Close this design before moving on to the next configuration:

```
close_project
```

Step 7: Implementing the Second Configuration

Now that the static design result is established and locked, you can use it as context for implementing further Reconfigurable Modules.

Implementing the Design

1. Create a new in-memory design by issuing the following command in the Tcl Console:

```
create_project -in_memory -part $part
```

2. Load the static design by issuing the following command:

```
add_files ./Checkpoint/static_route_design.dcp
```

3. Load the second two synthesis checkpoints for the shift and count functions by issuing these commands:

```
add_files ./Synth/shift_left/shift_synth.dcp

set_property SCOPED_TO_CELLS {inst_shift} [get_files ./Synth/shift_left/
shift_synth.dcp]

add_files ./Synth/count_down/count_synth.dcp

set_property SCOPED_TO_CELLS {inst_count} [get_files ./Synth/count_down/
count_synth.dcp]
```

4. Link the entire design together using the link_design command:

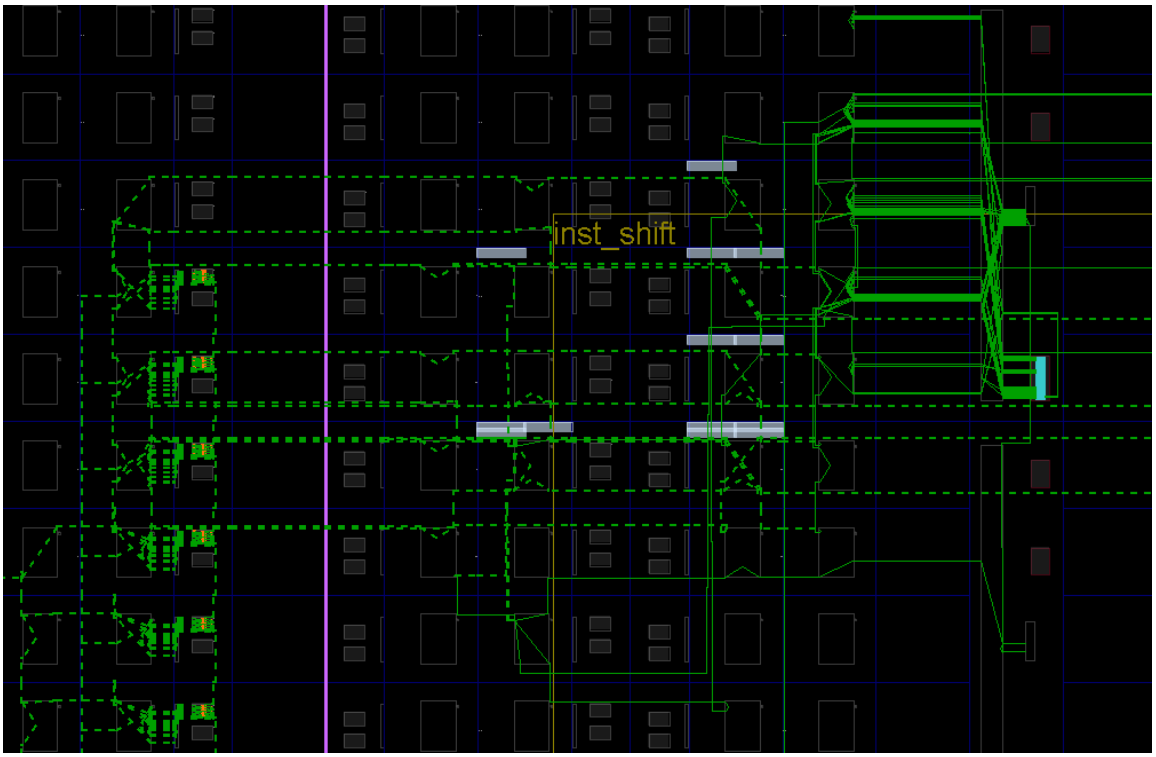
```
link_design -mode default -reconfig_partitions {inst_shift inst_count}
-part $part -top top
```

   At this point, a full configuration is loaded. This time, however, the static design is routed and locked, and the reconfigurable logic is still a netlist only. Place and route from here only applies to the RM logic.

5. Optimize, place, and route the new RMs in the context of static by issuing these commands:

```
opt_design

place_design

route_design
```

The design is again fully implemented, now with the new Reconfigurable Module variants. The routing is a mix of dashed (locked) and solid (new) routing segments, as shown below.



Saving the Results

1. Save the full design checkpoint and report files by issuing these commands:

```
write_checkpoint -force Implement/Config_shift_left_count_down_import/
top_route_design.dcp

report_utilization -file Implement/Config_shift_left_count_down_import/
top_utilization.rpt

report_timing_summary -file Implement/
Config_shift_left_count_down_import/top_timing_summary.rpt
```

2. [Optional] Save checkpoints for each of the Reconfigurable Modules by issuing these two commands:

```
write_checkpoint -force -cell inst_shift Checkpoint/
shift_left_route_design.dcp

write_checkpoint -force -cell inst_count Checkpoint/
count_down_route_design.dcp
```

   At this point, you have implemented the static design and all Reconfigurable Module variants. Repeat this process for designs that have more than two Reconfigurable Modules per Reconfigurable Partition.

Step 8: Examining the Results with Highlighting Scripts

With the routed configuration open in the IDE, run some visualization scripts to highlight tiles and nets. These scripts identify the resources allocated for Dynamic Function eXchange, and are automatically generated.

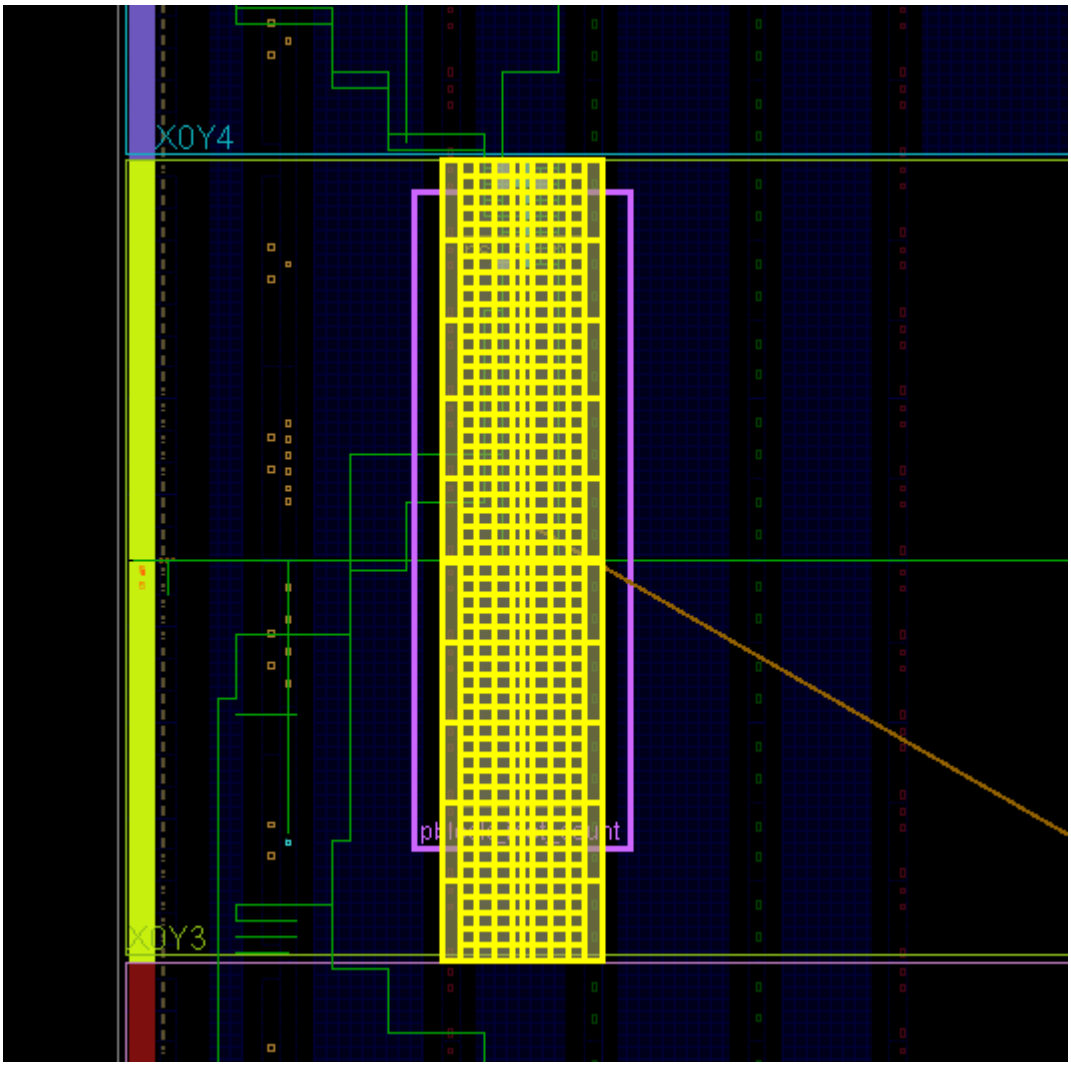
1. In the Tcl Console, issue the following commands from the <Extract_Dir> directory:

```
source hd_visual/pblock_inst_shift_AllTiles.tcl

highlight_objects -color blue [get_selected_objects]
```

2. Click somewhere in the Device view to deselect the frames (or enter `unselect_objects`), then issue the following commands:

```
source hd_visual/pblock_inst_count_AllTiles.tcl

highlight_objects -color yellow [get_selected_objects]
```

   The partition frames appear highlighted in the Device view, as shown in the following figure:



These highlighted tiles represent the configuration frames that are sent to bitstream generation to create the partial bitstreams. As shown above, the SNAPPING_MODE feature adjusted all four edges of pblock_count to account for RESET_AFTER_RECONFIG and legal reconfigurable partition widths.

The other "tile" scripts are variations on these. If you had not created Pblocks that vertically aligned to the clock region boundaries, the FrameTiles script would highlight the explicit Pblock tiles, while the AllTiles script extends those tiles to the full reconfigurable frame height. Note that these leave gaps where unselected frame types (for example: global clocks) exist.

The GlitchTiles script is a subset of frame sites, avoiding dedicated silicon resources; the other scripts are more informative than this one.

3. Close the current design:

```
close_project
```

Step 9: Generating Bitstreams

Verifying Configurations

RECOMMENDED: *Before generating bitstreams, verify all configurations to ensure that the static portion of each configuration match identically, so the resulting bitstreams are safe to use in silicon. The PR Verify feature examines the complete static design up to and including the partition pins, confirming that they are identical. Placement and routing within the Reconfigurable Modules is not checked, as different module results are expected here.*

1. Run the `pr_verify` command from the Tcl Console:

```
pr_verify Implement/Config_shift_right_count_up_implement/
top_route_design.dcp Implement/Config_shift_left_count_down_import/
top_route_design.dcp
```

   If successful, this command returns the following message.

```
INFO: [Vivado 12-3253] PR_VERIFY: check points Implement/
Config_shift_right_count_up/
top_route_design.dcp and Implement/Config_shift_left_count_down/
top_route_design.dcp are compatible
```

   By default, only the first mismatch (if any) is reported. To see all mismatches, use the `-full_check` option.

2. Close the project:

```
close_project
```

Generating Bitstreams

Now that the configurations are verified, you can generate bitstreams and use them to target your selected demonstration board.

***Note*:** The first configuration implements shift_right and count_up. The second configuration implements shift_left and count_down.

1. Read the first configuration into memory:

```
open_checkpoint Implement/Config_shift_right_count_up_implement/
top_route_design.dcp
```

2. Generate full and partial bitstreams for this design. Be sure to keep the bit files in a unique directory related to the full design checkpoint from which they were created.

```
write_bitstream -force -file Bitstreams/Config_RightUp.bit
close_project
```

   Three bitstreams are created:

   - `Config_RightUp.bit`

This is the power-up, full design bitstream. The four shift LEDs on the right shift right and the four count LEDs on the left count up.

- `Config_RightUp_Pblock_inst_shift_partial.bit`

  This is the partial bit file for the shift_right module.

- `Config_RightUp_Pblock_inst_count_partial.bit`

  This is the partial bit file for the count_up module that causes the count LEDs to count up.

> **IMPORTANT!** *When generated by a single call to write_bitstream, the names of the bit files currently do not reflect the name of the Reconfigurable Module variant to clarify which image is loaded. The current solution uses the base name given by the -file option and appends the Pblock name of the reconfigurable cell. It is critical to provide enough description in the base name to be able to identify the reconfigurable bit files clearly. All partial bit files have the _partial postfix.*

Using run_dfx.tcl to process the entire design through bitstream generation uses a different technique for generating the bitstreams. Opening a routed design checkpoint issues multiple calls to write_bitstream, which gives you more control over naming bitstreams and allows for different options (such a bitstream compression) to be applied to full versus partial bitstreams. For example, the names configured in the advanced_settings.tcl script are:

- Config_shift_right_count_up_implement_full.bit

  This is the power-up, full design bitstream.

- pblock_shift_shift_right_partial.bit

  This is the partial bit file for the shift_right module.

- pblock_count_count_up_partial.bit

  This is the partial bit file for the count_up module.

3. Generate full and partial bitstreams for the second configuration, again keeping the resulting bit files in the appropriate folder.

```
open_checkpoint Implement/Config_shift_left_count_down_import/
top_route_design.dcp
write_bitstream -force -file Bitstreams/Config_LeftDown.bit
close_project
```

Similarly, you see three bitstreams created, this time with a different base name.

4. Generate a full bitstream with grey boxes, plus blanking bitstreams for the Reconfigurable Modules. Blanking bitstreams can be used to “erase” an existing configuration to reduce power consumption.

```
open_checkpoint Checkpoint/static_route_design.dcp
update_design -cell inst_count -buffer_ports
update_design -cell inst_shift -buffer_ports
place_design
route_design
write_checkpoint -force Checkpoint/Config_greybox.dcp
write_bitstream -force -file Bitstreams/config_greybox.bit
close_project
```

   The base configuration bitstream has no logic for either reconfigurable partition. The update_design commands here insert constant drivers (ground) for all outputs of the Reconfigurable Partitions, so these outputs do not float. The term grey box indicates that the modules are not completely empty with these LUTs inserted, as opposed to black boxes, which would have dangling nets in and out of this region. The place_design and route_design commands ensure they are completely implemented.

Step 10: Partially Reconfiguring the FPGA

The `led_shift_count` design targets one of four demonstration boards. The current design supports the KC705, VC707, VC709 and AC701 boards, revisions Rev 1.0 and Rev 1.1.

Configuring the Device with a Full Image

1. Connect the board to your computer via the Platform Cable USB and power on the board.
2. From the main AMD Vivado™ IDE, select **Flow → Open Hardware Manager**.
3. Select **Open a new hardware target** on the green banner. Follow the steps in the wizard to establish communication with the board.
4. Select **Program device** on the green banner, and select the target device. Navigate to the `Bitstreams` folder to select **Config_RightUp.bit**, then click **OK** to program the device.

   You should now see the bank of GPIO LEDs performing two tasks. Four LEDs are performing a counting-up function (MSB is on the left), and the other four are shifting to the right. Note the amount of time it took to configure the full device.

   Note: The AC701 demonstration board only has a 4-bit LED bank. This design will show either the shift function or the count function at one time. To switch between the shift and count functions, toggle the switch 1 on the GPIO DIP switch (SW2).

Partially Reconfiguring the Device

At this point, you can partially reconfigure the active device with any of the partial bitstreams that you have created.

1. Select **Program device** on the green banner again. Navigate to the `Bitstreams` folder to select **Config_LeftDown_pblock_inst_shift_partial.bit**, then click **OK** to program the device.

   The shift portion of the LEDs changed direction, but the counter kept counting up, unaffected by the reconfiguration. Note the much shorter configuration time.

2. Select **Program device** on the green banner again. Navigate to the `Bitstreams` folder to select **Config_LeftDown_pblock_inst_count_partial.bit**, then click **OK** to program the device.

   The counter is now counting down, and the shifting LEDs were unaffected by the reconfiguration. This process can be repeated with the `Config_RightUp` partial bit files to return to the original configuration, or with the blanking (grey box) partial bit files to stop activity on the LEDs (that will stay on).

Lab 1 Conclusion

This concludes Lab 1. In this lab, you:

- Synthesized a design bottom-up to prepare for Dynamic Function eXchange implementation
- Created a valid floorplan for a Dynamic Function eXchange design
- Created two configurations with common static results
- Implemented these two configurations, saving the static design to be used in each
- Created checkpoints for static and reconfigurable modules for later reuse
- Examined framesets and verified the two configurations
- Created full and partial bitstreams
- Configured and partially reconfigured an FPGA

Lab 2

UltraScale and UltraScale+ Basic DFX Flow

Overview

This lab introduces the basic Dynamic Function eXchange (DFX) flow for AMD UltraScale™ and AMD UltraScale+™ devices. First, use a script to individually synthesize the static module and each reconfigurable design module variant. Then in the IDE, constrain the location of the reconfigurable modules (RM) using Pblocks and implement the initial configuration of the design. Next, implement alternate configurations by locking the static portion of the design, updating the reconfigurable modules with a variant, and re-running implementation. Finally, verify that each implemented RM is compatible with the static portion of the design and, if compatible, generate bitstreams.

Step 1: Extract the Tutorial Design Files

1. Download the reference design files.
2. Extract the zip file contents to any write-accessible location.
3. In the extracted files, navigate to `\led_shift_count_us`.

Step 2: Examining the Scripts

Start by reviewing the scripts provided in the design archive. The files `run_dfx.tcl` and `advanced_settings.tcl` are located at the root level. The `run_dfx.tcl` script contains the minimum required settings to run Dynamic Function eXchange. The `advanced_settings.tcl` contains default flow settings and should only be modified by experienced users.

The Main Script

In `\led_shift_count_us`, open `run_dfx.tcl` in a text editor. This is the master script where you define the design parameters, design sources, and design structure. This is the only file you have to modify to compile a complete Dynamic Function eXchange design. Find more details regarding `run_dfx.tcl`, `advanced_settings.tcl`, and the underlying scripts in the `README.txt` located in the `Tcl_HD` subdirectory.

Note the following details in this `run_dfx.tcl`:

- Under Define target demo board, you can select one of many demonstration boards supported for this design.
- Under flow control, you can control what phases of synthesis and implementation are run. In the tutorial, only synthesis is run by the script; implementation, verification, and bitstream generation are run interactively. To run these additional steps via the script, set the flow variables to 1 (for example, `run.prImpl`).
- The output and input directories set the file structure expected for design sources and results files. You must reflect any changes to your file structure here.
- The Top Definition and RP Module Definitions sections let you reference all source files for each part of your design. Top Definition covers all sources needed for the static design, including constraints and IP. The RP Module Definitions section does the same for reconfigurable partitions (RP). Identify each RP and list all Reconfigurable Module (RM) variants for each RP.
  - This design has two RPs (`inst_shift` and `inst_count`), and each RP has two module variants.
- The Configuration Definition sections define the sets of static and reconfigurable modules that make up a configuration.
  - This design has two configurations defined within the master script: `config_shift_right_count_up_implement` and `config_shift_left_count_down_import`.
  - You can create more configurations by adding RMs or by combining existing RMs.

The Supporting Scripts

Underneath the `Tcl_HD` subdirectory, several supporting Tcl scripts exist. The scripts are called by `run_dfx.tcl`, and they manage specific details for the Dynamic Function eXchange flow. Provided below are some details about a few of the key DFX scripts.



CAUTION! *Do not modify the supporting Tcl scripts.*

- **`step.tcl`:** Manages the current status of the design by monitoring checkpoints.
- **`synthesize.tcl`:** Manages all the details regarding the synthesis phase.

- `implement.tcl`: Manages all the details regarding the module implementation phase.
- `dfx_utils.tcl`: Manages all the details regarding the top-level implementation of a DFX design.
- `run.tcl`: Launches the actual runs for synthesis and implementation.
- `log_utils.tcl`: Handles report file creation at key points during the flow.

Remaining scripts provide details within these scripts (such as other `*_utils.tcl` scripts) or manage other Hierarchical Design flows (such as `hd_utils.tcl`).

Step 3: Synthesizing the Design

The `run_dfx.tcl` script automates the synthesis phase of this tutorial. Five iterations of synthesis are called, one for the static top-level design and one for each of four Reconfigurable Modules.

1. Open the AMD Vivado™ Tcl shell:
   - On Windows, select the Vivado desktop icon or select **Start → All Programs → Xilinx Design Tools → Vivado 2026.1 → Vivado 2026.1 Tcl Shell**.
   - On Linux, type `vivado -mode tcl`.
2. In the shell, navigate to `\led_shift_count_us`.
3. If you are using a target demonstration board other than the KCU105, modify the xboard variable in `run_dfx.tcl`. Valid alternatives are the VC707, VC709, and AC701 boards.
4. Run the `run_dfx.tcl` script by entering:

   ```
   source run_dfx.tcl -notrace
   ```

   After all five passes through Vivado synthesis are complete, the Vivado Tcl shell remains open. You can find log and report files for each module, alongside the final checkpoints, under each named folder in the Synth subdirectory

   TIP: *In `\led_shift_count_us`, multiple log files are created:*
   - `run.log` shows the summary as posted in the Tcl shell window
   - `command.log` echoes all the individual steps run by the script
   - `critical.log` reports all critical warnings produced during the run

   Note: The `command.log` file is itself a Tcl run script. This file can be modified if desired and sources as an input to reproduce the same results as an alternative to the more complex and parameterized Tcl_HD scripts.

Step 4: Assembling and Implementing the Design

Now that the synthesized checkpoints for each module, plus top, are available, you can assemble the design.

Run all flow steps from the Tcl Console, but you can use features within the IDE (such as the floorplanning tool) for interactive events.

TIP: *Copy and paste commands directly from the tutorial to avoid redundant effort and typos in the Vivado IDE. Copy and paste only one full command at a time. Some commands are long and span multiple lines.*

1. Open the Vivado IDE. You can open the IDE from the open Tcl shell by typing `start_gui` or by launching Vivado with the command `vivado -mode gui`.
2. Navigate to `\led_shift_count_us`, if you are not already there. The `pwd` command can confirm this.
3. Set variables that help with copying commands from this document into the Tcl Console. Select the part and board you are targeting for this lab, and apply them in Vivado:

```
set part "xcku040-ffva1156-2-e"
set board "kcu105"

set part "xcvu095-ffva2104-2-e"
set board "vcu108"

set part "xcku5p-ffvb676-2-e"
set board "kcu116"

set part "xcvu9p-flga2104-2l-e"
set board "vcu118"
```

4. Create an in-memory design by issuing the following command in the Tcl Console:

```
create_project -in_memory -part $part
```

5. Load the static design by issuing the following command:

```
add_files ./Synth/Static/top_synth.dcp
```

6. Load the top-level design constraints by issuing these commands:

```
add_files ./Sources/xdc/top_io_$board.xdc
set_property USED_IN {implementation}
[get_files ./Sources/xdc/top_io_$board.xdc]
```

   Selecting the top_io_$board version of the available xdc file loads the pin location and clocking constraints, but does not include floorplan information. The top_$board version includes pin location, clocking, and floorplanning constraints.

7. Load the first two synthesis checkpoints for the shift and count functions by issuing these commands:

```
add_files ./Synth/shift_right/shift_synth.dcp
set_property SCOPED_TO_CELLS {inst_shift}
[get_files ./Synth/shift_right/shift_synth.dcp]
add_files ./Synth/count_up/count_synth.dcp
set_property SCOPED_TO_CELLS {inst_count}
[get_files ./Synth/count_up/count_synth.dcp]
```

   The SCOPED_TO_CELLS property ensures that the proper assignment is made to the target cell. See the *Vivado Design Suite User Guide: Using Constraints* (UG903) for more information.

8. Link the entire design together using the `link_design` command:

```
link_design -mode default -reconfig_partitions {inst_shift inst_count}
-part $part -top top
```

   At this point a full configuration is loaded, including static and reconfigurable logic. Note that the Flow Navigator pane is not present while you are working in non-project mode.

   TIP: *Place the IDE in floorplanning mode by selecting **Layout → Floorplanning**. Make sure the Device window is visible.*

9. Save the assembled design state for this initial configuration:

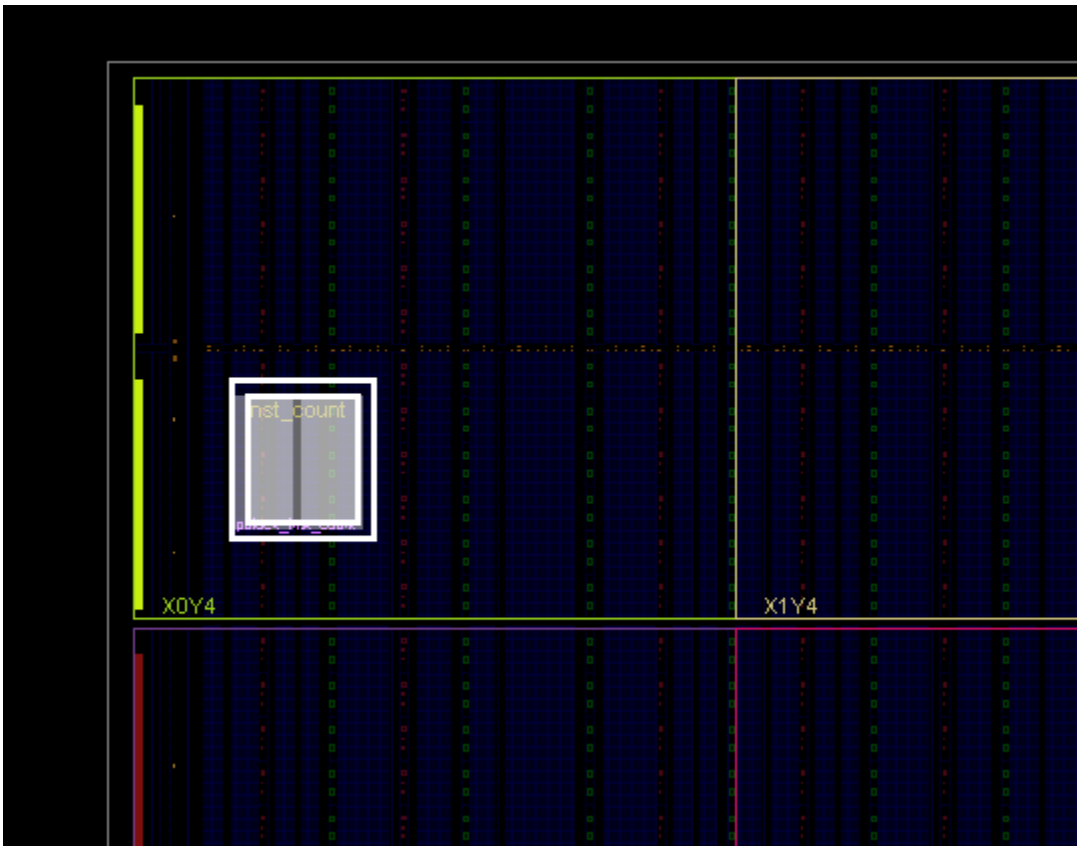
```
write_checkpoint -force ./Checkpoint/top_link_right_up.dcp
```

Step 5: Build the Design Floorplan

Next, create a floorplan to define the regions for Dynamic Function eXchange.

1. Select the inst_count instance in the Netlist window. Right-click and select **Floorplanning → Draw & Create Pblock** and draw a tall narrow box on the left side of the upper left corner of the device. The exact size and shape do not matter at this point, but keep the box within the clock region.

   Make sure that the Pblock is selected in the Device window before continuing.



Although this Reconfigurable Module only requires CLB resources, also include RAMB18, RAMB36, or DSP48 resources if the box encompasses those types. This allows the routing resources for these block types to be included in the reconfigurable region. The General view of the Pblock Properties window can be used to add these if needed. The Statistics view shows the resource requirements of the currently loaded Reconfigurable Module.

2. Repeat the previous step for the inst_shift instance, this time targeting clock region below the first. This Reconfigurable Module includes block RAM instances, so the resource type must be included. If omitted, the RAMB details in the Statistics view will be shown in red.
3. Run Dynamic Function eXchange Design Rule Checks by selecting **Reports → Report DRC**. You can uncheck **All Rules** and then check Dynamic Function eXchange to focus this report strictly on DFX DRCs.

   No DRC errors should be reported, as long as the inst_shift Pblock includes RAMB18 and RAMB36 resources. Advisory messages may still be reported, especially if the Pblock is located near the edge of the device. Note that for both Pblocks, SNAPPING_MODE is set to ON, as reported in the Properties view of the Pblock Properties window. This is always enabled for all UltraScale and UltraScale+ devices given the fine granularity of programmable units in this architecture.
4. Save these Pblocks and associated properties:

```
write_xdc ./Sources/xdc/top_all.xdc
```

This exports all the current constraints in the design, including those imported earlier from `top_io_$board.xdc`. These constraints can be managed in their own XDC file or managed within a run script (as is typically done with HD.RECONFIGURABLE).

Alternatively, the Pblock constraints themselves can be extracted and managed separately. A Tcl proc is available to help perform this task.

a. First source the proc which is found in one of the Tcl utility files:

```
source ./Tcl_HD/hd_utils.tcl
```

b. Then use the `export_pblocks` proc to write out this constraint information:

```
export_pblocks -file ./Sources/xdc/pblocks.xdc
```

This writes the Pblock constraint information for both Pblocks in the design. Use the `-pblocks` option to select only one if desired.

Step 6: Implementing the First Configuration

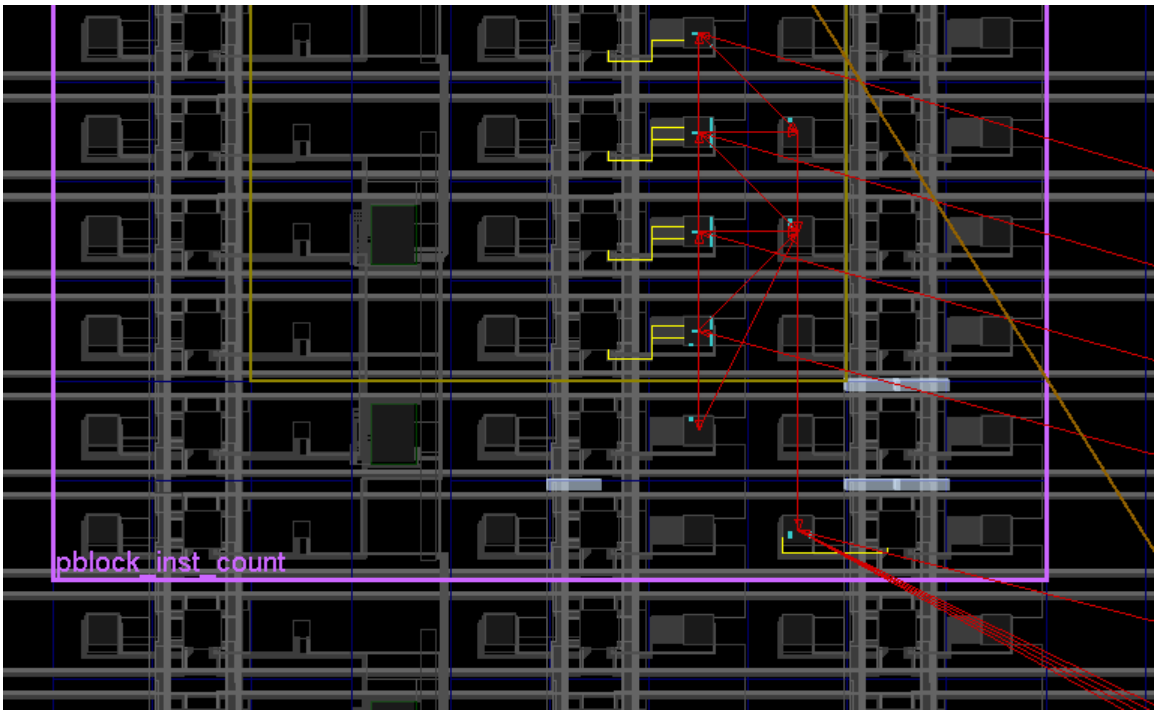
In this step, you place and route the design and prepare the static portion of the design for reuse with new Reconfigurable Modules.

Implementing the Design

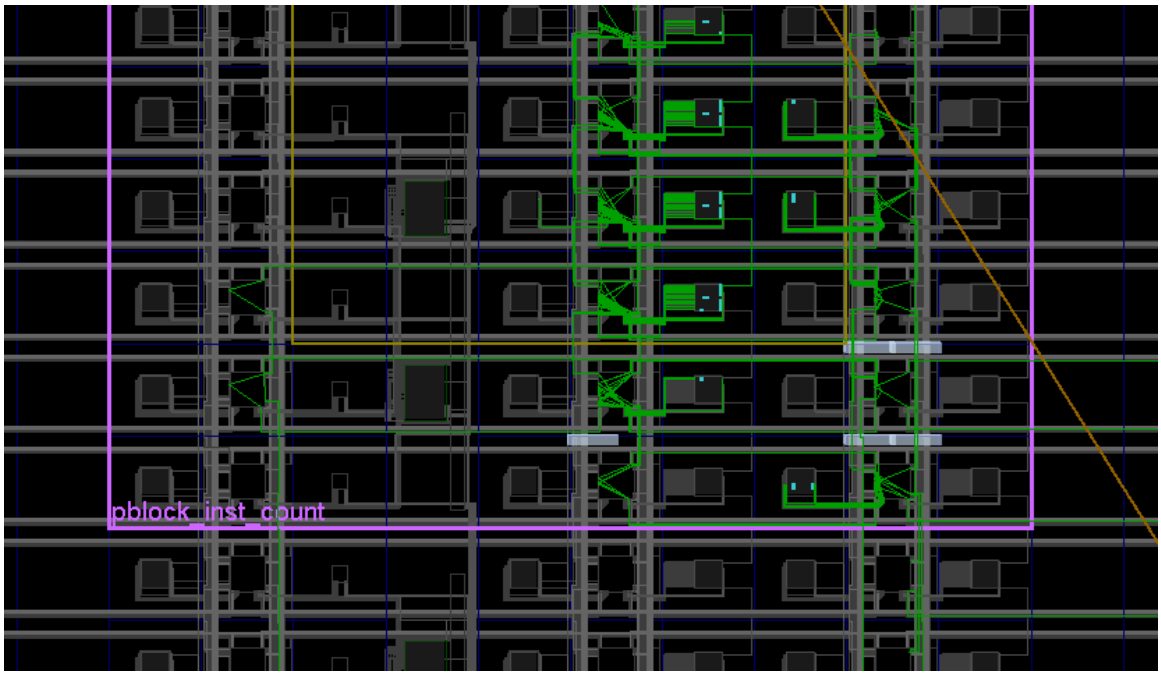
1. Optimize, place, and route the design by issuing the following commands:

```
opt_design
place_design
route_design
```

After both `place_design` and `route_design`, examine the state of the design in the Device view (see the following figure). One thing to note after `place_design` is the introduction of Partition Pins. These are the physical interface points between static and reconfigurable logic. They are anchor points within an interconnect tile through which each I/O of the Reconfigurable Module must route. They appear as white boxes in the placed design view. For `pblock_shift`, they appear in the top of that Pblock, as the connections to static are just outside the Pblock in that area of the device. For `Pblock_count`, they appear outside the user-defined region, as `SNAPPING_MODE` vertically collected more frames to be added to the Reconfigurable Partition.



2. To find these partition pins in the GUI easily:
   a. Select the Reconfigurable Module (for example, `inst_shift`) in the Netlist pane.
   b. Select the Cell Pins tab in the Cell Properties pane.
3. Select any pin to highlight it, or use Ctrl+A to select all. The Tcl equivalent of the latter is:
   ```
   select_objects [get_pins inst_shift/*]
   ```
4. Use the Routing Resources toolbar button to toggle between abstracted and actual routing information, and to change the visibility of the routing resources themselves. All nets in the design are fully routed at this point.



Saving the Results

1. Save the full design checkpoint and create report files by issuing these commands:

```
write_checkpoint -force Implement/Config_shift_right_count_up_implement/
top_route_design.dcp
report_utilization -file Implement/Config_shift_right_count_up_implement/
top_utilization.rpt
report_timing_summary -file Implement/
Config_shift_right_count_up_implement/top_timing_summary.rpt
```

2. [Optional] Save checkpoints for each of the Reconfigurable Modules by issuing these two commands:

```
write_checkpoint -force -cell inst_shift Checkpoint/
shift_right_route_design.dcp
write_checkpoint -force -cell inst_count Checkpoint/
count_up_route_design.dcp
```

> **TIP:** *When running `run_dfx.tcl` to process the entire design in batch mode, design checkpoints, log files, and report files are created at each step of the flow.*

   At this point, you have created a fully implemented Dynamic Function eXchange design from which you can generate full and partial bitstreams. The static portion of this configuration is used for all subsequent configurations. To isolate the static design, remove the current Reconfigurable Modules. Make sure routing resources are enabled, and zoom in to an interconnect tile with partition pins.

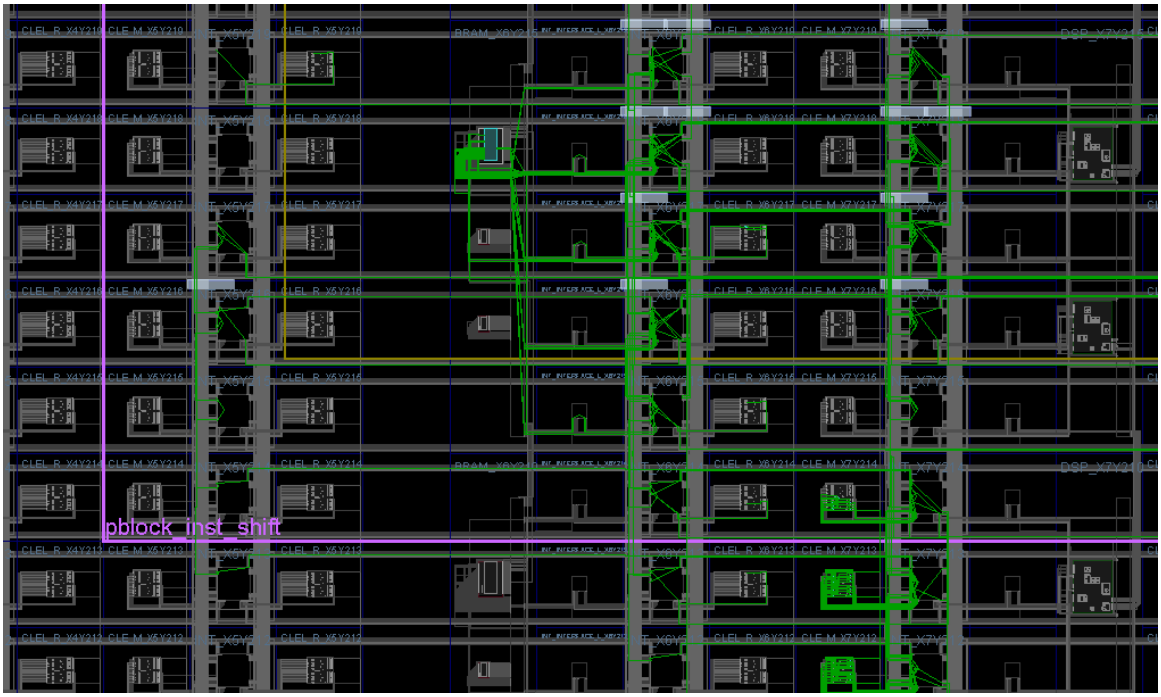
3. Clear out Reconfigurable Module logic by issuing the following commands:

```
update_design -cell inst_shift -black_box
update_design -cell inst_count -black_box
```

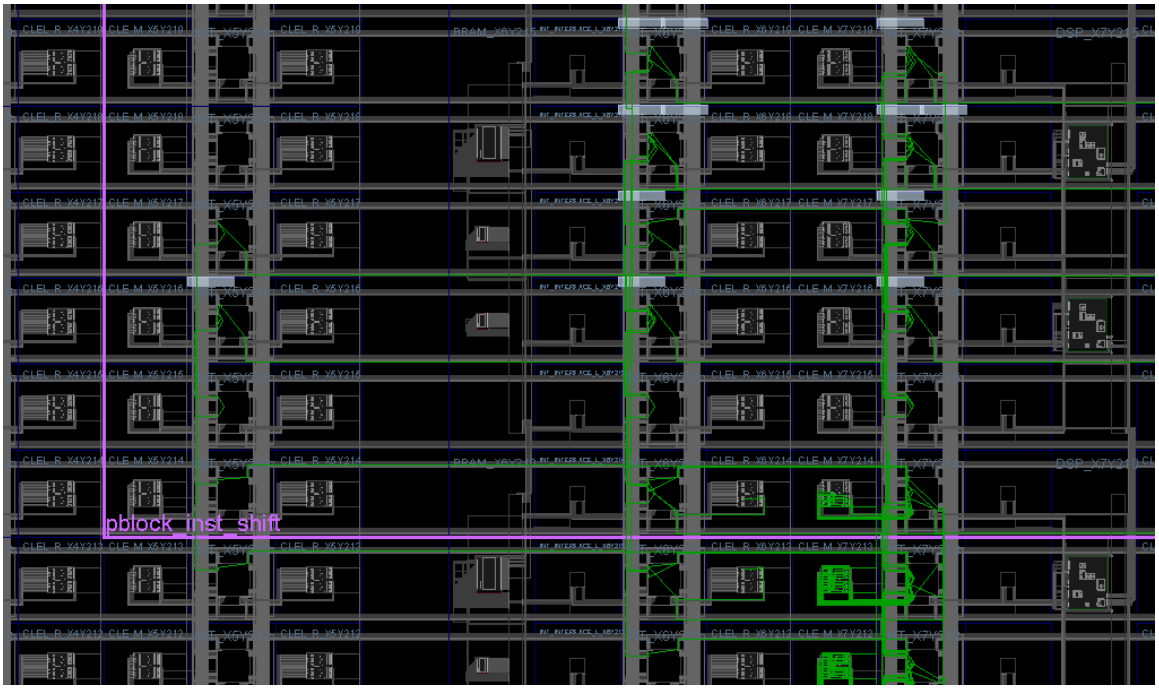
Issuing these commands results in many design changes as shown in the following figure:

- The number of Fully Routed nets (green) decreased.
- `inst_shift` and `inst_count` now appear in the Netlist view as empty.

The following figure shows the inst_shift module before `update_design -black_box`.



The following figure shows the inst_shift module after `update_design -black_box`.



1. Issue the following command to lock down all placement and routing:

   ```
   lock_design -level routing
   ```

   Because no cell was identified in the `lock_design` command, the entire design in memory (currently consisting of the static design with black boxes) is affected. All routed nets now display as locked, as indicated by dashed lines in Figure 18. All placed components changed from blue to orange to show they are also locked.

2. Issue the following command to write out the remaining static-only checkpoint:

   ```
   write_checkpoint -force Checkpoint/static_route_design.dcp
   ```

   This static-only checkpoint is used for future configurations.

3. Close this design before moving on to the next configuration:

   ```
   close_project
   ```

Step 7: Implementing the Second Configuration

Now that the static design result is established and locked, and you can use it as context for implementing further Reconfigurable Modules.

Implementing the Design

1. Create a new in-memory design by issuing the following command in the Tcl Console:

   ```
   create_project -in_memory -part $part
   ```

2. Load the static design by issuing the following command:

   ```
   add_files ./Checkpoint/static_route_design.dcp
   ```

3. Load the second two synthesis checkpoints for the shift and count functions by issuing these commands:

   ```
   add_file ./Synth/shift_left/shift_synth.dcp
   set_property SCOPED_TO_CELLS {inst_shift} [get_files ./Synth/shift_left/
   shift_synth.dcp]
   add_file ./Synth/count_down/count_synth.dcp
   set_property SCOPED_TO_CELLS {inst_count} [get_files ./Synth/count_down/
   count_synth.dcp]
   ```

4. Link the entire design together using the link_design command:

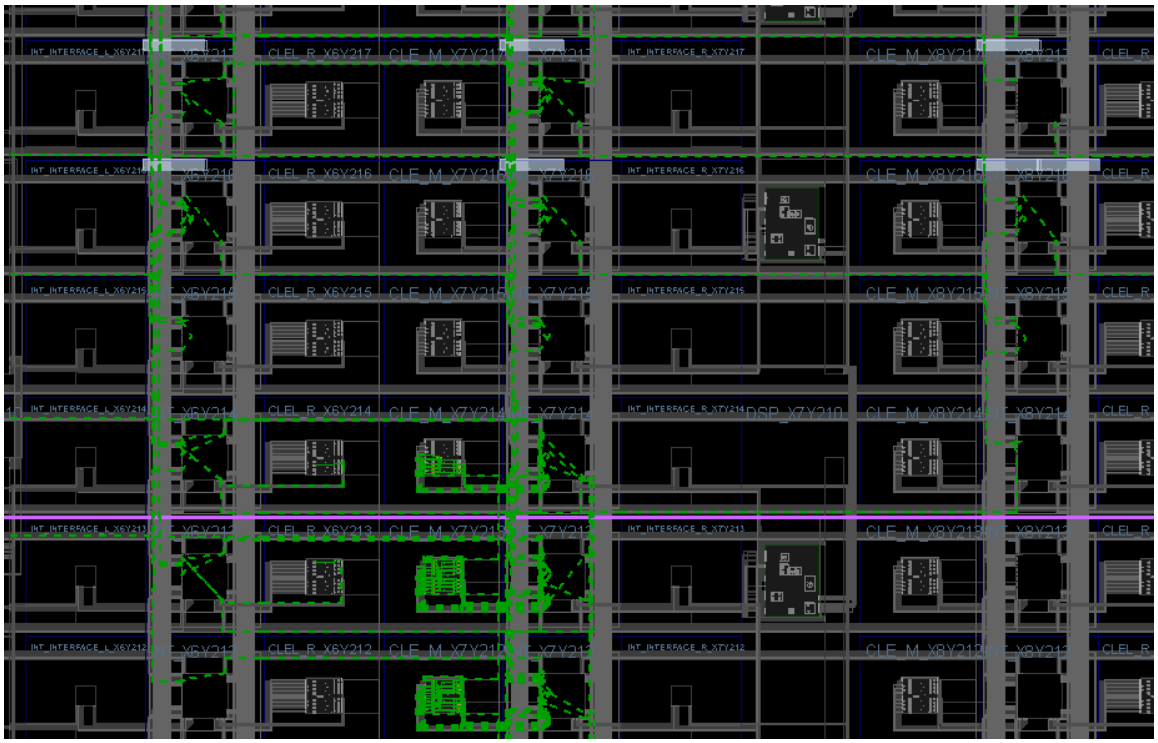
   ```
   link_design -mode default -reconfig_partitions {inst_shift inst_count}
   -part $part -top top
   ```

At this point, a full configuration is loaded. This time, however, the static design is routed and locked, and the reconfigurable logic is still just a netlist. Place and route from here only applies to the RM logic.

5. Optimize, place and route the new RMs in the context of static by issuing these commands:

```
opt_design
place_design
route_design
```

The design is again fully implemented, now with the new Reconfigurable Module variants. The routing is a mix of dashed (locked) and solid (new) routing segments, as shown below.



Saving the Results

1. Save the full design checkpoint and report files by issuing these commands:

```
write_checkpoint -force Implement/Config_shift_left_count_down_import/
top_route_design.dcp
report_utilization -file Implement/Config_shift_left_count_down_import/
top_utilization.rpt
report_timing_summary -file Implement/
Config_shift_left_count_down_import/top_timing_summary.rpt
```

2. [Optional] Save checkpoints for each of the Reconfigurable Modules by issuing these two commands:

```
write_checkpoint -force -cell inst_shift Checkpoint/
shift_left_route_design.dcp
write_checkpoint -force -cell inst_count Checkpoint/
count_down_route_design.dcp
```

At this point, you have implemented the static design and all Reconfigurable Module variants. This process would be repeated for designs that have more than two Reconfigurable Modules per Reconfigurable Partition.

Step 8: Examine the Results with Highlighting Utilities

With the routed configuration open in the IDE, use the `get_dfx_footprint` utility to highlight placement and routing footprints for the reconfigurable partitions. This utility is available for all architectures newer than 7 series.

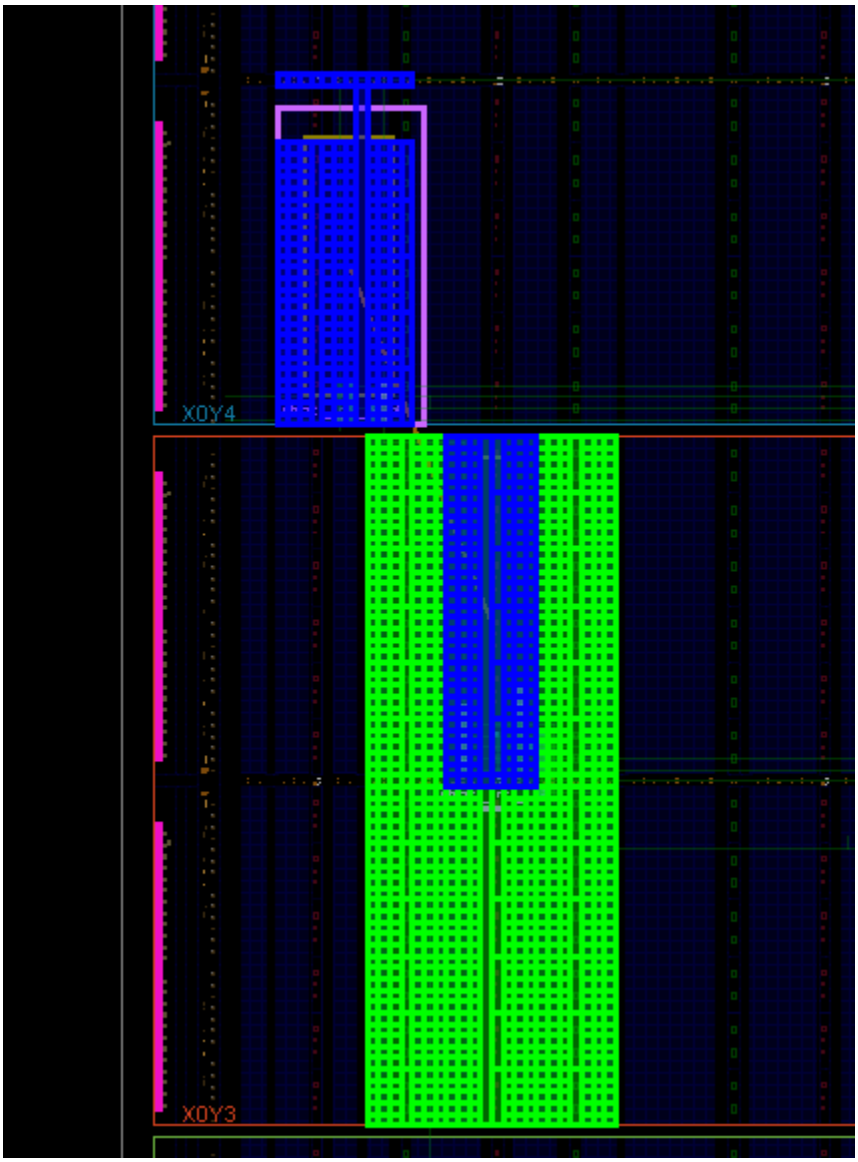
1. In the Tcl Console, issue the following commands from the `\led_shift_count_us` directory:

```
highlight_objects -color green [get_dfx_footprint -route -of_objects
[get_cells inst_shift]]
```

2. Next, issue the following commands:

```
highlight_objects -color blue [get_dfx_footprint -place -of_objects
[get_cells inst_shift]]

highlight_objects -color blue [get_dfx_footprint -place -of_objects
[get_cells inst_count]]
```

The partition frames appear highlighted in the device view, as shown in the following figure.



These highlighted tiles represent the configuration frames used for placement (blue) and routing (green) for each RM. The green tiles are sent to bitstream generation to create the partial bitstream (for inst_shift). The SNAPPING_MODE feature adjusted three of four edges of pblock_shift to account for alignment to programmable unit boundaries. This snapping behavior explains why it appears that static logic might appear to have been placed inside Reconfigurable Partitions, as seen in prior steps. In actuality, the effective boundary is one CLB row higher than the user-defined Pblock boundary indicates, so this static logic is placed correctly. This effective boundary can also be seen in the shading of the Pblock during creation, as shown in Step 5: Build the Design Floorplan.

***Note*:** RCLK rows matching the width of the Pblocks are included. Global clocks driving logic in these Reconfigurable Partitions are connected to the spines running through these rows and are enabled or disabled during Dynamic Function eXchange.

3. Close the current design:

```
close_project
```

Step 9: Generating the Bitstreams

Verifying the Configurations

RECOMMENDED: *Before generating bitstreams, verify all configurations to ensure that the static portion of each configuration match identically, so the resulting bitstreams are safe to use in silicon. The PR Verify feature examines the complete static design up to and including the partition pins, confirming that they are identical. Placement and routing within the Reconfigurable Modules is not checked, as different module results are expected here.*

1. Run the `pr_verify` command from the Tcl Console:

```
pr_verify Implement/Config_shift_right_count_up_implement/
top_route_design.dcp Implement/Config_shift_left_count_down_import/
top_route_design.dcp
```

If successful, this command returns the following message.

```
INFO: [Vivado 12-3253] PR_VERIFY: check points
Implement/Config_shift_right_count_up/top_route_design.dcp and Implement/
Config_shift_left_count_down/top_route_design.dcp are compatible
```

By default, only the first mismatch (if any) is reported. To see all mismatches, use the `-full_check` option.

2. Close the project:

```
close_project
```

Generating Bitstreams

Now that the configurations have been verified, you can generate bitstreams and use them to target your selected demonstration board.

***Note*:** The first configuration implements `shift_right` and `count_up`. The second configuration implements `shift_left` and `count_down`.

1. Read the first configuration into memory:

```
open_checkpoint Implement/Config_shift_right_count_up_implement/
top_route_design.dcp
```

2. Generate full and partial bitstreams for this design. Be sure to keep the bit files in a unique directory related to the full design checkpoint from which they were created.

```
write_bitstream -force -file Bitstreams/Config_RightUp.bit
close_project
```

Notice that five (or three, if you are using an AMD UltraScale+™ device) bitstreams have been created:

- `Config_RightUp.bit` This is the power-up, full design bitstream. The four shift LEDs on the right will shift right and the four count LEDs on the left will count up.
- `Config_RightUp_pblock_inst_shift_partial.bit` This is the partial bit file for the shift_right module that causes the shift LEDs to shift right.
- `Config_RightUp_pblock_inst_count_partial.bit` This is the partial bit file for the count_up module that causes the count LEDs to count up.
- `Config_RightUp_pblock_inst_shift_partial_clear.bit` This is the clearing bit file for the shift_right module for UltraScale devices only. It safely clears a right shift to allow the shift module to be reconfigured.
- `Config_RightUp_pblock_inst_count_partial_clear.bit` This is the clearing bit file for the count_up module for UltraScale devices only. It safely clears an up count to allow the count module to be reconfigured.



IMPORTANT! *When generated by a single call to `write_bitstream`, the names of the bit files currently do not reflect the name of the Reconfigurable Module variant to clarify which image is loaded. The current solution uses the base name given by the `-file` option and appends the Pblock name of the reconfigurable cell. It is critical to provide enough description in the base name to be able to identify the reconfigurable bit files clearly. All partial bit files have the `_partial` postfix, and all clearing bit files have the `_partial_clear` postfix.*

Using `run_dfx.tcl` to process the entire design through bitstream generation, uses a different technique for generating the bitstreams. Opening a routed design checkpoint issues multiple calls to `write_bitstream`, which gives you more control over naming bitstreams and allows for different options (such a bitstream compression) to be applied to full versus partial bitstreams. For example, the names configured in the `run_dfx.tcl` script are:

- `Config_shift_right_count_up_implement_full.bit` This is the power-up, full design bitstream.
- `pblock_shift_shift_right_partial.bit` This is the partial bit file for the `shift_right` module.
- `pblock_count_count_up_partial.bit` This is the partial bit file for the `count_up` module.
- `pblock_shift_shift_right_partial_clear.bit` This is the clearing bit file for the `shift_right` module for UltraScale devices only.
- `pblock_count_count_up_partial_clear.bit` This is the clearing bit file for the `count_up` module for UltraScale devices only.

1. Generate full and partial bitstreams for the second configuration, again keeping the resulting bit files in the appropriate folder.

```
open_checkpoint Implement/Config_shift_left_count_down_import/
top_route_design.dcp
write_bitstream -force -file Bitstreams/Config_LeftDown.bit
close_project
```

   Similarly, five (or three) bitstreams are created, this time with a different base name.

2. Generate a full bitstream with grey boxes, plus blanking bitstreams for the Reconfigurable Modules. Blanking bitstreams can be used to "erase" an existing configuration to reduce power consumption.

   ***Note*:** Note: Grey box blanking bitstreams are not the same as clearing bitstreams. Clearing bitstreams are required to prepare the global signal mask for the next partial bitstream, ensuring the GSR event occurs properly.

```
open_checkpoint Checkpoint/static_route_design.dcp
update_design -cell inst_count -buffer_ports
update_design -cell inst_shift -buffer_ports
place_design
route_design
write_checkpoint -force Checkpoint/config_grey_box.dcp
write_bitstream -force -file Bitstreams/config_grey_box.bit
close_project
```

   The base configuration bitstream has no logic for either reconfigurable partition. The `update_design` commands here insert constant drivers (ground) for all outputs of the Reconfigurable Partitions, so these outputs do not float. The term grey box indicates that the modules are not completely empty with these LUTs inserted, as opposed to black boxes, which would have dangling nets in and out of this region. The `place_design` and `route_design` commands ensure they are completely implemented. As valid Reconfigurable Modules, note that these instances also have clearing bitstreams for UltraScale devices only.

Step 10: Partially Reconfiguring the FPGA

The `led_shift_count` design targets one of four demonstration boards. The current design supports the KCU105, VCU108, KCU116, and VCU118 boards, revisions Rev 1.0 and newer.

Configuring the Device with a Full Image

1. Connect the board to your computer using the Platform Cable USB and power on the board.
2. From the main Vivado IDE, select **Flow → Open Hardware Manger**.
3. Select **Open target** on the green banner. Follow the steps in the wizard to establish communication with the board.
4. Select **Program device** on the green banner and pick the target device, e.g., xcku040_0.

5. Navigate to the Bitstreams folder to select **Config_RightUp.bit**, then click **OK** to program the device.

   You should now see the bank of GPIO LEDs performing two tasks. Four LEDs are performing a counting-up function (MSB is on the left), and the other four are shifting to the right. Note the amount of time it took to configure the full device.

Partially Reconfiguring the Device

At this point, you can partially reconfigure the active device with any of the partial bitstreams that you have created, starting first with the appropriate clearing bitstream.

1. UltraScale targets only: Select **Program device** on the green banner again. Navigate to the Bitstreams folder to select **Config_RightUp_pblock_inst_shift_partial_clear.bit**, then click **OK** to program the device.

   The shift portion of the LEDs stopped, but the counter kept counting up, unaffected by the reconfiguration. Note the much shorter configuration time, as well as the fact that the DONE LED turned off.

2. Select Program device on the green banner again. Navigate to the Bitstreams folder to select **Config_LeftDown_pblock_inst_shift_partial.bit**, then click **OK** to program the device.

   The shift portion of the LEDs restarted in the opposite direction, and the DONE LED is back on.

3. UltraScale targets only: Select Program device on the green banner again. Navigate to the Bitstreams folder to select **Config_RightUp_pblock_inst_count_partial_clear.bit**, then click **OK** to program the device.

   The count portion of the LEDs stopped, but the shifter kept shifting, unaffected by the reconfiguration.

4. Select Program device on the green banner again. Navigate to the Bitstreams folder to select **Config_LeftDown_pblock_inst_count_partial.bit**, then click **OK** to program the device.

   The counter is now counting down, and the shifting LEDs were unaffected by the reconfiguration. This process can be repeated with the Config_RightUp partial bit files to return to the original configuration, or with the blanking partial bit files to stop activity on the LEDs (they will stay on). Keep track of the currently loaded module for each partition to ensure the correct clearing bitstream is loaded before the next partial bitstream.

Lab 2 Conclusion

This concludes Lab 2. In this lab, you:

- Synthesized a design bottom-up to prepare for Dynamic Function eXchange implementation
- Created a valid floorplan for a Dynamic Function eXchange design

- Created two configurations with common static results
- Implemented these two configurations, saving the static design to be used in each
- Created checkpoints for static and reconfigurable modules for later reuse
- Examined framesets and verified the two configurations
- Created full and partial bitstreams
- Created and used clearing bitstreams for AMD UltraScale™ devices
- Configured and partially reconfigured an FPGA

Lab 3

DFX RTL Project Flow

Overview

In this lab, you create a new project and set up all the sources and runs defining the structure of a DFX design. The design used in this lab is based on the simple design in labs 1 and 2, but modified so it has two instances of the shift module instead of one shift and one count. This shows the implications of Partition Definitions in the project flow.

This lab currently targets the following AMD development platforms:

- KCU116 (AMD Kintex™ UltraScale+™ devices)
- VCU118 (AMD Virtex™ UltraScale+™ devices)
- KCU105 (AMD Kintex™ UltraScale™ devices)
- VCU108 (AMD Virtex™ UltraScale™ devices)
- KC705 (AMD Kintex™ 7 devices)
- VC707 (AMD Virtex™ 7 devices)
- VC709 (AMD Virtex™ 7 devices)

Step 1: Extract the Tutorial Design Files

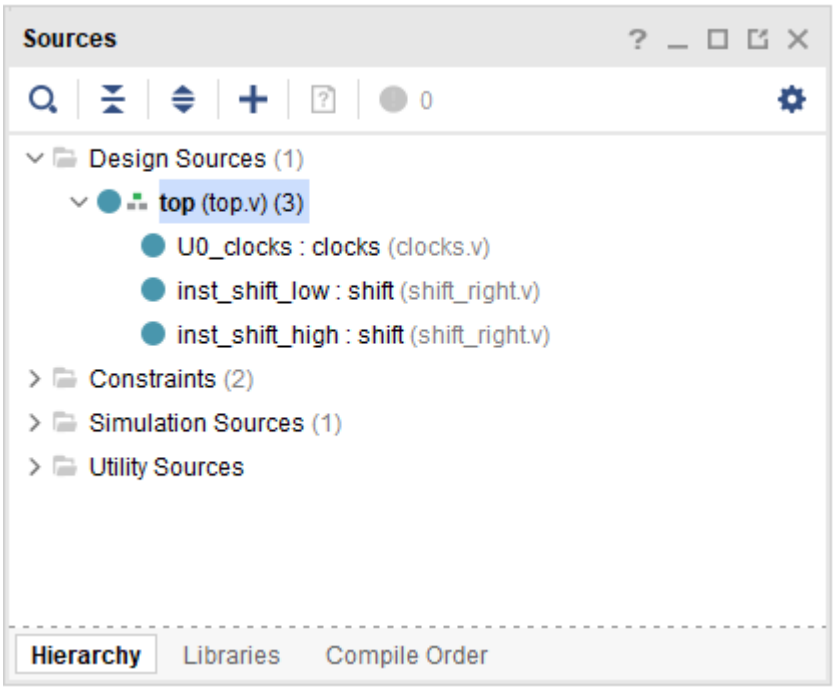
1. Download the reference design files.
2. Extract the zip file contents to any write-accessible location.
3. In the extracted files, navigate to `\led_shift_count_7s`.

Step 2: Load Initial Design Sources

The first unique step in any DFX design flow (project based or otherwise) is to define the parts of the design that is marked reconfigurable. This is done via context menus in the Hierarchical Source View in project mode. These steps walk through initial project creation through definition of partitions in a simple design.

1. Extract the design from the archive. The `dfx_project` data directory is referred to in this tutorial as the `<Extract_Dir>`.
2. Open the AMD Vivado™ IDE and select **New Project**.
3. Select the `<Extract_Dir>` as the Project location. Leave the Project name as `project_1`, and leave the Create project subdirectory option checked. Click **Next**.
4. Select **RTL Project** and ensure the Do not specify sources at this time checkbox is unchecked, then click **Next**.
5. Click the **Add Directories** button and select these Sources directories to be added to the design:
   - `<Extract_Dir>\Sources\hdl\top`
   - `<Extract_Dir>\Sources\hdl\shift_right`
6. Click **Next** to get to the Add Constraints page, then select these files:
   - `<Extract_Dir>\Sources\xdc\top_io_<board>.xdc`
   - `<Extract_Dir>\Sources\xdc\pblocks_<board>.xdc`

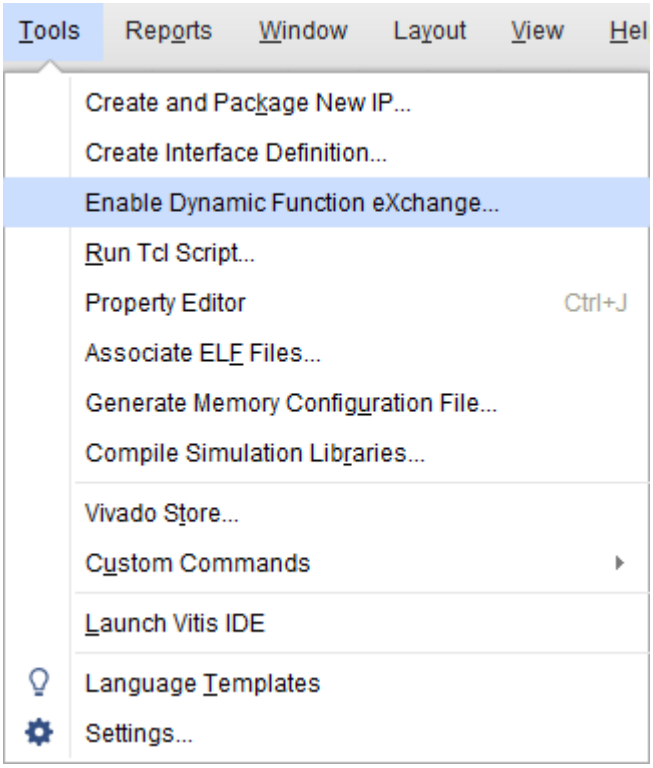
   ***Note*:** These constraint files are full design constraints, scoped to the top level design. If you would like to perform your own floorplanning, only select the `top_io_<board>.xdc`, omitting the `pblocks_<board>.xdc`. Stop the flow after synthesis to create your own floorplan.
7. Click **Next** to choose the part. In the Default Part page, click **Boards** and choose the target board matching the constraint file(s) you have selected. Then click **Next** and then **Finish** to complete project creation. The Sources window shows a standard hierarchical view of the design.



At this point, a standard project is open. Nothing specific to Dynamic Function eXchange has been done.

8. Select **Tools → Enable Dynamic Function eXchange**.

   This prepares the project for the DFX design flow. After this is set it cannot be undone, so archive your project before selecting this option.



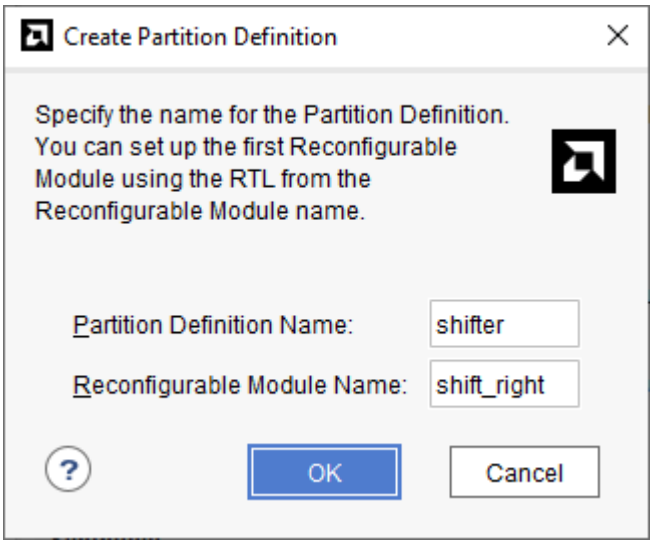
In the ensuing dialog box, click **Convert** to turn this project into a DFX project.

9. Right-click on one of the *shift* instances and select the **Create Partition Definition** option.

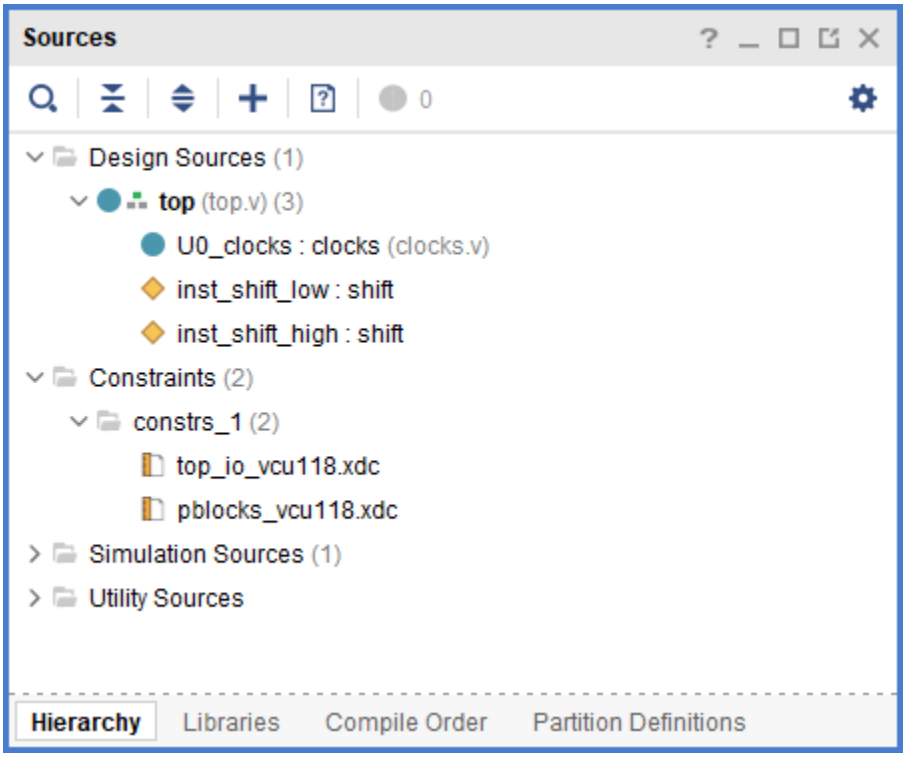
   This action defines *both* shift instances as Reconfigurable Partitions in the design. Because each instance has come from the same RTL source, they are logically identical. Out-of-context synthesis runs to keep this module separated from top, and the one post-synthesis checkpoint is used for both shift instances.

   > **TIP:** *If there are multiple instantiations of the same module within a design, but not all need to be reconfigurable, then the modules must be manually modified to become unique. Then you can independently tag desired instances as Reconfigurable Partitions.*

10. In the dialog box that appears, give names to both the Partition Definition and the Reconfigurable Module. The Partition Definition is the general reference for the workspace into which all Reconfigurable Modules are inserted, so give it an appropriate name, such as `shifter`. The Reconfigurable Module refers to this specific RTL instance, so give it a name that references its functionality, such as `shift_right`, then click **OK**.



The Sources window has now changed slightly, with both shift instances now shown with a yellow diamond, indicating they are Partitions. You also see a Partition Definitions tab in this window, showing the list and contents of all Partition Definitions (one at this point) in the design. In addition, an out-of-context module run has been created for synthesizing the shift_right module.



At this point, more Reconfigurable Module sources can be added. This is done via the Dynamic Function eXchange Wizard.

IMPORTANT! *After Partitions have been defined, all additional RMs must be added via the DFX Wizard, and any management of RM sources, configurations, and runs must also be done via this wizard.*

Step 3: Completing the Design with the Dynamic Function eXchange Wizard

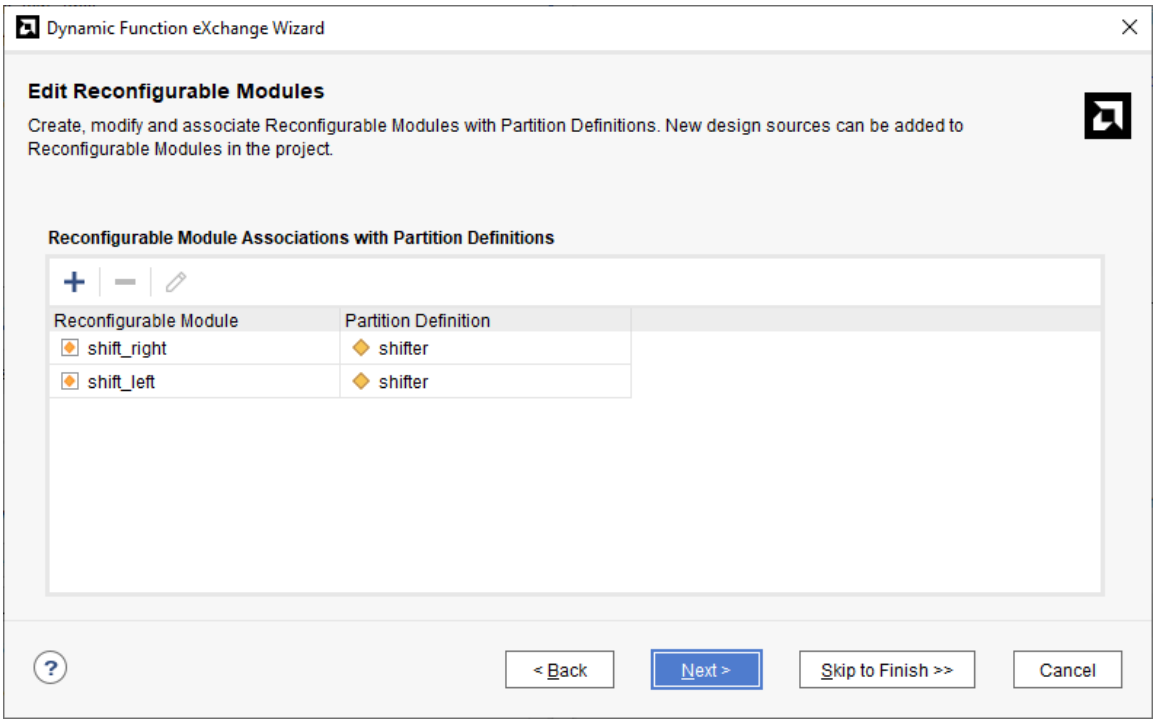
1. Launch the Dynamic Function eXchange Wizard by selecting this option under the Tools menu or from the Flow Navigator.
2. Click **Next** to get to the Edit Reconfigurable Modules page. Here you can see the shift_right RM already exists, and there are add, remove and edit buttons on the left hand side of the window, above the RMs. Click on the blue + icon to add a new RM.
3. Click the **Add Directories** button to select the shift_left folder:

   ```
   <Extract_Dir>\Sources\hdl\shift_left
   ```

   Or use the **Add Files** button to select the shift_left.v file residing in this directory. If module-level constraints were needed, they would be added here and would need to be scoped to the level of hierarchy for this Partition.

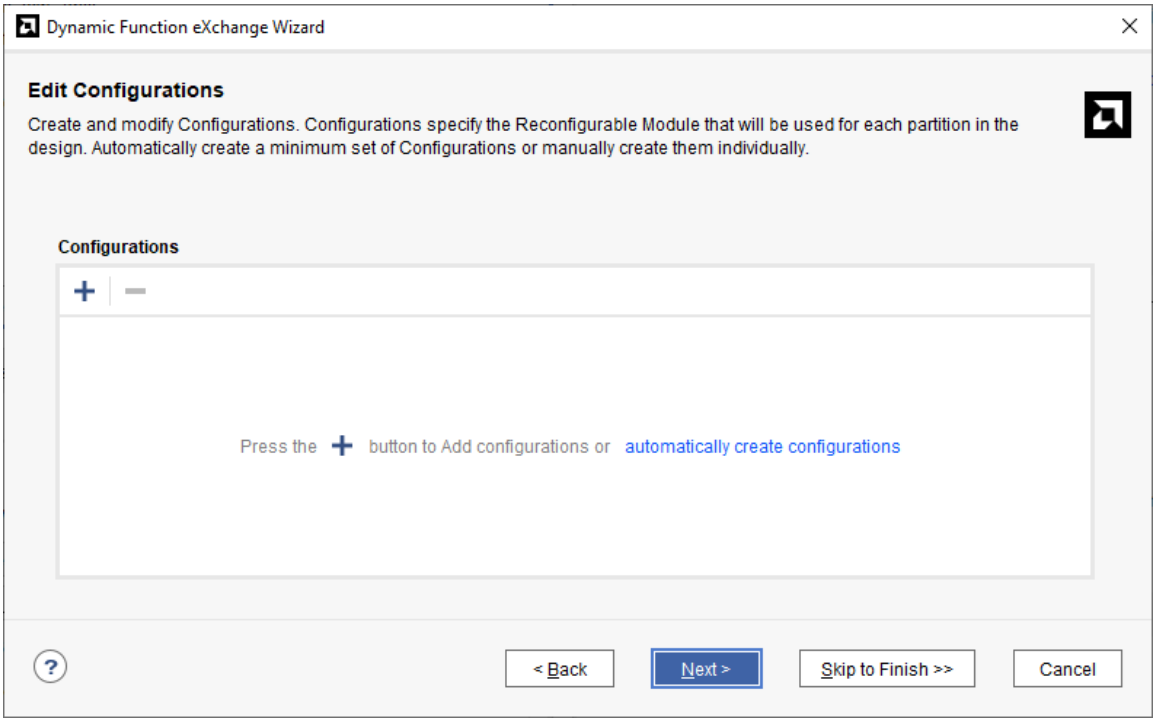
   Fill in the Reconfigurable Module field to be `shift_left`. Set the Partition Definition to be `shifter`, leave Top Module field empty and the Sources are already synthesized check box unchecked. Click **OK** to create the new module.

   Two Reconfigurable Modules are now available for the shifter Reconfigurable Partition. Click **Next** to continue.



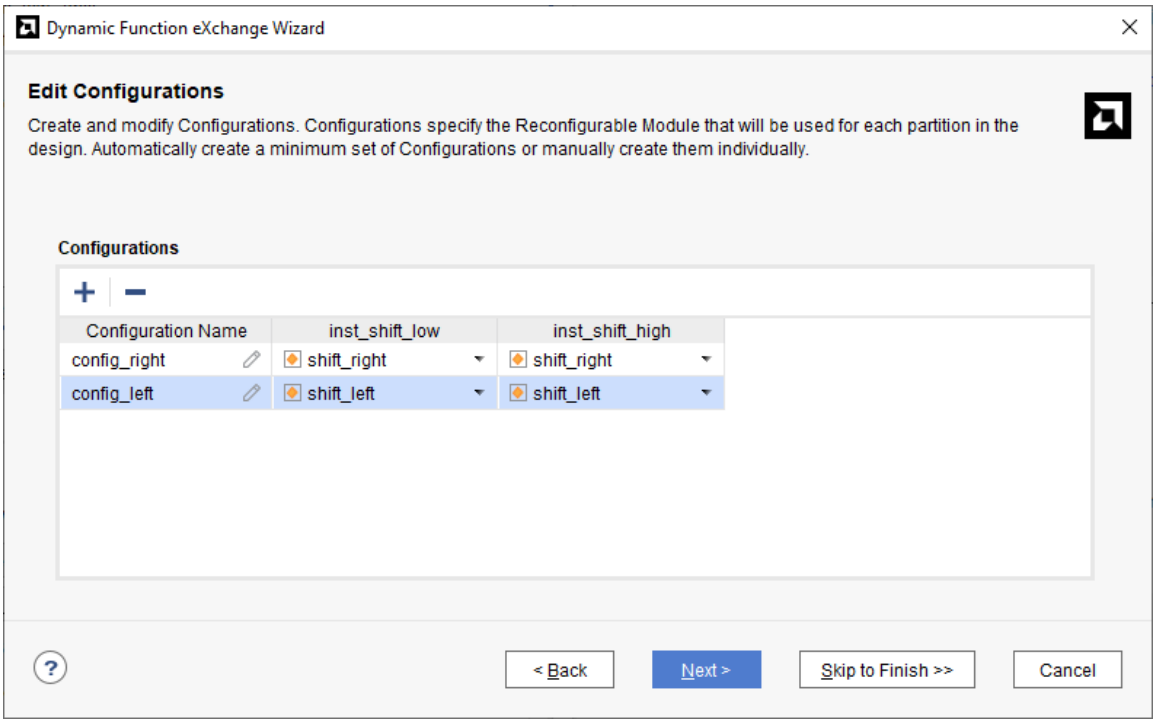
On the next page, Configurations are defined. Configurations are full design images consisting of the static design and one RM per RP. You can either create any desired set of configurations, or simply let the wizard select them for you.

4. Select the **automatically create configurations** link to let the Wizard create the configurations.



After selecting this option, the minimum set of two configurations is created. Each shift instance is given `shift_right` in the first configuration and `shift_left` in the second configuration.

Note: The Configuration Name is editable. In the example below, the names are updated to `config_right` and `config_left` to reflect the Reconfigurable Modules contained within each one.



Additional configurations can be created by using these two Reconfigurable Modules, but two is all you need to create all the partial bitstreams necessary for this version of the design, as the maximum number of RMs for any RP is two (not including greybox RMs).

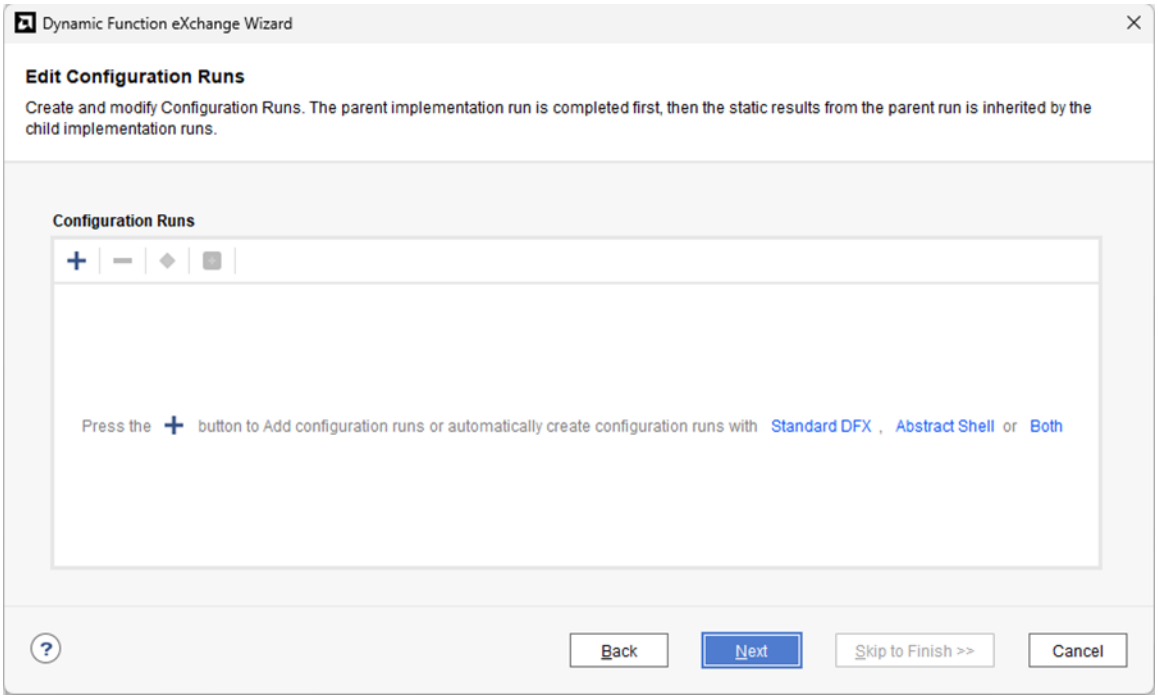
5. Click **Next** to get to the Edit Configuration Runs page.

   As with configurations themselves, the runs used to implement each configuration can be automatically or manually created. A parent-child relationship will define how the runs interact – the parent run implements the static design and all RMs within that configuration, then child runs reuse the locked static design while implementing the RMs within that configuration in that established context.

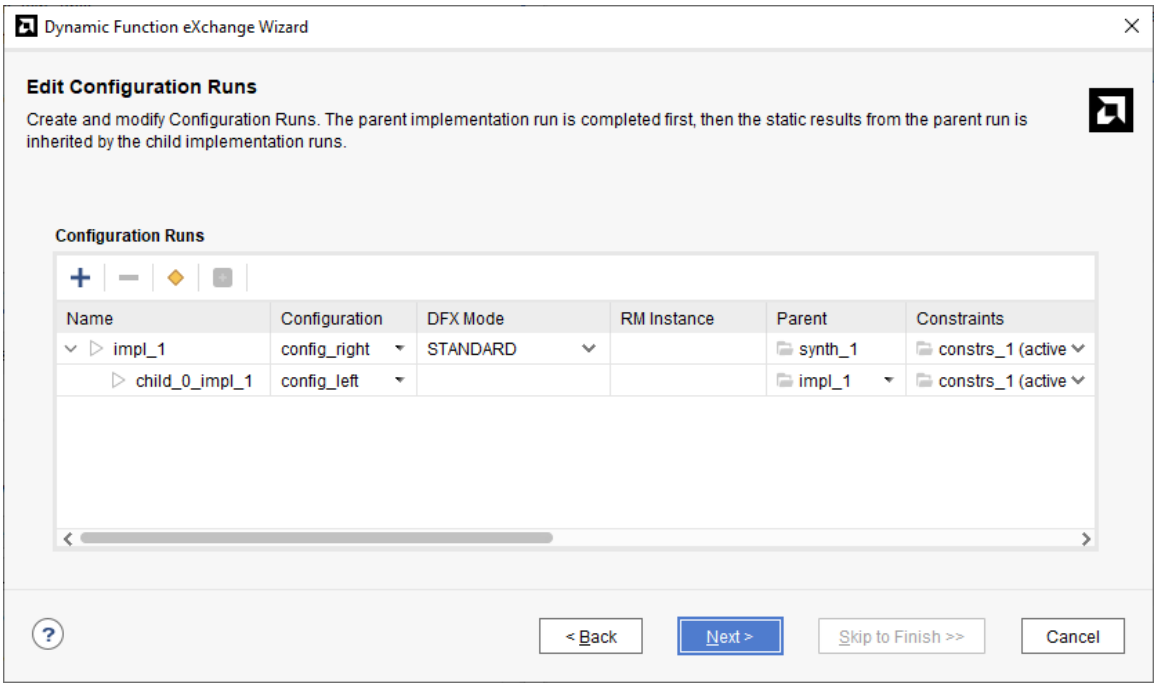
6. If targeting UltraScale+ devices, click the **Standard DFX** link to populate the Configuration Runs page with the minimum set of runs. Abstract Shell configuration runs are covered in a different lab.

   If targeting 7 series or UltraScale devices, click **automatically create configuration runs** to populate the Configuration Runs page with the minimum set of runs. The Abstract Shell flow is not available for these architectures.

   Note: The following figures show a design that targets an UltraScale+ device.



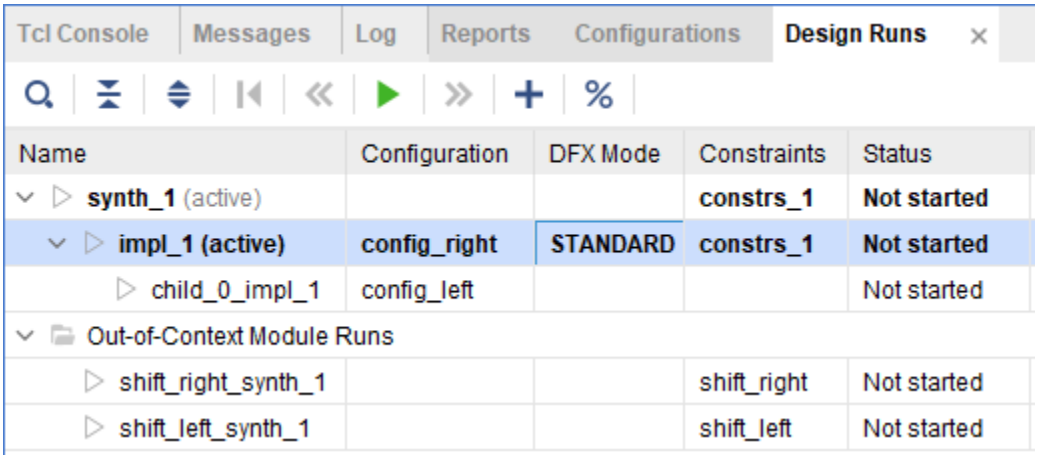
This creates two runs, consisting of one parent configuration (`config_right`) and one child configuration (`config_left`). Any number of independent or related runs can be created within this wizard, with options for using different strategies or constraint sets for any of them. For now, leave this set to the two runs set here. The names of the runs are not editable.



7. Click **Next** to see the Summary page, and click **Finish** to complete the design setup and exit the Wizard.

   IMPORTANT! *Nothing is created or modified until you click Finish to exit the DFX Wizard. All actions are queued until this last click, so it is possible to step forward and back as needed without implementing changes until you are ready.*

   Back in the AMD Vivado™ IDE, you will see that the Design Runs window has been updated. A second out-of-context synthesis run has been added for the shift_left RM, and a child implementation run (child_0_impl_1) has been created under the parent (impl_1). You are now ready to process the design.



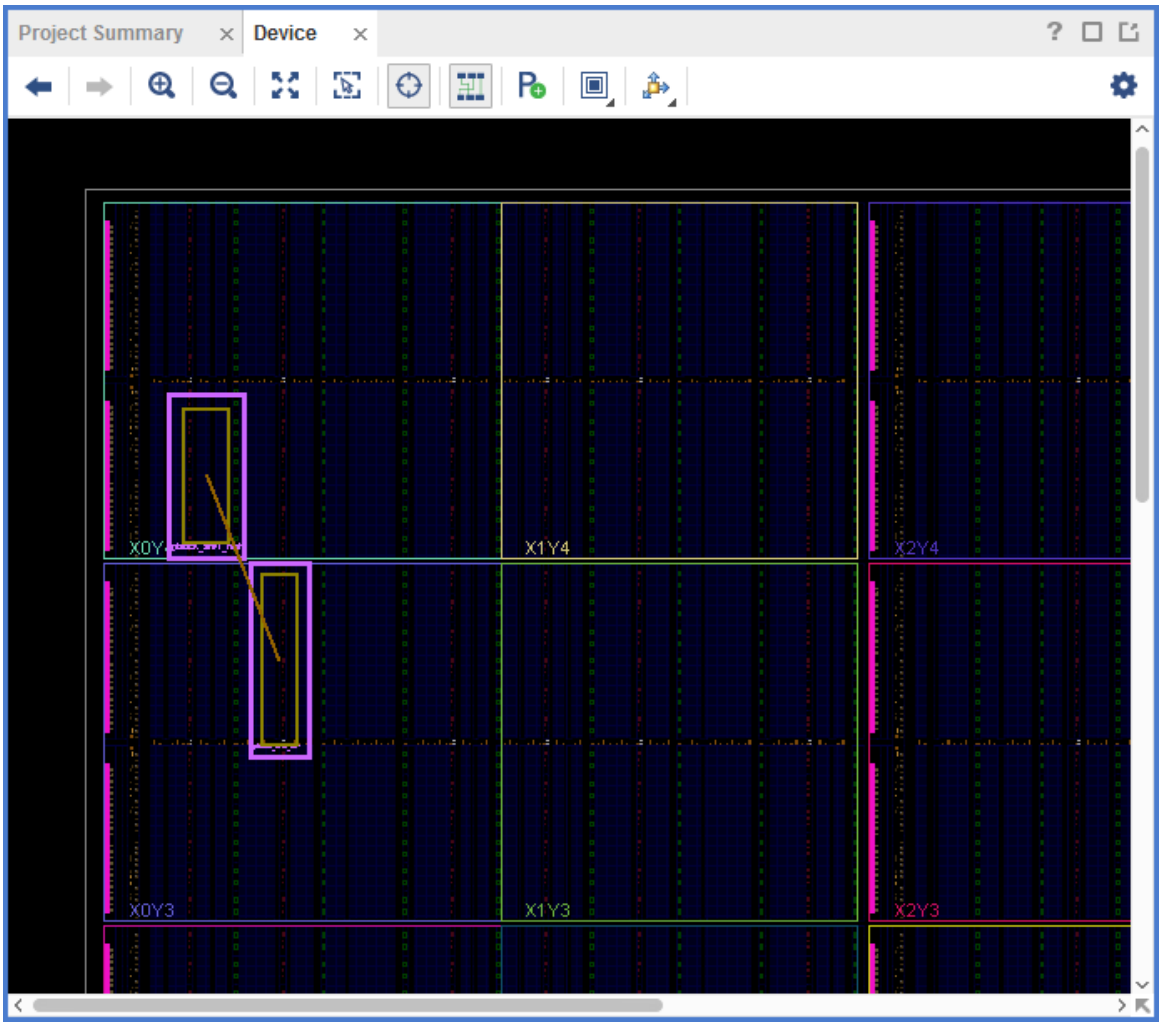
Step 4: Synthesizing and Implementing the Current Design

With the design from above open in the AMD Vivado™ IDE, examine the Design Runs window. The top-level design synthesis run (synth_1) and the parent implementation run (impl_1) are marked “active.” The Flow Navigator actions apply to these active runs and their child runs, so clicking on Run Synthesis or Run Implementation pulls the design through only these runs, as well as the OOC synthesis runs needed to complete them. You can select a specific parent or child implementation run, right-click and select Launch Runs to pull through the entire flow for that ultimate target.

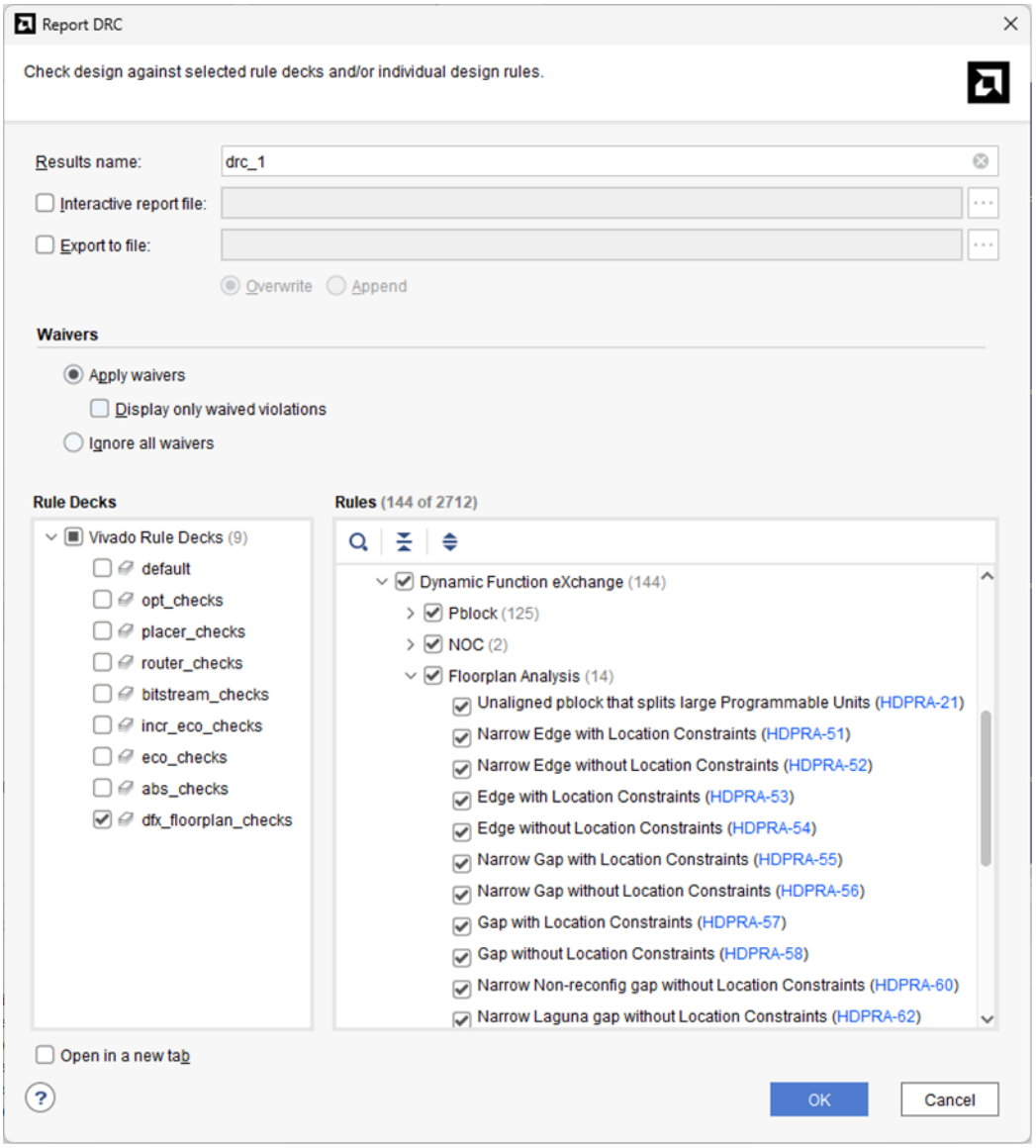
1. In the Flow Navigator, click **Run Synthesis**. Click **OK** to begin synthesis. When synthesis completes, select **Open Synthesized Design**, and click **OK** to open the synthesized design.

   This action synthesizes all OOC modules, followed by synthesis of the top level design. This is no different than any design with OOC modules (IP or otherwise).

In the post-synthesis design that opens, two Pblocks are already defined. These were supplied in `pblocks_<board>.xdc` and map to the two shift instances in top. If no Pblocks had existed with the design sources, they could be created at this step in the flow. This can be done by right-clicking on an inst_shift instance in the design hierarchy to select **Floorplanning → Draw Pblock**. Each instance requires its own unique Pblock.



2. Select one of the two Pblocks in the floorplan and note its properties. The last two properties listed are RESET_AFTER_RECONFIG (7 series only) and SNAPPING_MODE, two properties specific to DFX. Note that both of these options have been enabled in the Pblocks xdc.
3. Run DFX-specific design rule checks by selecting **Reports → Report DRC**. To save time, you can deselect all checkboxes other than the one for Dynamic Function eXchange.



DRC checks report no errors with the supplied sources and constraints. Advisory messages may be given for certain devices with suggestions on how to improve the quality of the given Pblocks. These can be ignored for this simple design.

If you created your own floorplan and DRCs were reported, fix the issues before moving on. Note that both modules will require block RAM resources, and remember that SNAPPING_MODE will resolve any errors related to horizontal or vertical alignment.

TIP: *Run DFX Design Rule Checks early and often.*

4. In the Flow Navigator, select **Run Implementation** to run place and route on all configurations.

This action runs implementation first for impl_1 and then for child_0_impl_1. Behind the scenes, Vivado takes care of all the details. In addition to running place and route for the two runs with all the DFX requirements in place, it does a few more tasks specific to DFX. After impl_1 completes, Vivado automatically:

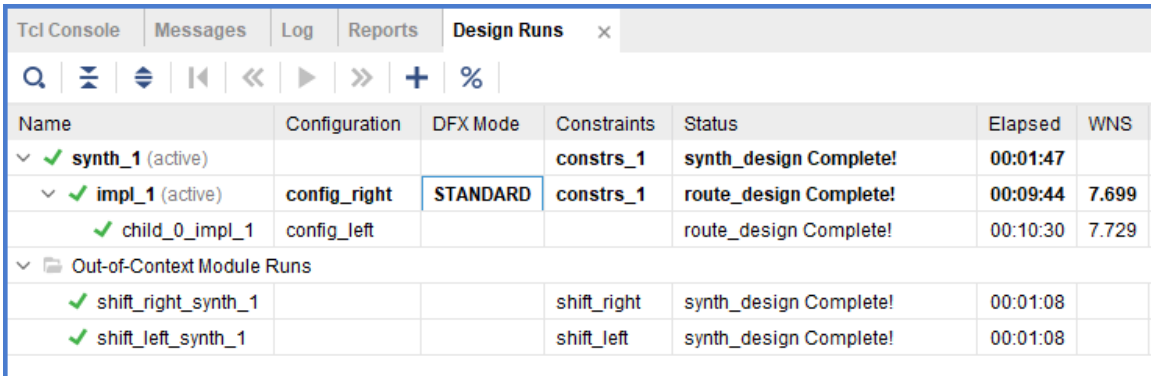
- Writes module-level (OOC) checkpoints for each routed shift_right RM.
- Carves out the logic in each RP to create a static-only design image for the top. This is done by calling `update_design -black_box` for each instance.
- Locks all placement and routing for the static-only portion of the design. This is done by calling `lock_design -level` routing.
- Saves the locked static parent image to be reused for all child runs.

In addition, when the child run completes, module-level checkpoints are created for the routed shift_left RMs. A locked static design image would be identical to the parent, so this step is not necessary.

If you want only specific configuration runs, you can individually select the runs within the Design Runs window. A parent run must be completed successfully before a child run can be launched, as the child run starts with the locked static design from the parent.

5. When Implementation completes, click **Cancel** in the resulting pop-up dialog.

> **CAUTION!** *Even though the design has been processed through to the child implementation run, selecting Open Implemented Design opens the parent run by default. Use the pulldown selection to choose the desired implementation run.*



Name	Configuration	DFX Mode	Constraints	Status	Elapsed	WNS
synth_1 (active)			**constrs_1**	**synth_design Complete!**	**00:01:47**	
impl_1 (active)	**config_right**	**STANDARD**	**constrs_1**	**route_design Complete!**	**00:09:44**	**7.699**
child_0_impl_1	config_left			route_design Complete!	00:10:30	7.729
Out-of-Context Module Runs						
shift_right_synth_1			shift_right	synth_design Complete!	00:01:08	
shift_left_synth_1			shift_left	synth_design Complete!	00:01:08	

At this point, there are two steps remaining. The first is running PR Verify to compare the two configurations to ensure consistency of the static part of the design images. This step is highly recommended and occurs automatically within the Vivado project. The second step is to generate the bitstreams themselves.

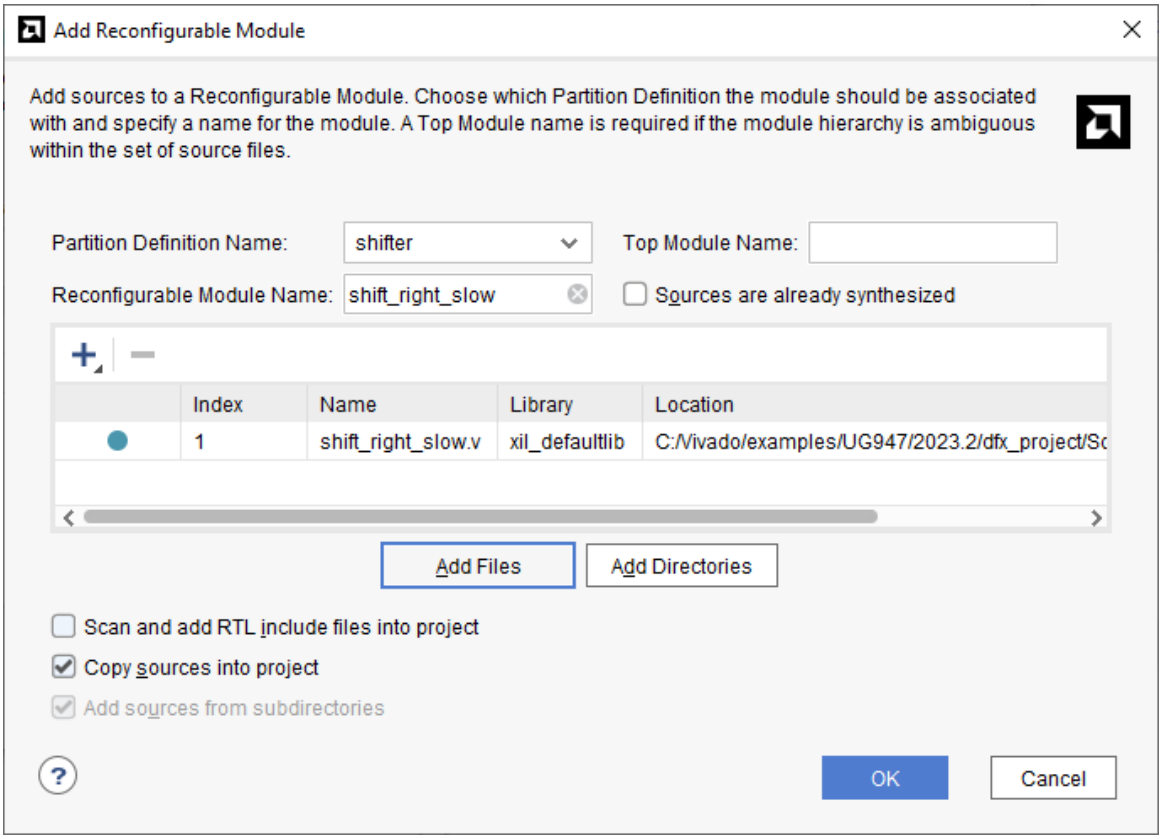
6. In the Flow Navigator, click **Generate Bitstream**. This action launches bitstream generation on the active parent run, and launches PR Verify and then bitstream generation on all implemented child runs.

   For each configuration run, both full and partial bitstreams are generated by default.

The entire Dynamic Function eXchange flow can be run in a project environment. All steps, from module-level synthesis to bitstream generation can be done without leaving the GUI.

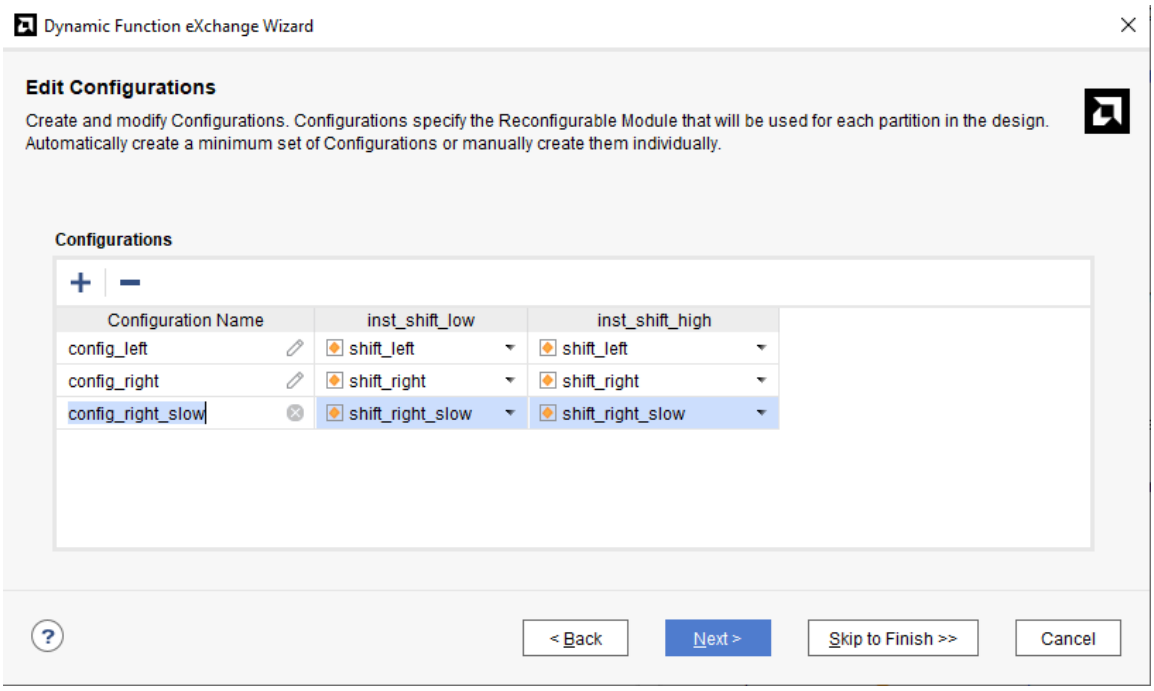
Step 5: Adding an Additional Reconfigurable Module and Corresponding Configuration

1. With the design open in the Vivado IDE, open the Dynamic Function eXchange Wizard.
2. On the Edit Reconfigurable Modules page, click the **+** button to add a new RM.
3. Select the `shift_right_slow.v` file in `<Extract_Dir>\Sources\hdl\shift_right_slow`, and click **OK**.
4. Enter `shift_right_slow` for the Reconfigurable Module name, click **OK**, and click **Next**.



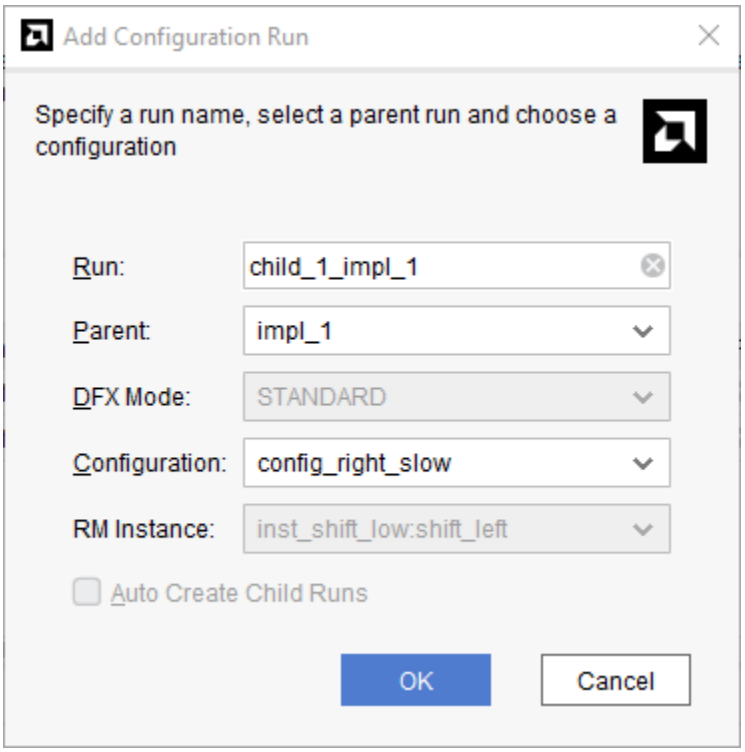
In the Edit Configurations page, there is no longer an option to automatically create configurations, as you already have two existing configurations. You can re-enable this option by removing all existing configurations, but this will recreate all configurations and remove all existing results.

5. To create a new configuration, click the **+** button, enter the name `config_right_slow`, and press **Enter**. Select **shift_right_slow** for each Reconfigurable Partition instance.



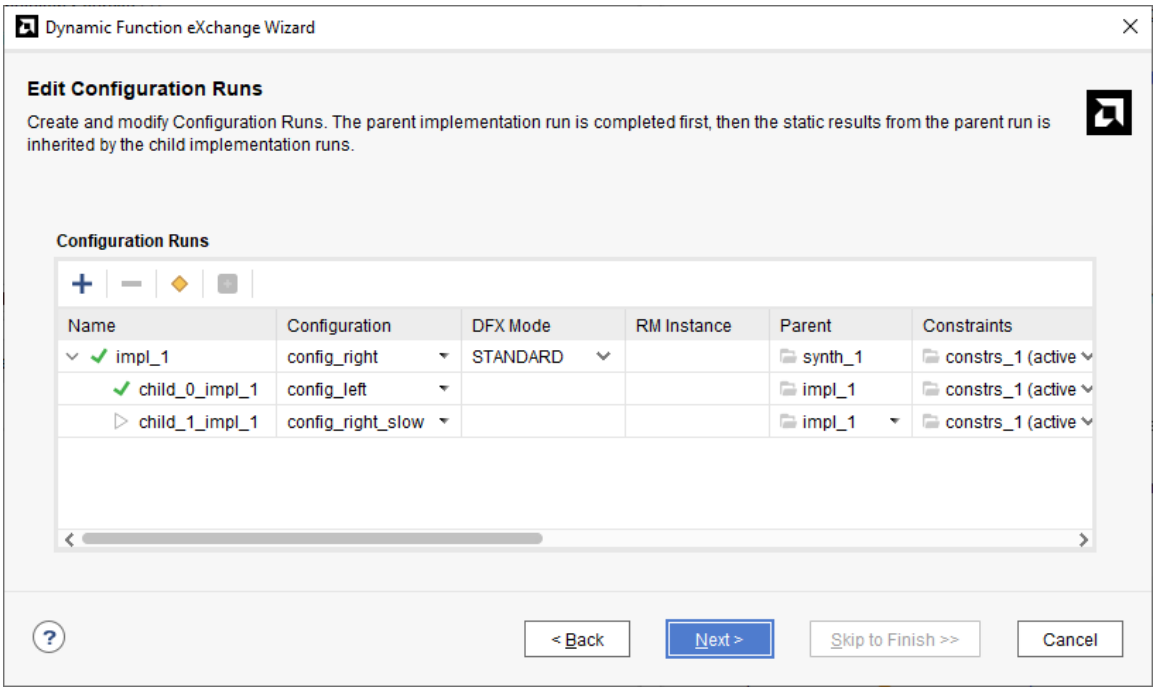
6. Click **Next** to advance to the Configuration Runs. Use the **+** button to create a new configuration with these properties:
   - **Run:** `child_1_impl_1` simply matches the existing convention
   - **Parent:** `impl_1` makes this configuration a child run of the existing parent run
   - **Configuration:** `config_right_slow` is the one with the new RMs just defined

   ***Note*:** The remaining options cannot be edited because they are not appropriate in this context.

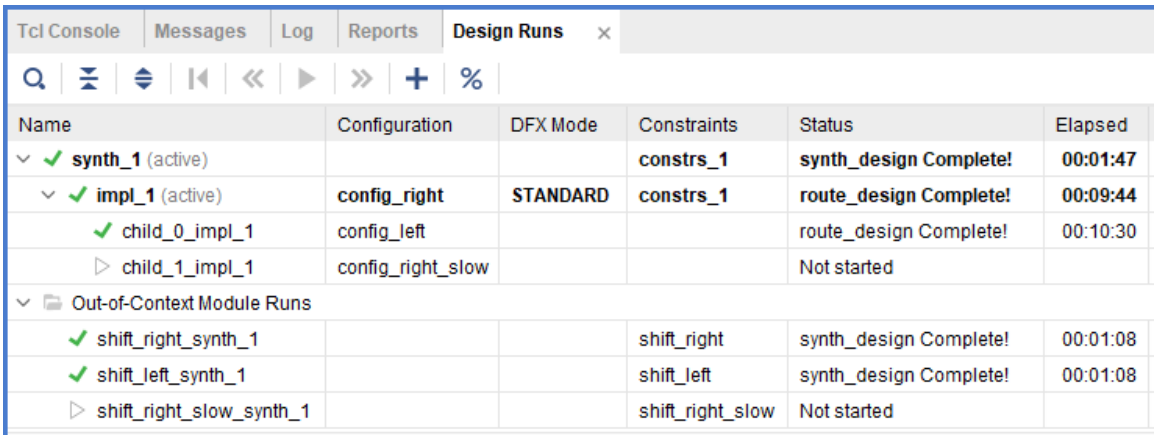


7. Click **OK** to add the new Configuration Run.

   This new configuration, as a child of the existing `impl_1`, reuses the static design implementation results like `config_left` did. Three runs now exist, with two as children of the initial parent. The green check marks indicate that two of the runs are currently complete.



8. Click **Next**, and click **Finish** to build this new configuration run.



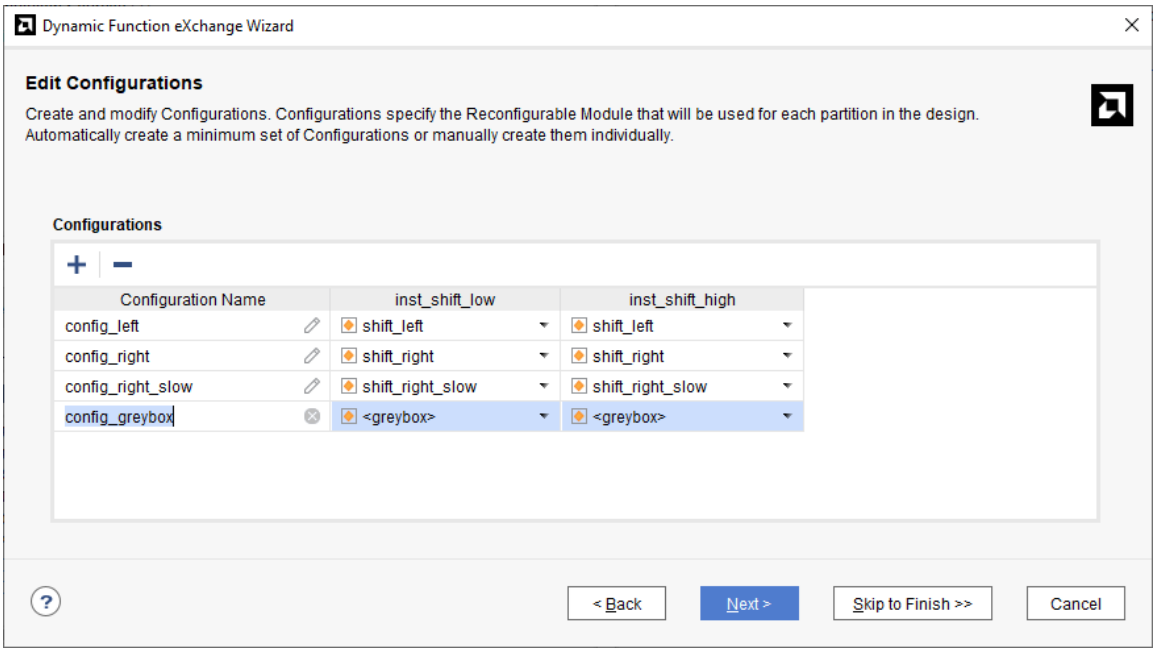
9. Select this new child implementation run, right-click, and select **Launch Runs**. This runs the OOC synthesis on the `shift_right_slow` module and then implements this module within the context of the locked static design.

Step 6: Creating and Implementing a Greybox Module

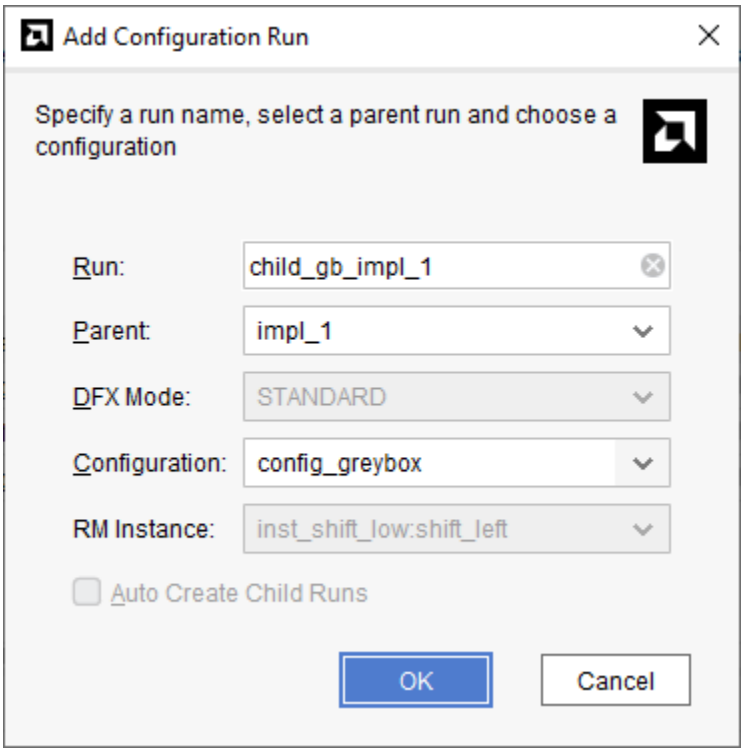
For some designs, the desired initial configuration of the device might be an image with no function resident in a Reconfigurable Partition. A greybox configuration can be created as a child run.

A greybox is a module that starts off as a blackbox, but then has LUTs automatically inserted for all ports during implementation. Output ports are driven to a logic 0 (by default; 1 is selectable via property) so they do not float. This module allows the design to be processed even if no RMs are available. Although a configuration with greybox RMs can be the parent run, this is only recommended when no other RMs exist and/or when budgeting constraints are used to optimize the RP interface placement. Therefore, this capability is not permitted in the Vivado IDE. It is recommended that a greybox configuration is a child configuration, where the parent run is the most difficult configuration or uses representative functionality within each RP to produce the highest quality static design possible.

1. Open the Dynamic Function eXchange (DFX) Wizard and move to the Configurations page. No new Reconfigurable Modules need to be defined in this case, because this is a dedicated feature. Create a new configuration with the name of `config_greybox`, and enter `<greybox>` for each Reconfigurable Partition instance.

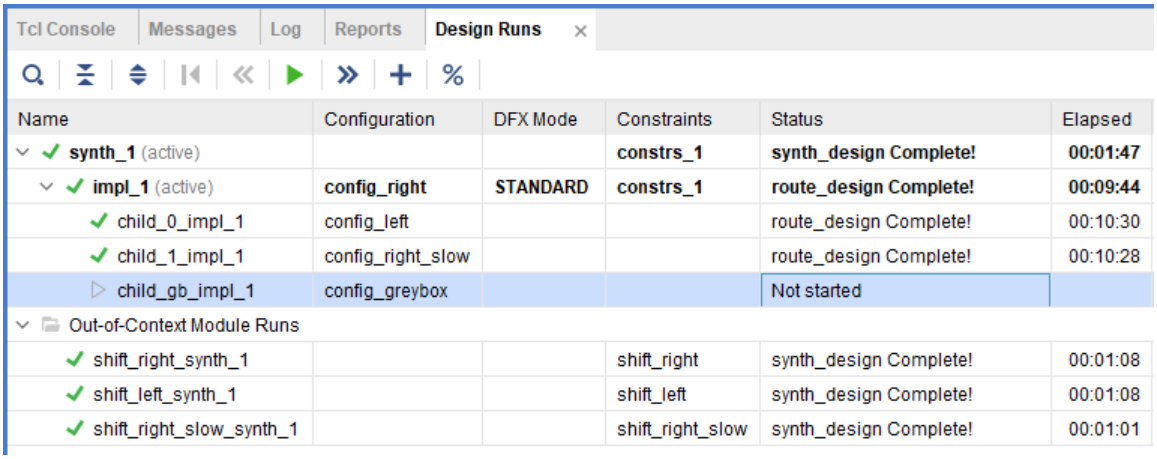


2. Click **Next** to get to the Configuration Runs page, then create another new configuration run, this time for the greybox configuration.
   - **Run:** `child_gb_impl_1`
   - **Parent:** `impl_1` makes this configuration run a new child run, starting from the existing parent run
   - **Configuration:** `config_greybox` the RMs consist only of LUT tie-offs
3. Click **OK** to create this new configuration run.



4. Click **Next**, and click **Finish** to create this new run.

   Now there are four implementation runs and three out-of-context runs shown in the Design Runs window. The greybox module does not require synthesis. It is an embedded feature in the DFX solution.



Name	Configuration	DFX Mode	Constraints	Status	Elapsed
synth_1 (active)			**constrs_1**	**synth_design Complete!**	**00:01:47**
impl_1 (active)	**config_right**	**STANDARD**	**constrs_1**	**route_design Complete!**	**00:09:44**
child_0_impl_1	config_left			route_design Complete!	00:10:30
child_1_impl_1	config_right_slow			route_design Complete!	00:10:28
child_gb_impl_1	config_greybox			Not started	
Out-of-Context Module Runs					
shift_right_synth_1			shift_right	synth_design Complete!	00:01:08
shift_left_synth_1			shift_left	synth_design Complete!	00:01:08
shift_right_slow_synth_1			shift_right_slow	synth_design Complete!	00:01:01

   At this point the greybox configuration can be implemented.

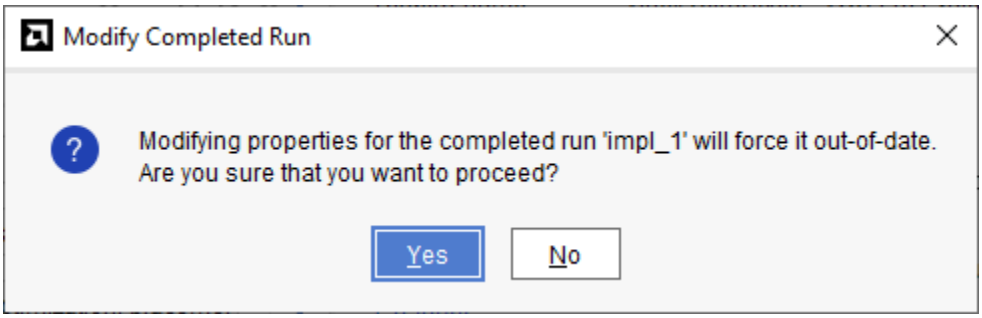
5. Select the `impl_greybox design` run, right-click, and select **Launch Runs** to implement this child configuration.

Step 7: Modifying a Design Source or Options

The Vivado IDE tracks dependencies between design runs. This is a critical feature for Dynamic Function eXchange given the interdependencies of configurations. If any aspect of the parent configuration or implementation results are modified, it and all children must be recompiled.

1. Select the impl_1 design run.
2. In the Options tab of the Run Properties window, change the Strategy to Performance_Explore.

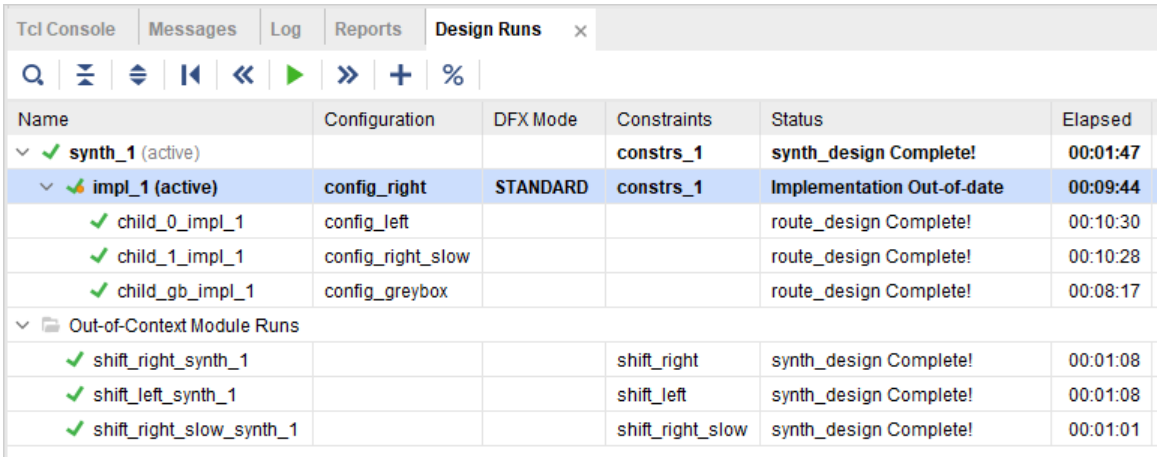
   A pop-up dialog will alert you to the fact that impl_1 will be forced out of date if you proceed.



3. Click **Yes**.

   The parent run impl_1 is now marked out-of-date. The resulting files still exist in their respective folders, but will be deleted as soon as implementation is launched.

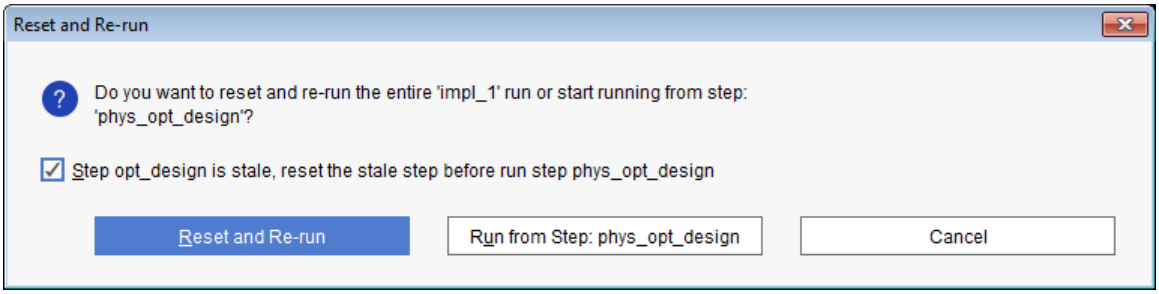
   ***Note*:** The Strategy option for each of the child runs remains at Vivado Implementation Defaults; child runs do not inherit options from the parent run. However, any strategy or option in child runs will only have an effect on the Reconfigurable Module implementation, as the static design is already routed and locked.



Name	Configuration	DFX Mode	Constraints	Status	Elapsed
synth_1 (active)			**constrs_1**	**synth_design Complete!**	**00:01:47**
impl_1 (active)	**config_right**	**STANDARD**	**constrs_1**	**Implementation Out-of-date**	**00:09:44**
child_0_impl_1	config_left			route_design Complete!	00:10:30
child_1_impl_1	config_right_slow			route_design Complete!	00:10:28
child_gb_impl_1	config_greybox			route_design Complete!	00:08:17
Out-of-Context Module Runs					
shift_right_synth_1			shift_right	synth_design Complete!	00:01:08
shift_left_synth_1			shift_left	synth_design Complete!	00:01:08
shift_right_slow_synth_1			shift_right_slow	synth_design Complete!	00:01:01

4. In the Flow Navigator click **Run Implementation**.

A dialog will appear to confirm if you want to reset runs before continuing. Because the stale step is the first step in the parent run, the first options completely reset both parent and all child runs to the beginning of implementation. Click either Reset and Re-run or Run from Step: phys_opt_design to continue.



This implements all four runs. First, the parent impl_1 run will complete, then the three child runs will run in parallel.

Lab 3 Conclusion

The Dynamic Function eXchange Project Flow allows a great deal of flexibility, enabling users to manage their design environment and explore different options. Users must remain careful to track implementation results and bitstreams to ensure that only compatible bitstreams, built from a single fixed static image, are downloaded to the target device.

Lab 4

Vivado Debug and the DFX Project Flow

Overview

This lab covers more project-based features for the Dynamic Function eXchange (DFX) solution. The following topics are covered in this lab:

- The DFX project flow in the AMD Vivado™ IDE
- IP support within Reconfigurable Modules (RM)
- Inserting Vivado Debug cores within Reconfigurable Modules
- Improvements to reporting unique to DFX
- Debugging within the Vivado Hardware Manager

It differs from the DFX flow in Lab 3 in that while this Project Flow does not show greybox implementation and a few other features, it covers IP and debugging. This lab supports the following development platforms:

- KCU116 (AMD Kintex™ UltraScale+™ devices)
- VCU118 (AMD Virtex™ UltraScale+™ devices)
- KCU105 (AMD Kintex™ UltraScale™ devices)
- VCU108 (AMD Virtex™ UltraScale™ devices)

Step 1: Extract the Tutorial Design Files

1. Download the reference design files.
2. Extract the zip file contents to any write-accessible location.
3. In the extracted files, navigate to `\dfx_project_debug`.

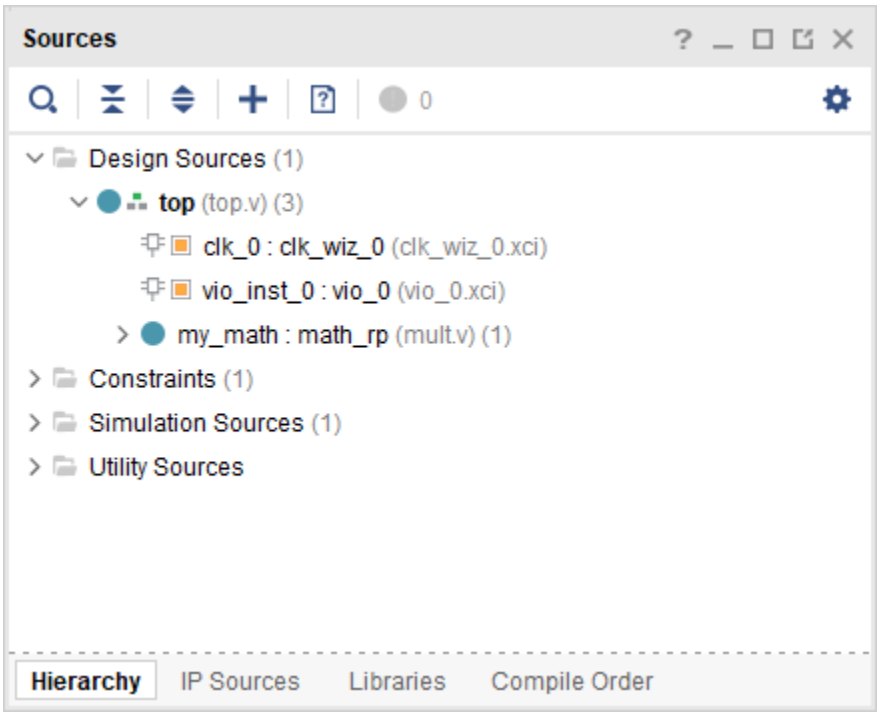
Step 2: Loading Initial Design Sources

The first unique step in any DFX design flow (project based or otherwise) is to define the parts of the design to be marked as reconfigurable. This is done via context menus in the Hierarchical Source View in project mode.

1. Extract the design from the TSC archive. The `dfx_project_debug` data directory is referred to in this tutorial as the `<Extract_Dir>`.
2. Open the AMD Vivado™ IDE and select **New Project**.
3. Select the `<Extract_Dir>` as the Project location. Leave the Project name as `project_1`, and leave the Create project subdirectory option checked. Click **Next**.
4. Select **RTL Project** and ensure the **Do not specify sources at this time** check box is unchecked, then click **Next**.
5. Click the **Add Files** button and select these sources to add to the design:
   - `<Extract_Dir>\Sources\hdl\top.v`
   - `<Extract_Dir>\Sources\hdl\multiplier\mult.v`
   - `<Extract_Dir>\Sources\ip\<board>\clk_wiz\clk_wiz_0.xci`
   - `<Extract_Dir>\Sources\ip\<board>\vio\vio_0.xci`

   Do not select `add.v` or `mult_no_ila.v` (in the `adder` and `multiplier_without_ila` folders, respectively), as these are the sources for RMs are added later.
6. Select the **Copy sources into project** check box.
7. Click **Next** to get to the Add Constraints page, then click the **Add Files** button, and select the following file: `<Extract_Dir>\Sources\xdc\top_io_<board>.xdc`
8. Select the **Copy constraints files into project** checkbox.

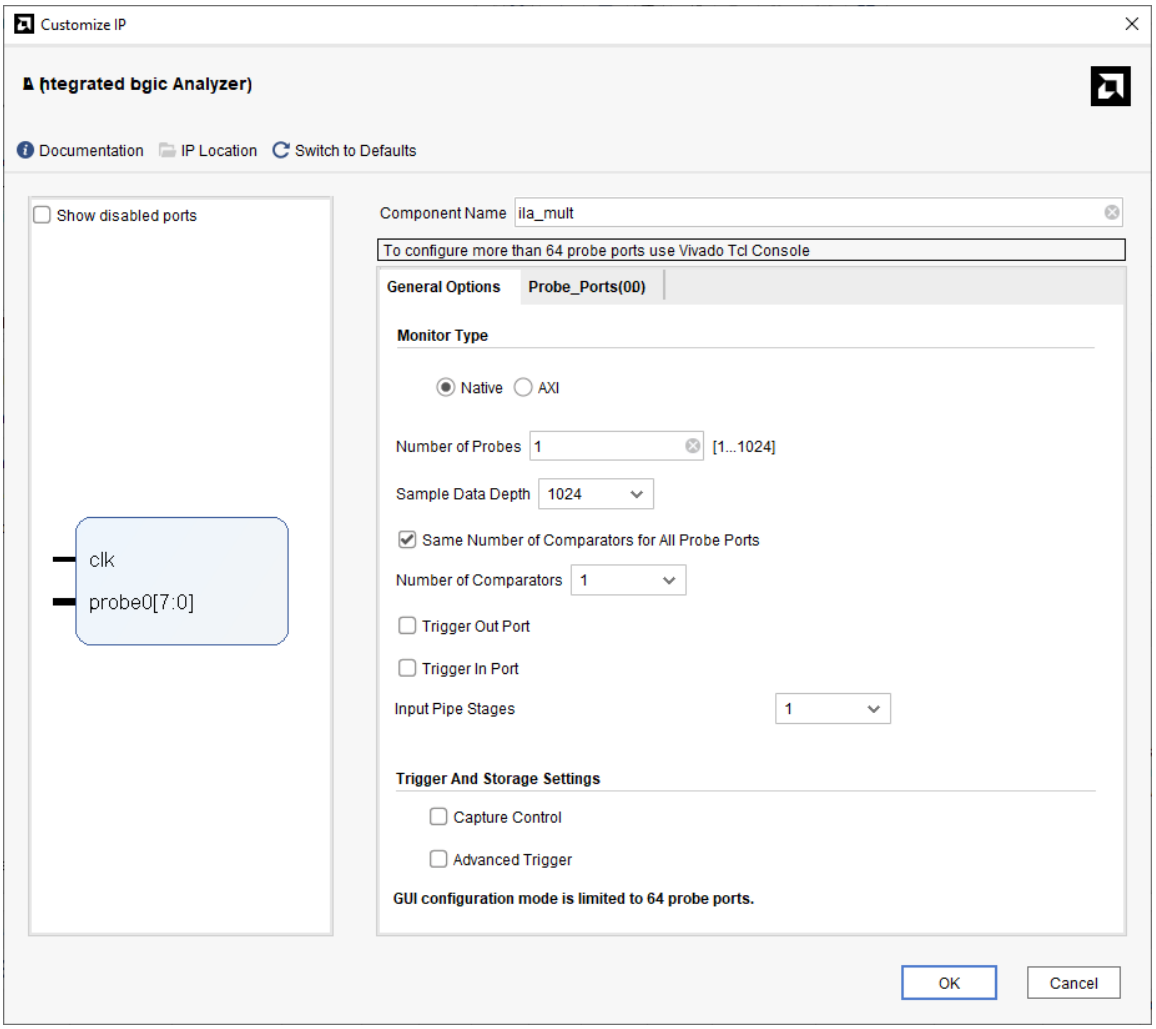
   ***Note*:** These constraint files are full design constraints, scoped to the top-level design. This constraint file does not include a floorplan.
9. Click **Next** to choose the part. In the Default Part page, click on **Boards** and (using filters if needed) choose the appropriate target platform:
   - AMD Kintex™ UltraScale™ KCU105 Evaluation Platform
   - AMD Virtex™ UltraScale™ VCU108 Evaluation Platform
   - AMD Kintex™ UltraScale+™ KCU116 Evaluation Platform
   - AMD Virtex™ UltraScale+™ VCU118 Evaluation Platform
10. Click **Next** and then **Finish** to complete project creation. The Sources window shows a standard hierarchical view of the design.



If red lock icons appear on either IP, as shown above, select **Reports → Report IP Status** to see if they can be upgraded. Ensure any out-of-date IP are checked, then click **Upgrade Selected** to bring them to the most recent version available. Leave Core Container disabled and click **Skip** when asked to generate output products.

At this point, a standard project is open. Nothing specific to Dynamic Function eXchange has been done yet. Next, you add an ILA core.

11. In the Flow Navigator, under Project Manager, open the IP Catalog, and select **Foundational Elements → Debug & Verification**.
12. Right-click **ILA (Integrated Logic Analyzer)**, and select **Customize IP**. Customize the IP with these non-default options on the General Options and Probe_Ports(0..0) tabs:
    - Component Name: **ila_mult**
    - Input Pipe Stages: **1**
    - Probe Width of PROBE0: **8**



13. Click **OK** and then **Skip** to create the IP.

    Do not select Generate. Leave the Synthesis Options set to Out of context per IP.

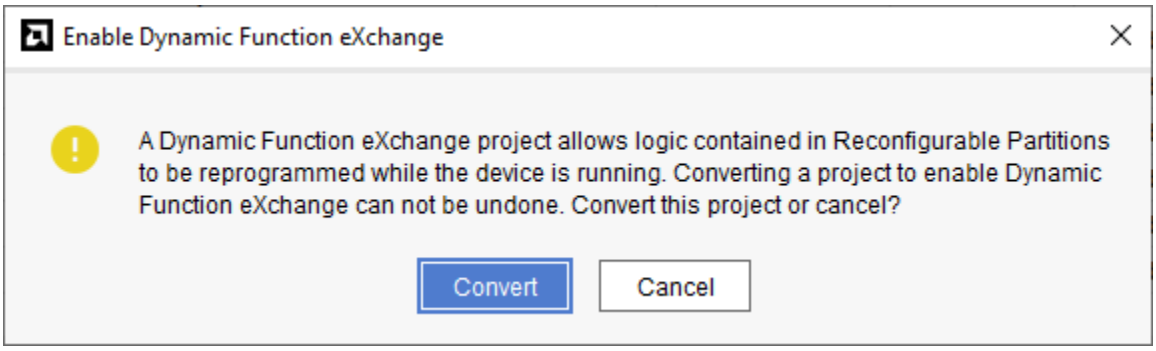
    This IP now fills in underneath the `my_math` hierarchy. The ILA core monitors the multiply function. You have now completed a full design hierarchy.

Step 3: Setting Up the Design for DFX

1. Select **Tools → Enable Dynamic Function eXchange**.

   This prepares the project for the DFX design flow. Once this is set it cannot be undone, so AMD recommends archiving your project before selecting this option.

   In the dialog box, click **Convert** to turn this project into a DFX project.



2. Right-click **my_math** in the sources window and select the **Create Partition Definition...** option.

   This defines this instance as a Reconfigurable Partition in the design. Out-of-context synthesis is run to keep this module separated from top, and the post-synthesis checkpoint is used for the math_rp instance.

   TIP: *If there are multiple instantiations of a module within a design, each are marked as reconfigurable. If they all do not need to be reconfigurable, then the modules must be manually modified to remain unique. Then you can independently tag desired instances as Reconfigurable Partitions.*

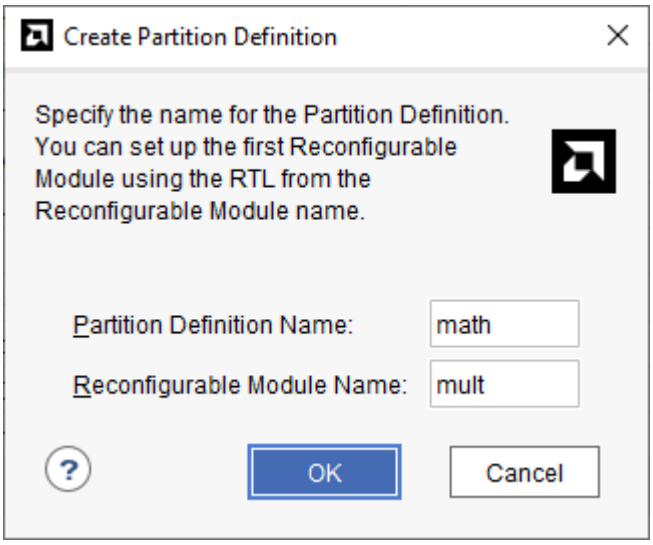
   ***Note*:** IP placed within Reconfigurable Modules can be synthesized as Global or Out-of-Context. For this lab, leave the ILA IP set as the default of Out-of-Context.

3. In the dialog box that appears, name both the Partition Definition and the Reconfigurable Module.

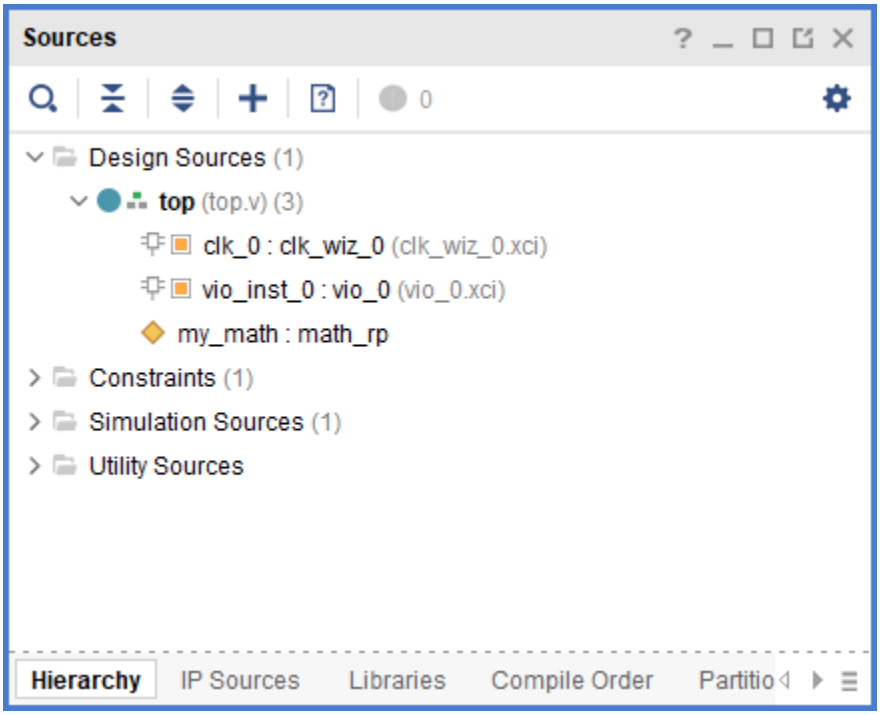
   The Partition Definition is the general reference for the workspace into which all Reconfigurable Modules will be inserted, so give it an appropriate name: `math`.

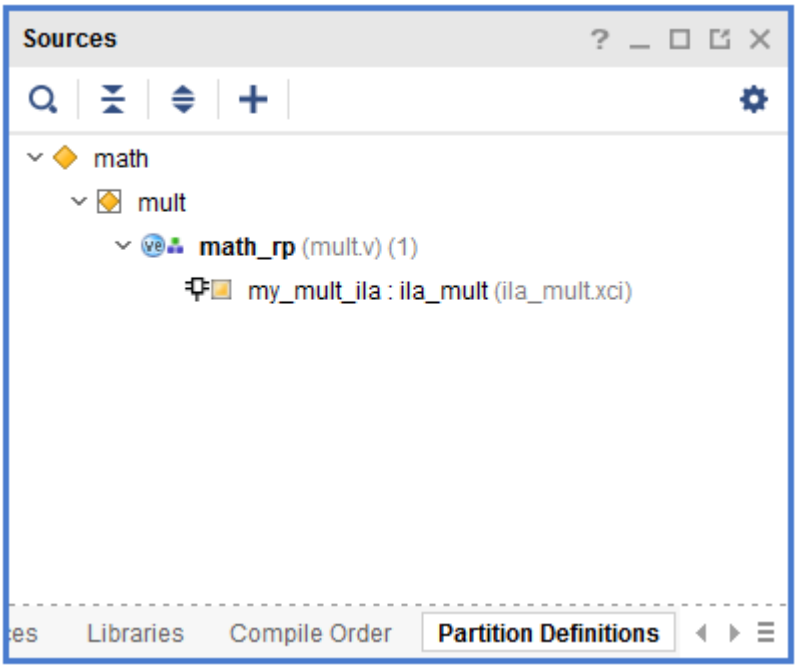
   The Reconfigurable Module refers to this specific RTL instance, so give it a name that references its functionality: `mult`.

4. Then click **OK**.



The Sources view has now changed slightly, with the my_math instance now shown with a yellow diamond, indicating it is a Partition. The Partition Definitions tab in this window shows the list and contents of all Partition Definitions (just one in this case) in the design. In addition, an out-of-context module run has been created for synthesizing the mult module.





At this point, new Reconfigurable Modules can be added (or modified) via the Dynamic Function eXchange Wizard.

IMPORTANT! *After Partitions are defined, all additional RMs must be added via the DFX Wizard, and any management of RM sources, configurations, and runs must also be done via this wizard.*

Step 4: Using the DFX Wizard to Complete the Rest of the Design

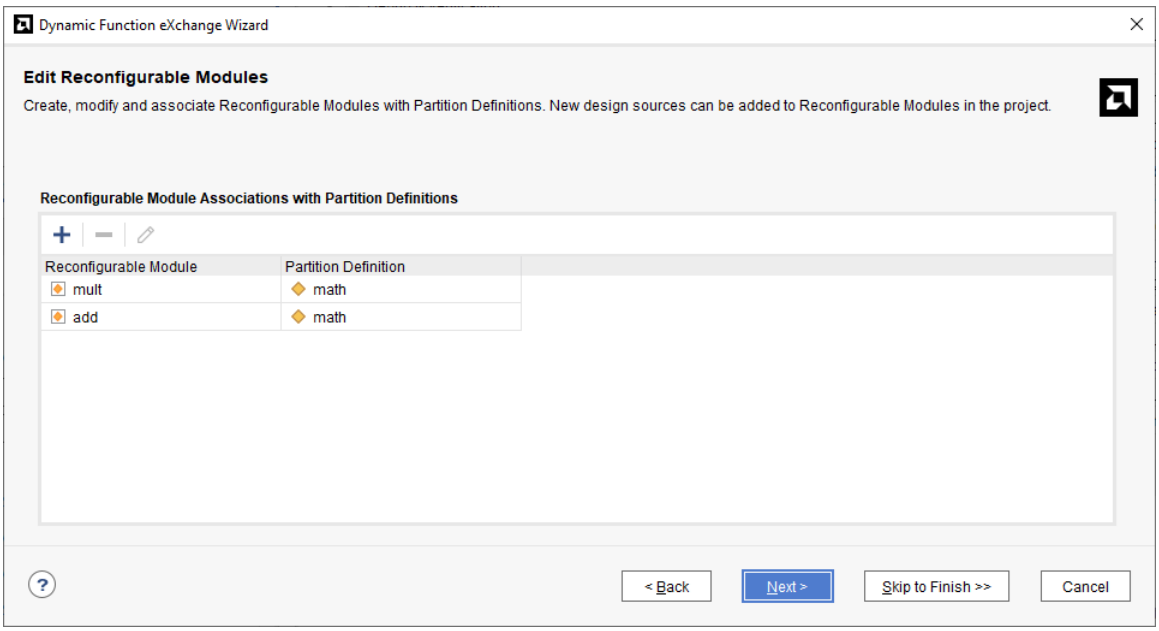
1. Launch the Dynamic Function eXchange Wizard by selecting this option under the Tools menu or from the Flow Navigator.
2. Click **Next** to get to the Edit Reconfigurable Modules page. Here you can see the mult RM already exists, and there are add, remove and edit buttons on the left-hand side of the page. Click on the + icon to add a new RM.
3. Click the **Add Files** button to select the top level of the add function:

   ```
   <Extract_Dir>\Sources\hdl\adder\add.v
   ```

   If module-level constraints were needed, they would be added here. The constraints would need to be scoped to the level of hierarchy for this Partition.
4. Select **add.v** either by double-clicking or by single-clicking and selecting **OK**.

5. Fill in the Reconfigurable Module name to `add`. Set the Partition Definition name to be `math`, leave Top Module name empty and the Sources are already synthesized option unchecked. Select the **Copy sources into project** checkbox. Click **OK** to create the new module.

   Two Reconfigurable Modules are now available for the math Reconfigurable Partition.

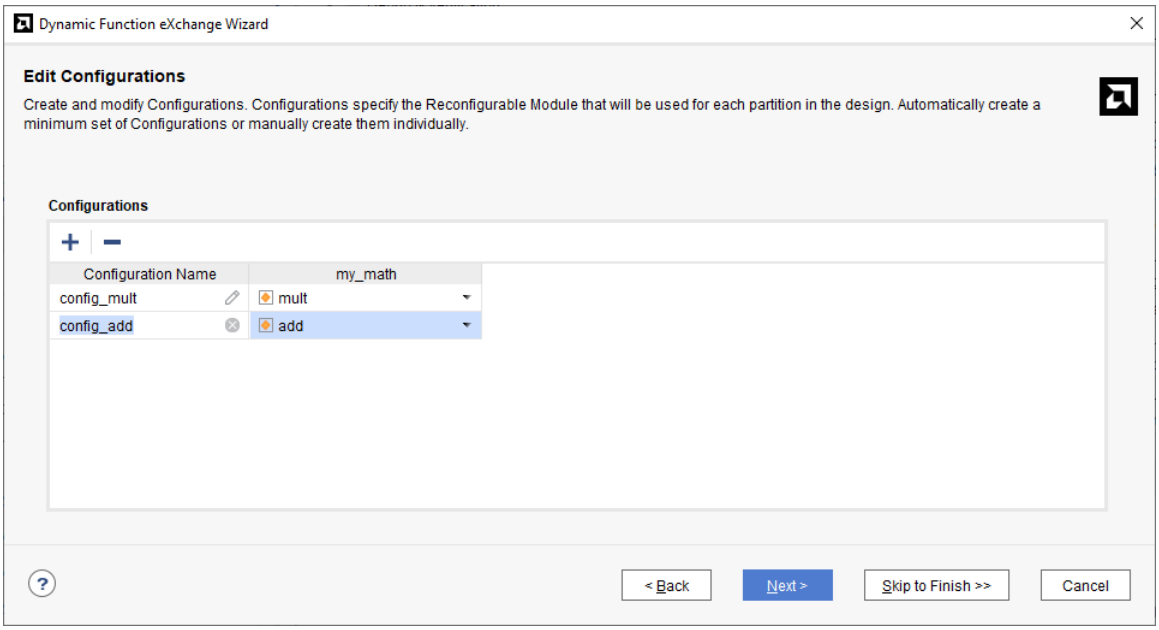


6. Click **Next** to define configurations.
7. Select the **automatically create configurations** link to let the Wizard create the configurations.

   Configurations are full design images consisting of the static design and one RM per RP. You can either create any desired set of configurations, or simply let the wizard select them for you, as you did above.

   The minimum set of two configurations is now created. The math instance has been given mult in the first configuration and add in the second configuration.

   ***Note*:** The Configuration Name is editable, and the names have been updated to config_mult and config_add to reflect the Reconfigurable Modules contained within each configuration.



Additional configurations can be created by using these two Reconfigurable Modules when desired. Greybox (blackbox with LUT tie-offs) configurations can also be selected, but this feature is not used in this lab.

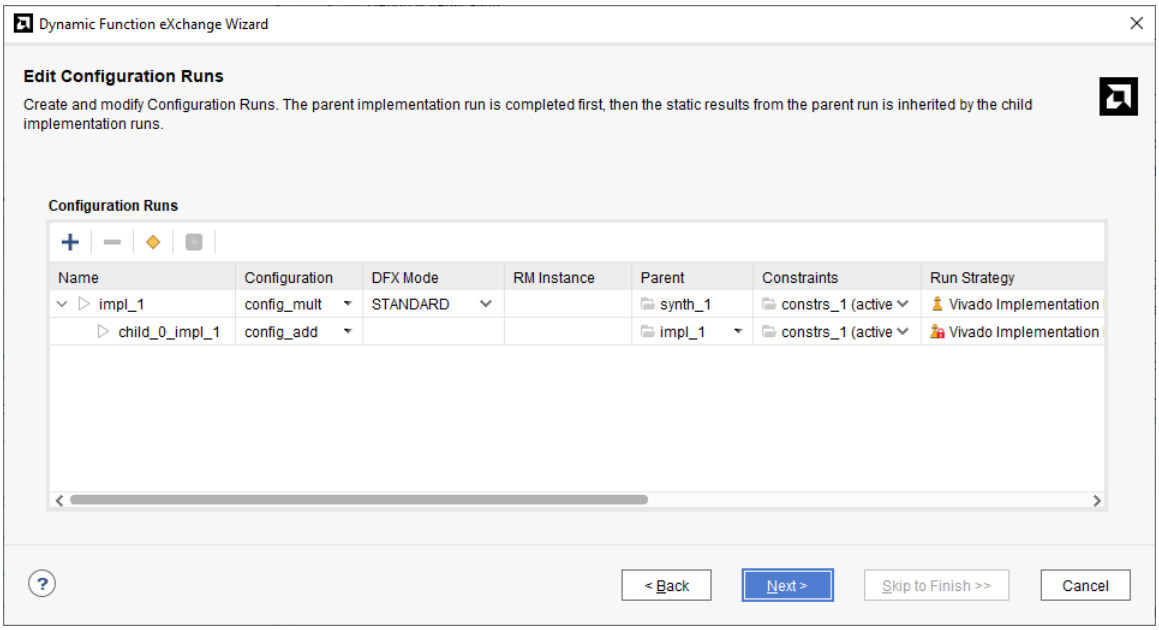
8. Click **Next** to get to the Edit Configuration Runs page.

   As with configurations themselves, the runs used to implement each configuration can be automatically or manually created. A parent-child relationship will define how the runs interact – the parent run implements the static design and all RMs within that configuration, then child runs reuse the locked static design while implementing the RMs within that configuration in that established context.

9. Click the **Standard DFX** link (UltraScale+ device targets) or the **automatically create configuration runs** link (UltraScale or 7 series device targets) to populate the Configuration Runs page with the minimum set of runs.

   ***Note*:** The following figures show a design that targets an UltraScale+ device.

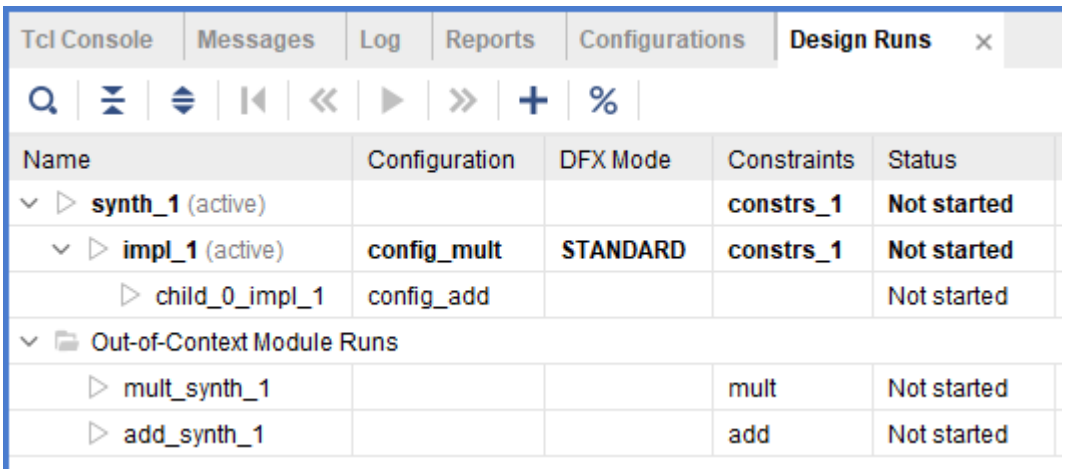
   This creates two runs, consisting of one parent configuration (config_mult) and one child configuration (config_add). Any number of independent or related runs can be created within this wizard with options for using different strategies or constraint sets for any of them. For now, leave this set to the two runs set here. Note that the names of the runs are not editable.



10. Click **Next** to see the Summary page, and click **Finish** to complete the setup and exit the Wizard.

> **IMPORTANT!** *Nothing is created or modified until you click Finish to exit the DFX Wizard. All actions are queued until this last click, so it is possible to step forward and back as needed without implementing changes until you are ready.*

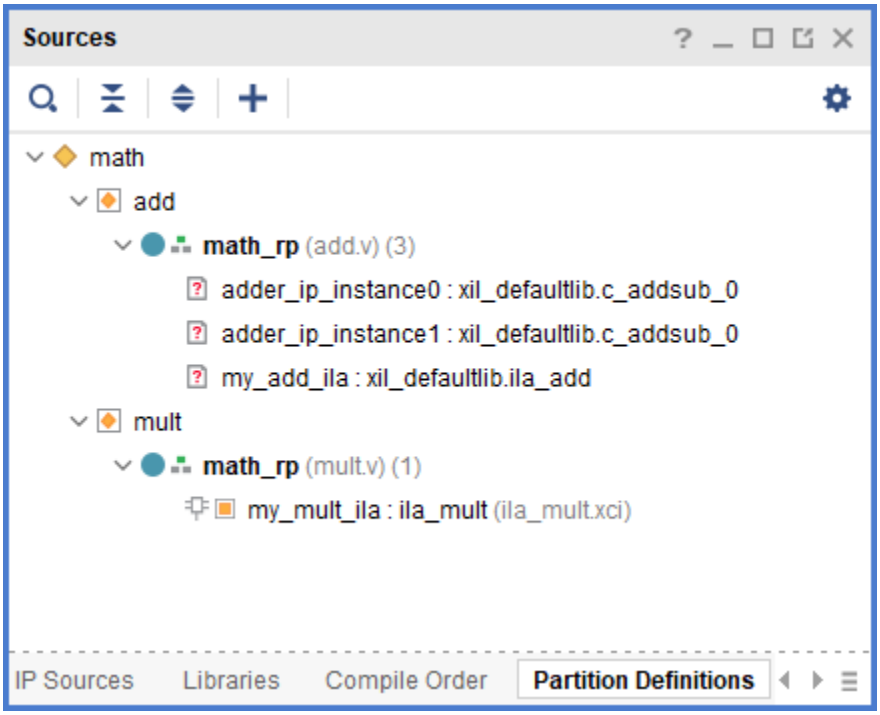
Back in the Vivado IDE, the Design Runs window has been updated. A second out-of-context synthesis run has been added for the math RM, and a child implementation run (child_0_impl_1) has been created under the parent (impl_1).



Step 5: Adding IP in the Reconfigurable Module

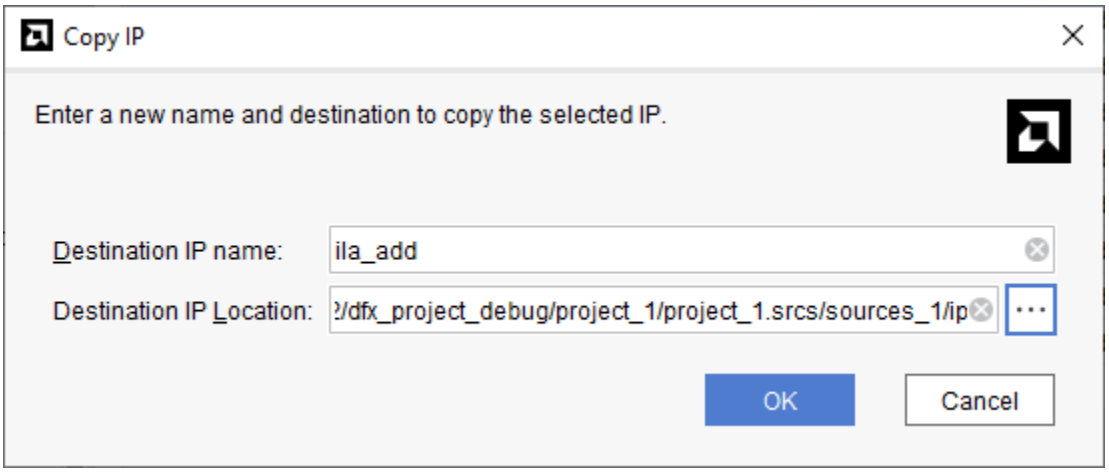
Looking back at the Partition Definitions tab, expand the added RM math_rp to see that there are three submodules that must be added to complete its functionality, as indicated by the question mark icons.

Figure 1: **Partition Definitions Tab with Missing Sources for add**



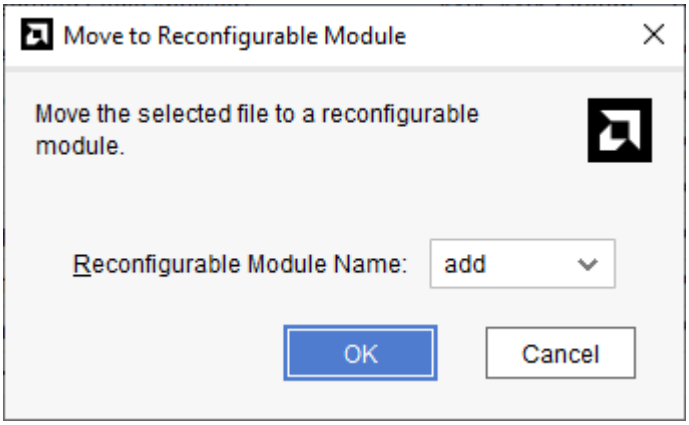
All three of these missing modules are IP. The IP instances must be unique within each RM, so the same ILA core instance cannot be used from static or another RM.

1. Right-click the **ila_mult** instance within the mult RM and select **Copy IP**.
2. Set the Destination IP Name to ila_add and leave the Destination IP Location as is, then click OK.



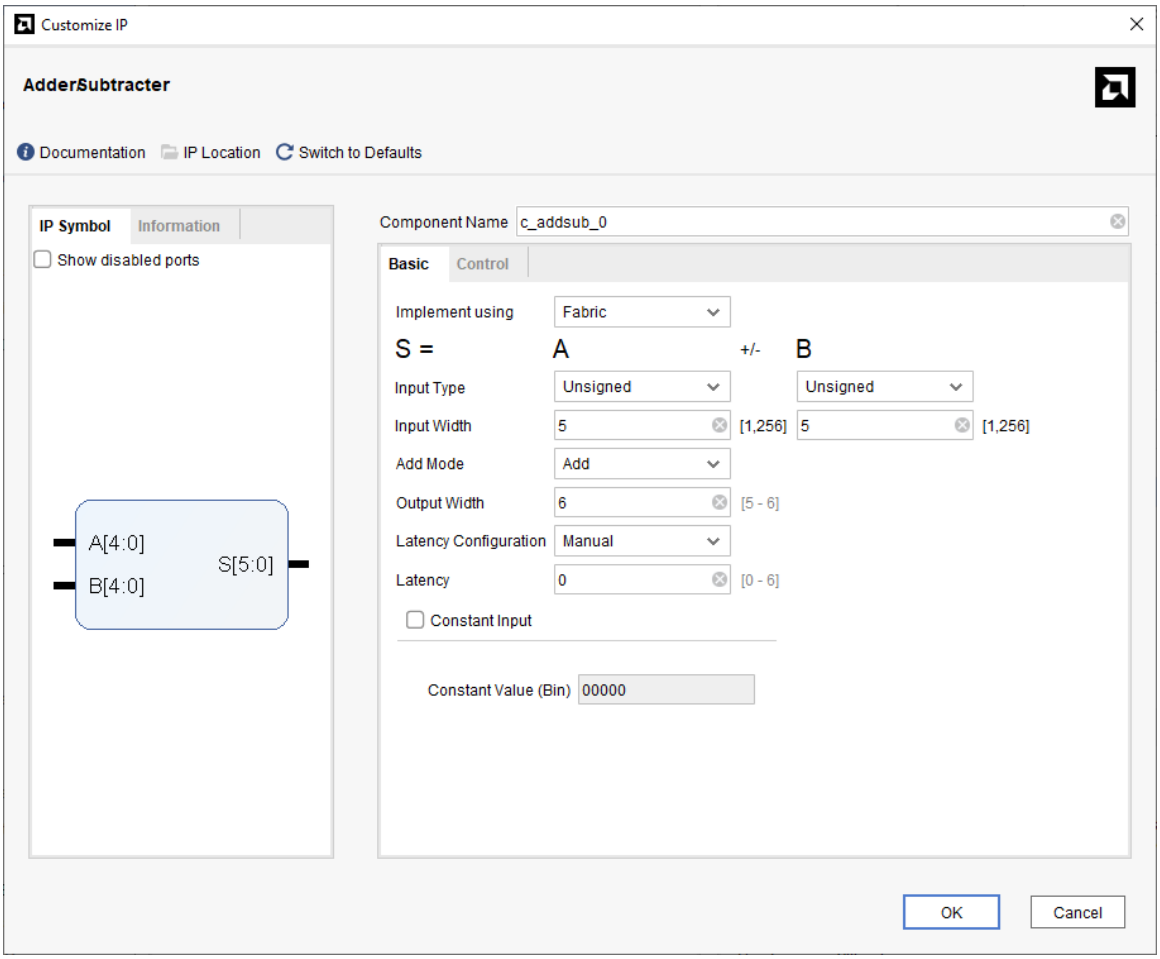
This copied IP will be placed in the main Sources hierarchy window in the primary design fileset, so it must be moved to the add RM blockset.

3. In the Hierarchy tab, right-click on the ila_add instance and select **Move to Reconfigurable Module**. Select the **add** RM and click **OK**.



If you navigate back to the partitions definitions tab, you'll see the ILA IP instance was properly moved under the add RM.

4. Open the IP catalog and search on `add` to find the Adder/Subtracter IP. Open this IP and customize it with these non-default options, leaving the name set to c_addsub_0:
   - Input Type: Unsigned (for both A and B)
   - Input Width: 5 (for both A and B)
   - Output Width: 6
   - Latency: 0
   - Uncheck the Clock Enable on the Control tab
5. Click **OK**.



6. Click **Skip** to complete IP generation.

   Like with the ILA IP, this has been added to the main source set, so follow the same procedure to move it to the add RM.

7. In the Hierarchy tab of the Sources window, right-click on the **c_addsub_0** instance and select **Move to Reconfigurable Module**.

   Select **add** RM and click **OK**.

Note: This IP is used for both adder function instances within the add RM. At this point, the entire design has been loaded, and you are ready to move on to implementation.

Step 6: Synthesizing the Design and Creating a Floorplan

Before launching synthesis, take a look at the naming convention that inserts the Debug Hubs necessary for the Vivado Debug solution.

1. Open `mult.v` and examine the port list in this file.

   The port list includes twelve ports that start with S_BSCAN_. These ports are used to connect the Debug Hubs that are inserted in the static and reconfigurable parts of the design. The insertion of these hubs is automatic. Connections are automatically made as long as the port list matches this naming convention.

   > **CAUTION!** *These exact port names must be used to have the inference occur. If the port names differ at all, then the attributes shown in the comments of these RTL files must be used to assign the new signal names to the debug properties as indicated.*

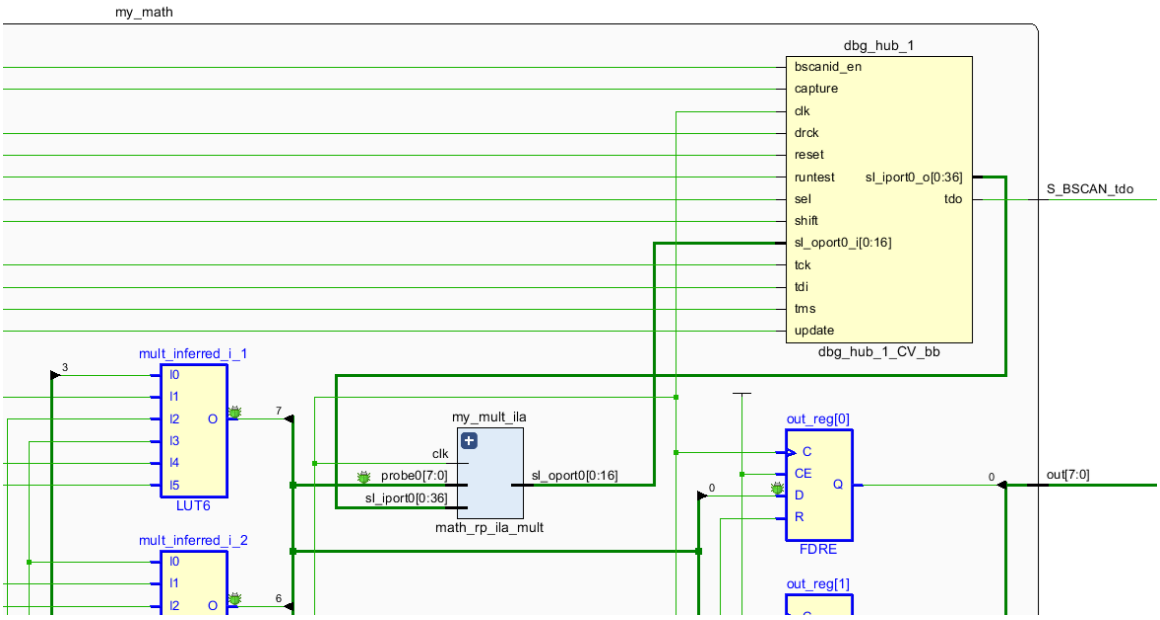
   With the design from section one open in the Vivado IDE, take a look at the Design Runs window. The top-level design synthesis run (synth_1) and the parent implementation run (impl_1) are marked "active." The Flow Navigator actions apply to these active runs, so clicking on Run Synthesis or Run Implementation will pull the design through only these runs, as well as the OOC synthesis runs needed to complete them. You could select the child implementation run, right-click, and select **Launch Runs** to pull through the entire flow, but we'll run synthesis separately here.

2. In the Flow Navigator, click **Run Synthesis**. When synthesis completes, select **Open Synthesized Design**.

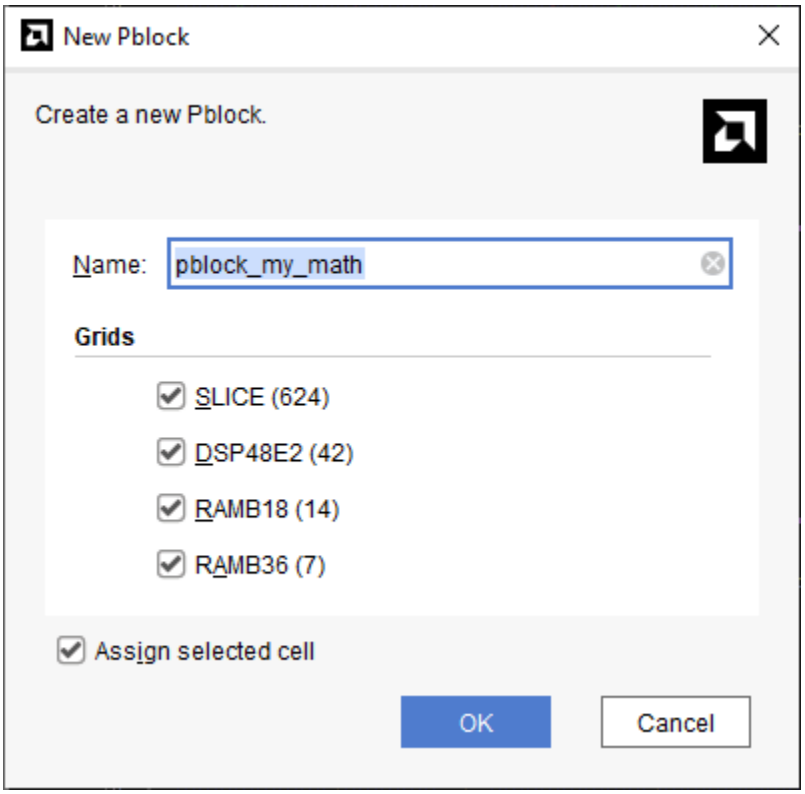
   This action synthesizes all OOC modules, followed by synthesis of the top level design. This is no different than any design with OOC modules (IP or otherwise).

3. Open the schematic for the post-synthesis view to see the insertions performed during synthesis.

   In the top level design, see that a dbg_hub instance was inserted. Its sl_* ports are connected to the VIO debug core at the top level. Next, descend into the my_math hierarchy to see that another dbg_hub instance has been inserted, with its sl_* ports connected to the ILA debug core in that module. Note that this Reconfigurable Module is the multiplier, as this is the schematic view of the active parent run.

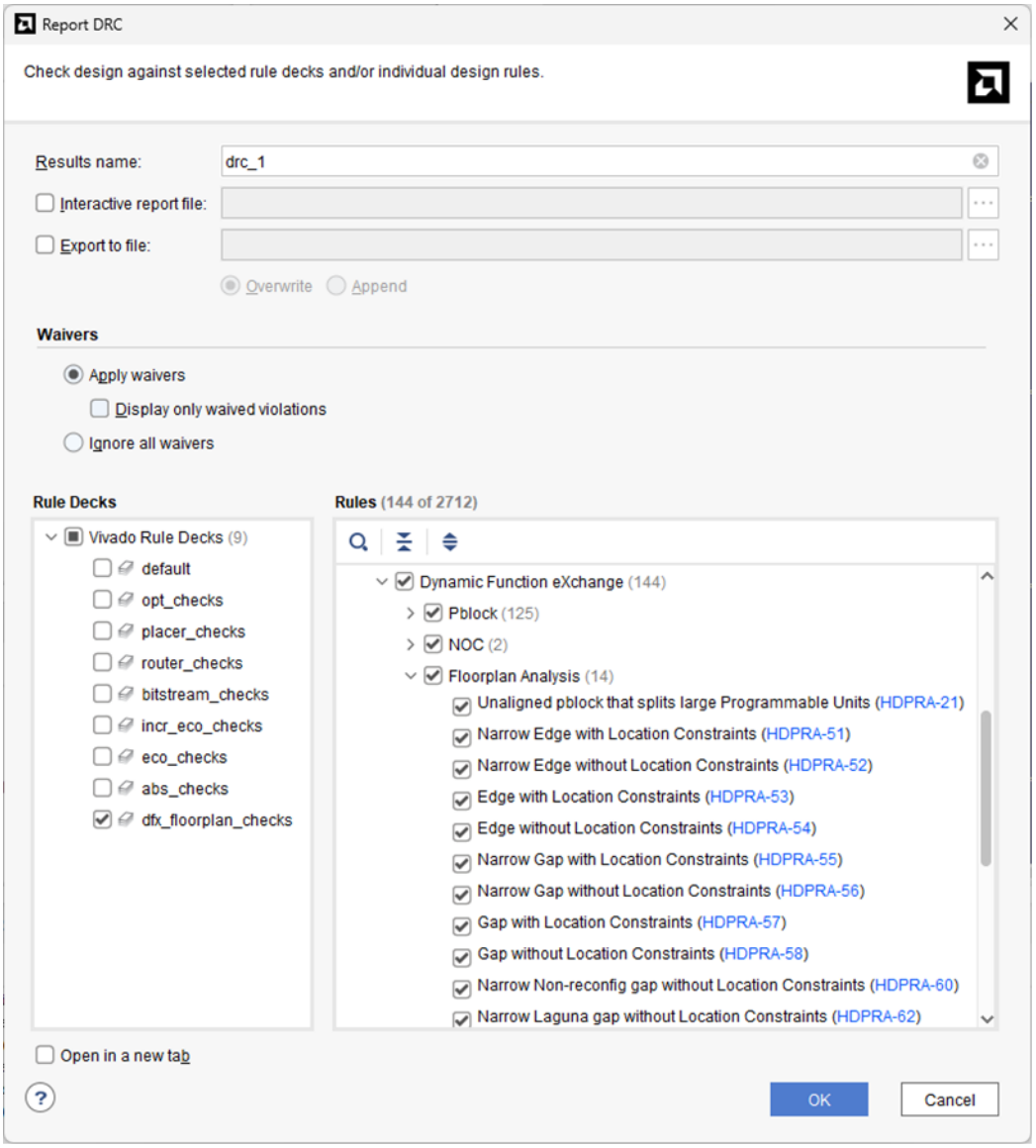


4. Select **Layout → Floorplanning** to put Vivado in floorplanning mode. Then in the Netlist window, right-click on the my_math instance and select **Floorplanning → Draw Pblock**. Create a Pblock in any location. In the dialog box that appears, keep the name pblock_my_math and leave only SLICE, DSP and BRAM resource types checked.



If the region you have selected does not have enough resources of any particular type, these resource types appear in red in the Statistics tab of the Pblock Properties window. Make adjustments as necessary, then save the floorplan. Each RM contains an ILA core, which requires block RAM. Also, this design has a high number of control sets, so the region required might be a little bigger than expected. An area of at least 3000 CLBs plus a column of block RAM is suggested.

5. Run DFX-specific design rule checks by selecting **Reports → Report DRC**. To save time, you can deselect all checkboxes other than the one for Dynamic Function eXchange. Click **OK** to run.



Fix any errors that might appear. Advisory messages might appear for certain devices with suggestions on how to improve the quality of the given Pblocks.

TIP: *Run DFX Design Rule Checks early and often.*

6. Save your constraints by clicking **Save Constraints** in the top toolbar.

Step 7: Running the PR Configuration Analysis Report

The PR Configuration Analysis tool compares each Reconfigurable Module that you select to give you input on your DFX design. It examines resource usage, floorplanning, clocking, and timing metrics to help you manage the overall DFX design. The PR Configuration Analysis tool is run through the Tcl Console.

1. In the Tcl Console, cd into the project directory. Next, enter this command to run a report on the two RMs available in this design:

```
report_pr_configuration_analysis -cells my_math -dcps
 {./project_1.runs/add_synth_1/math_rp.dcp ./project_1.runs/mult_synth_1/
math_rp.dcp}
```

Note: If your project is not named "project_1" you'll need to adjust this in the Tcl command.

This runs the analysis with the default settings, gathering data for the first three focus areas listed below. Use the `-help` option to see that you can focus on three specific areas.

- The `-complexity` switch focuses the report on resource usage, including the maximum resources required for the RP.
- The `-clocking` switch focuses the report on clock usage and loads for each RM.
- The `-timing` switch focuses the report on boundary interface timing details.
- The `-rent` switch adds rent metrics to the report, but can take a long time to run.
- The `-file` switch redirects the report to a file.

  By examining the report in the Tcl Console, you see a Complexity summary in section 2. It shows the current RM (the multiplier), RMs 1 and 2 (the adder and multiplier, respectively), and a column for the maximum. This table examines the resource utilization of each module to find the maximum of each so you can construct Pblocks appropriately.

  Note that the resource counts of RM1 and RM2 appear to be low. Above the report in the log there are a few critical warnings:

```
CRITICAL WARNING: [Project 1-486] Could not resolve non-primitive
black box cell 'math_rp_c_addsub_0' instantiated as
 'adder_ip_instance0'
```

  The post-synthesis checkpoints do not include the submodule IP as those were generated out-of-context. In order to see the complete picture of each RM, these lower-level checkpoints must be linked in, or the IP must be synthesized set to Global.

2. In the Partition Definitions tab, expand the hierarchy if necessary to be able to right-click on the **my_mult_ila** IP and select **Generate Output Products**.
3. Change the Synthesis Options value to Global, then click **Apply** and then **Cancel**.
4. Repeat this process for both IP under the adder module, `my_add_ila` and `adder_ip`. The latter has two instances but this process only needs to happen once, as the IP instances are identical.
5. Synthesis for these modules are now out of date. Select **Run Synthesis** in the Flow Navigator. Accept all the dialogs that ask about resetting and resynthesizing all runs.
6. When synthesis completes, rerun the `report_pr_configuration_analysis` command from step 1 and examine the log and results.

Step 8: Implementing the Design

1. In the Flow Navigator, select **Run Implementation** to run place and route on all configurations.

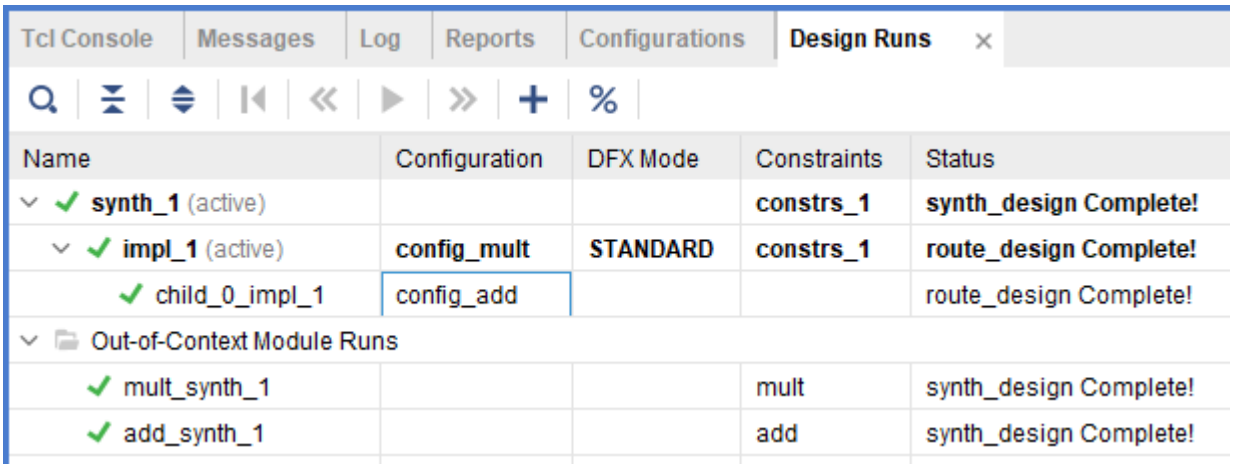
   This action runs implementation first for `impl_1` and then for `child_0_impl_1`. In addition to running place and route for the two runs with all the DFX requirements in place, it does a few more tasks specific to DFX. After `impl_1` completes, Vivado automatically:

   - Writes a module-level (OOC) checkpoint for the routed multiplier RM.
   - Carves out the logic in the RP to create a static-only design image. This is done by calling `update_design -black_box` for the RP instance.
   - Locks all placement and routing for this static-only design. This is done by calling `lock_design -level` routing.
   - Saves this locked static parent image to be reused for all child runs.

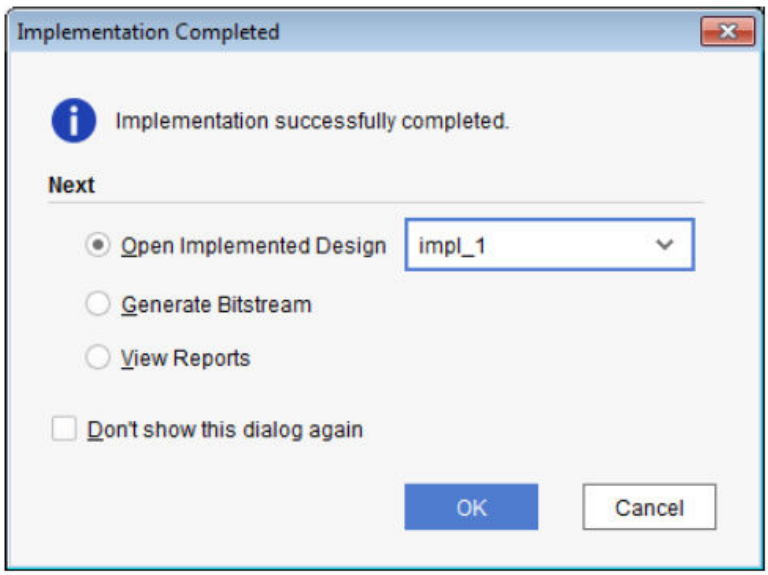
   In addition, when the child run completes, a module-level checkpoint is created for the routed adder RM. A locked static design image would be identical to the parent, so this step is not necessary.

   In Vivado projects, dependency management is handled by the Vivado IDE. If sources are modified, any applicable runs are marked out-of-date. The parent-child relationship means these checks must understand dependencies. For example, if `add.v` is modified, only its OOC synthesis run and the child implementation run would be marked out of date.

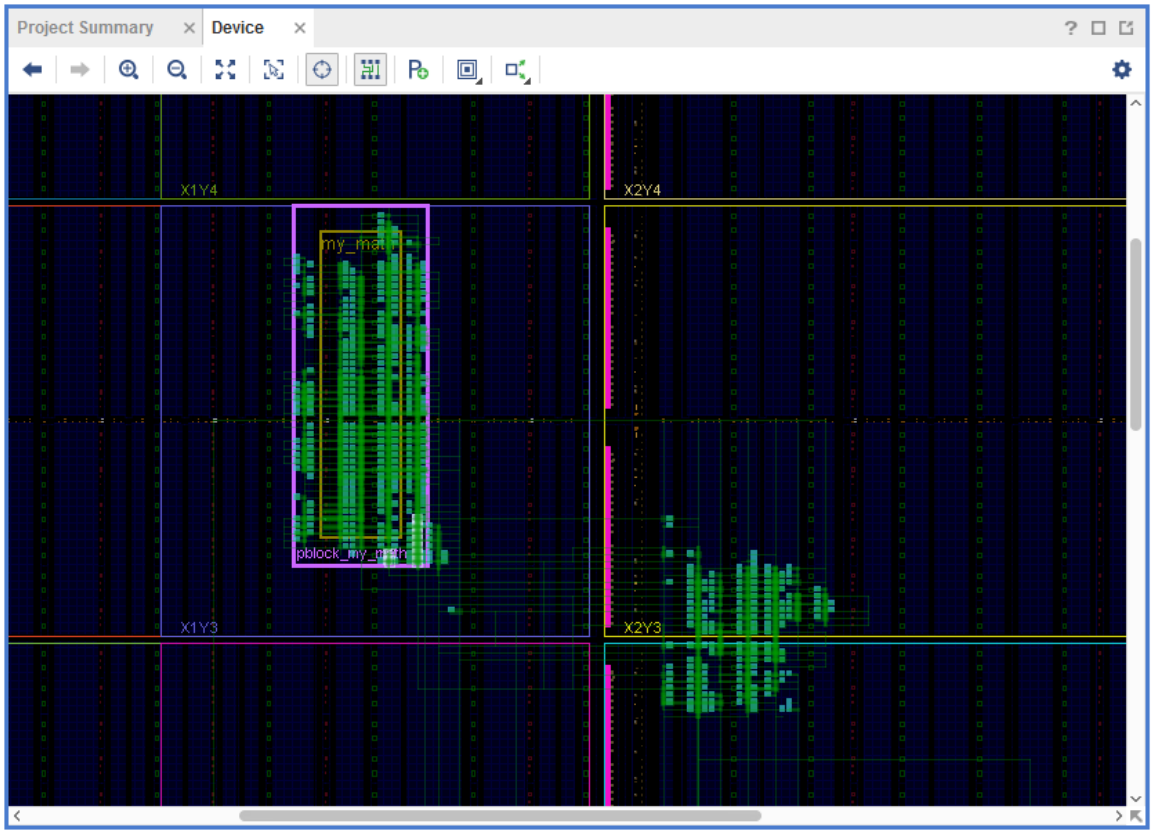
   If only specific configuration runs are desired, these can be individually selected within the Design Runs window. A parent run must be completed successfully before a child run can be launched, as the child run starts by importing the locked static design from the parent.



2. When the implementation runs complete, select **Open Implemented Design**, and click **OK** in the resulting pop-up dialog.



CAUTION! *Even though the design has been processed through to the child implementation run, selecting* ***Open Implemented Design*** *opens the parent run by default. Use the pulldown selection to choose the desired implementation run.*



This is the routed design for the multiplier configuration. Next, take a look at the frameset of the placement and routing areas.

3. In the Tcl Console, cd to the current project directory (if you are not already there). Then run this command to highlight the routing footprint:

```
highlight_objects -color yellow [get_dfx_footprint -route -of_objects
[get_cells my_math]]
```

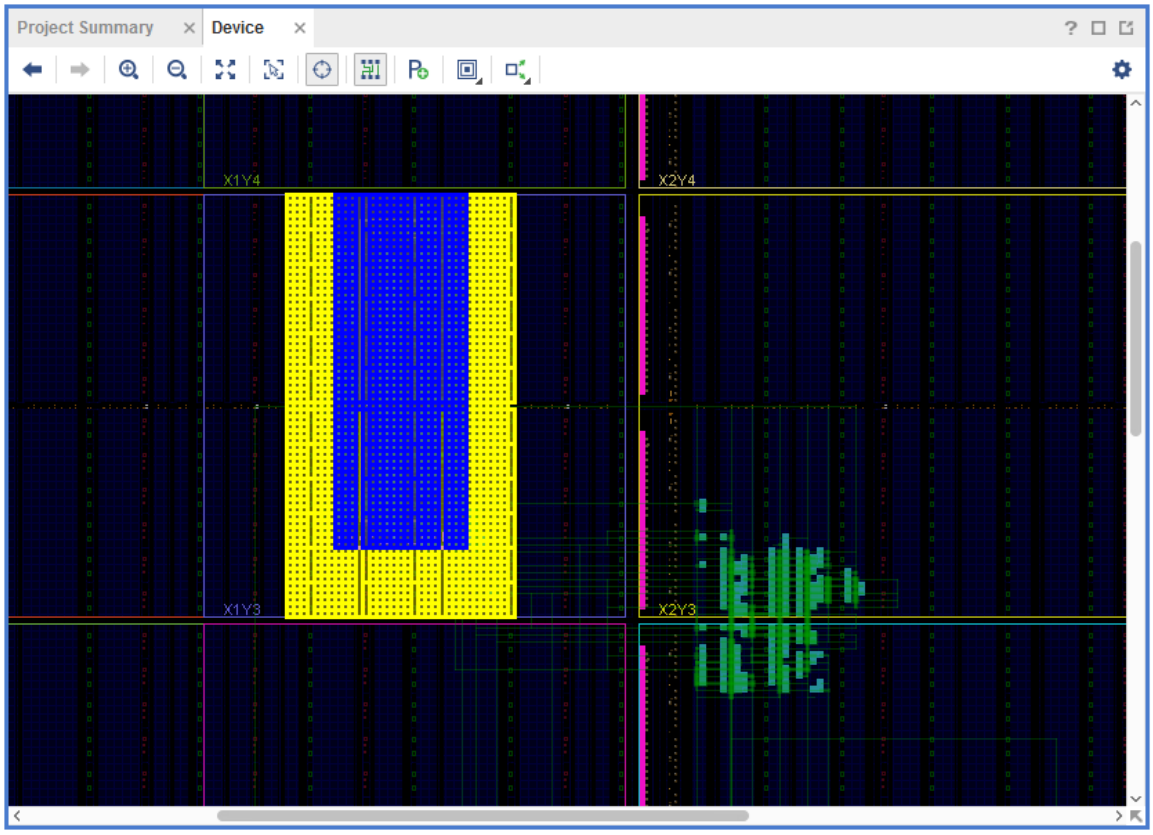
This command identifies the frames that are valid for routing the reconfigurable part of the design. It extends to the height of the clock region(s) occupied by the Pblock, and extends left and right by two programmable units. (Programmable units are pairs of resource columns.)

4. Run this command to identify the frameset used for placing the reconfigurable part of the design.

```
highlight_objects -color blue [get_dfx_footprint -place -of_objects
[get_cells my_math]]
```

This highlighted region is either the Pblock area itself, or an area smaller than the Pblock if the Pblock did not align with programmable units boundaries.

Your device view should look something like the following figure with the math RP highlighted showing placement in blue and routing in yellow boundaries.

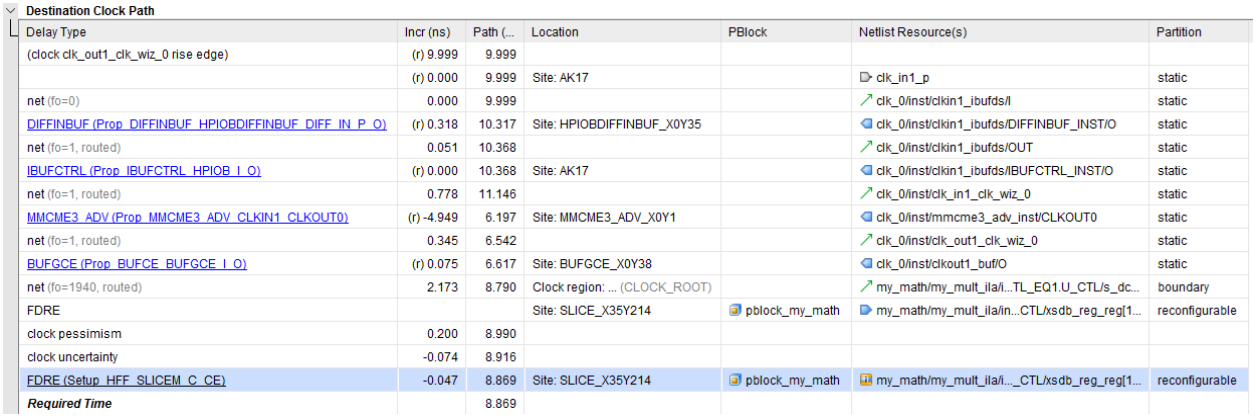


Static logic can be placed in the expanded routing region, which is now the remaining yellow region. Static routing can use any resources in the device.

5. In the Flow Navigator, select **Report Timing Summary** and click **OK** to analyze the design timing.
6. In the Timing tab, select the Design Timing Summary and click on the value for the worst negative slack (WNS) to bring up the top ten worst paths. Double-click on the first path to open the timing summary on that path.

   In this timing report, the Clock Paths and the Data Path, there is a new column labeled Partition that shows which partition (or boundary) that particular part of the path is in.

> **CAUTION!** *You might need to toggle the visibility of the Partition column by right-clicking on the table header and checking Partition. Then expand the timing report window or adjust the column widths to see the Partition column, the last column on the right.*



Destination Clock Path

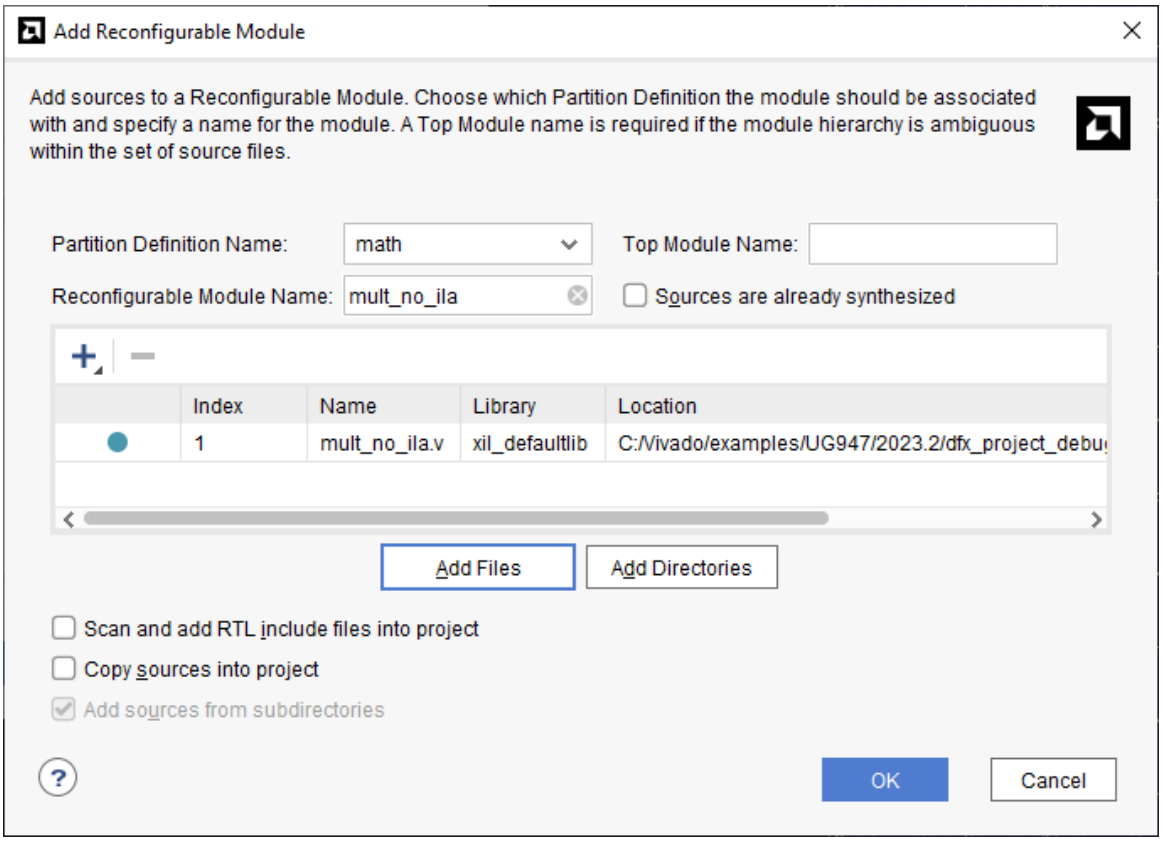
Delay Type	Incr (ns)	Path (...	Location	PBlock	Netlist Resource(s)	Partition
(clock clk_out1_clk_wiz_0 rise edge)	(r) 9.999	9.999				
	(r) 0.000	9.999	Site: AK17		clk_in1_p	static
net (fo=0)	0.000	9.999			clk_0/inst/clkin1_ibufds/I	static
DIFFINBUF (Prop_DIFFINBUF_HPIOBDIFFINBUF_DIFF_IN_P_O)	(r) 0.318	10.317	Site: HPIOBDIFFINBUF_X0Y35		clk_0/inst/clkin1_ibufds/DIFFINBUF_INST/O	static
net (fo=1, routed)	0.051	10.368			clk_0/inst/clkin1_ibufds/OUT	static
IBUFCTRL (Prop_IBUFCTRL_HPIOB_I_O)	(r) 0.000	10.368	Site: AK17		clk_0/inst/clkin1_ibufds/IBUFCTRL_INST/O	static
net (fo=1, routed)	0.778	11.146			clk_0/inst/clk_in1_clk_wiz_0	static
MMCME3_ADV (Prop_MMCME3_ADV_CLKIN1_CLKOUT0)	(r) -4.949	6.197	Site: MMCME3_ADV_X0Y1		clk_0/inst/mmcme3_adv_inst/CLKOUT0	static
net (fo=1, routed)	0.345	6.542			clk_0/inst/clk_out1_clk_wiz_0	static
BUFGCE (Prop_BUFCE_BUFGCE_I_O)	(r) 0.075	6.617	Site: BUFGCE_X0Y38		clk_0/inst/clkout1_buf/O	static
net (fo=1940, routed)	2.173	8.790	Clock region: ... (CLOCK_ROOT)		my_math/my_mult_ila/i...TL_EQ1.U_CTL/s_dc...	boundary
FDRE			Site: SLICE_X35Y214	pblock_my_math	my_math/my_mult_ila/in...CTL/xsdb_reg_reg[1...	reconfigurable
clock pessimism	0.200	8.990				
clock uncertainty	-0.074	8.916				
FDRE (Setup_HFF_SLICEM_C_CE)	-0.047	8.869	Site: SLICE_X35Y214	pblock_my_math	my_math/my_mult_ila/i..._CTL/xsdb_reg_reg[1...	reconfigurable
Required Time		8.869				

7. To close the `impl_1` implemented design, select **File → Close Implemented Design**.

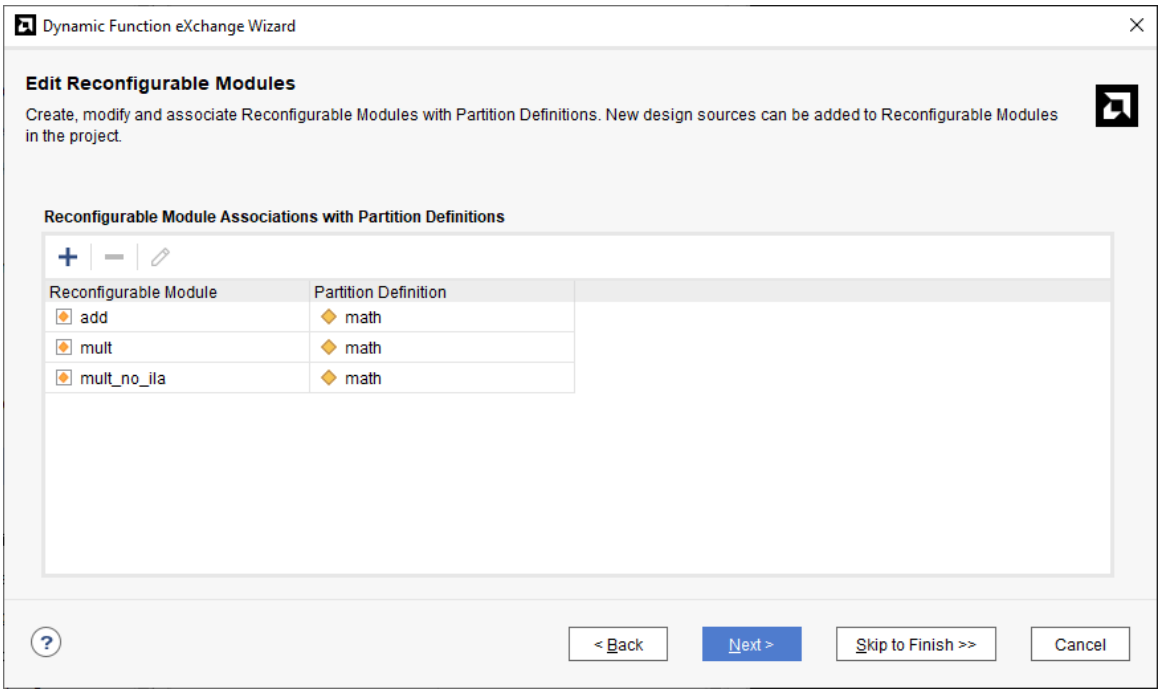
Step 9: Adding an Additional Reconfigurable Module and Corresponding Configuration

In this step, add a third RM and implement its configuration. This new RM is the same multiplier function but with the ILA instantiation commented out. Even though there are no debug cores in this module, the Debug-specific port names (and corresponding attributes if used) are still required for consistency across all RMs. These ports are tied off via LUTs much like the greybox flow.

1. Open the Dynamic Function eXchange Wizard.
2. On the Edit Reconfigurable Modules page, click the **+** button to add a new RM.

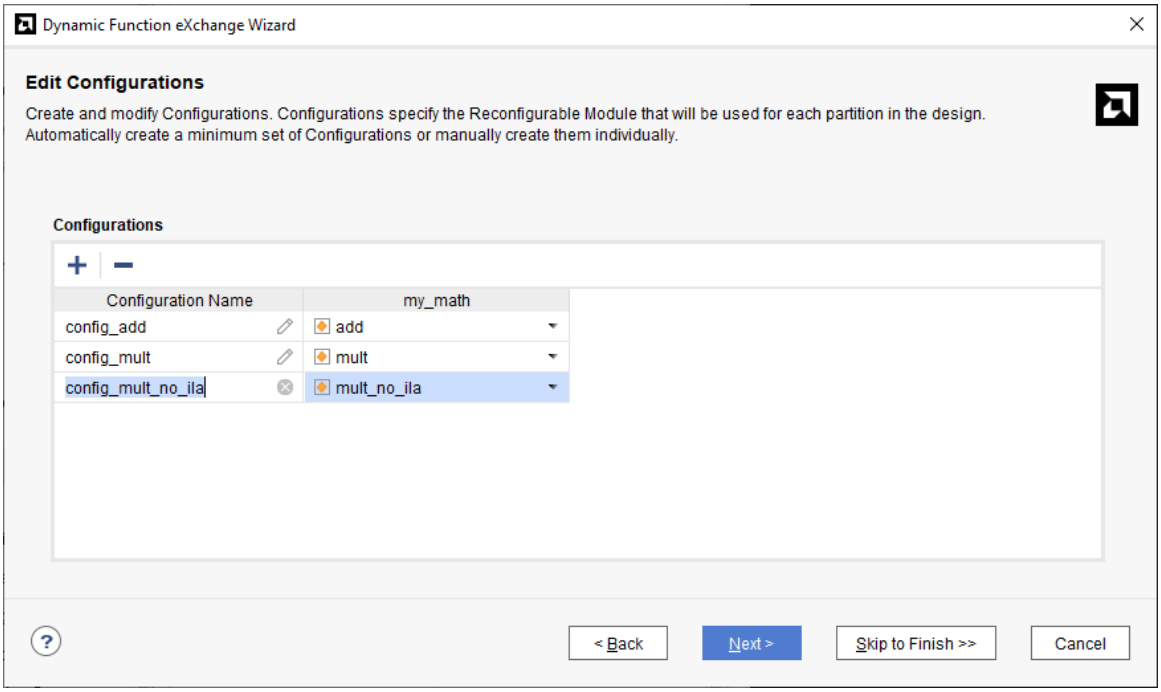


3. Select the `mult_no_ila.v` file in `<Extract_Dir>\Sources\hdl\multiplier_without_ila`. Name the Reconfigurable Module `mult_no_ila`, and then click **OK** and click **Next**.



Note: On the Edit Configurations page, there is no longer an option to automatically create configurations, as you already have two existing ones. You can re-enable this option by removing all existing configurations, but this recreates all configurations and removes all existing results.

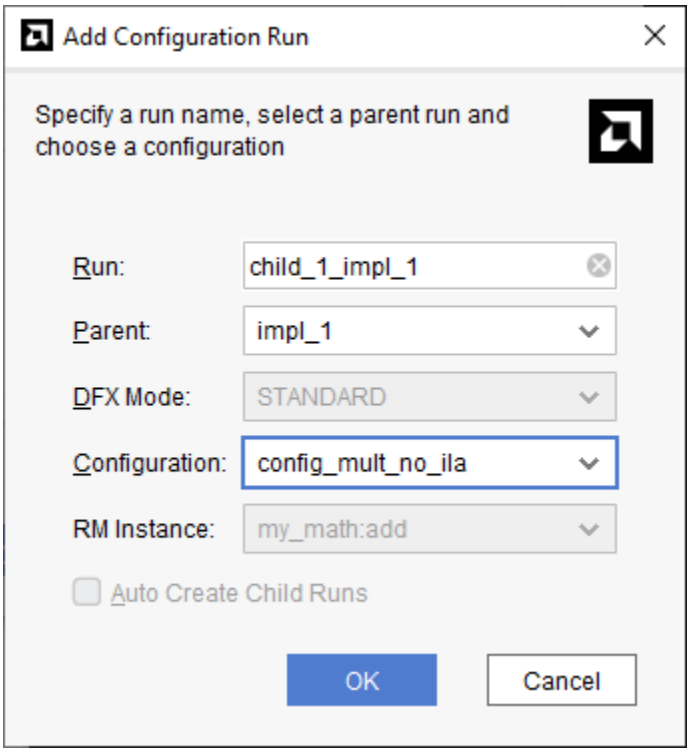
4. Create a new configuration by clicking the + button, entering the name `config_mult_no_ila`, then clicking **OK**. Select `mult_no_ila` as the Reconfigurable Module.



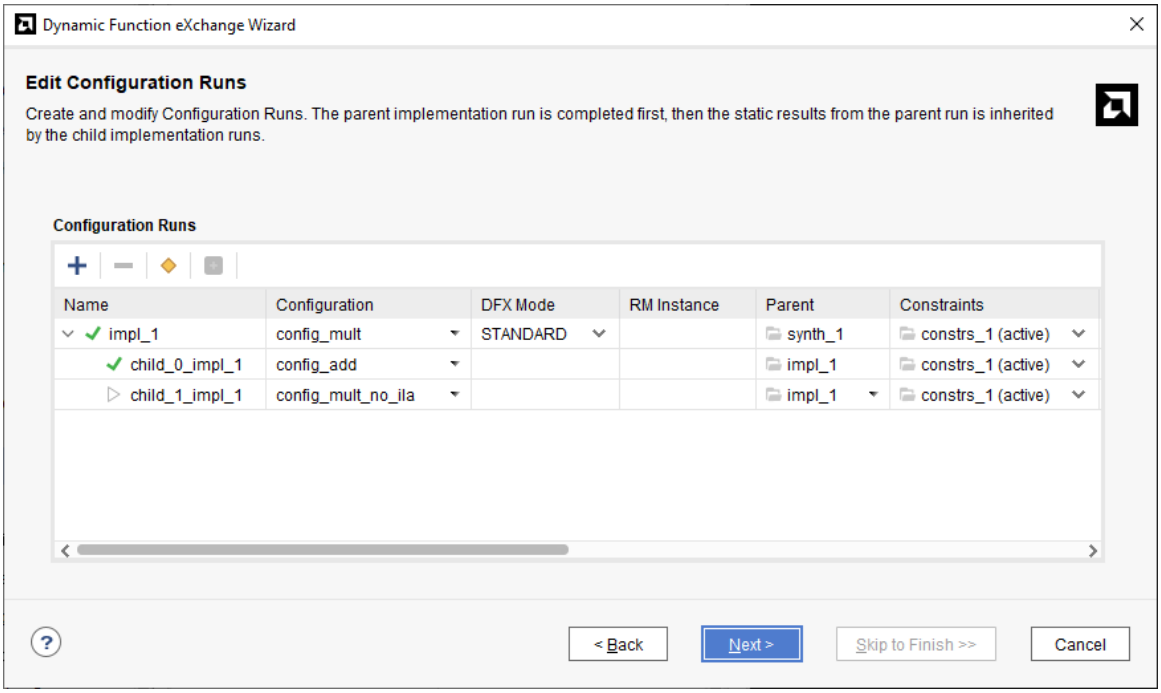
5. Click **Next** to advance to the Configuration Runs. Use the + button to create a new configuration with these properties:

   - **Run:** `child_1_impl_1` matches the existing convention, although it can be named anything.
   - **Parent:** `impl_1` makes this configuration a child run of the existing parent run.
   - **Configuration:** `config_mult_no_ila` is the one with the new RM.

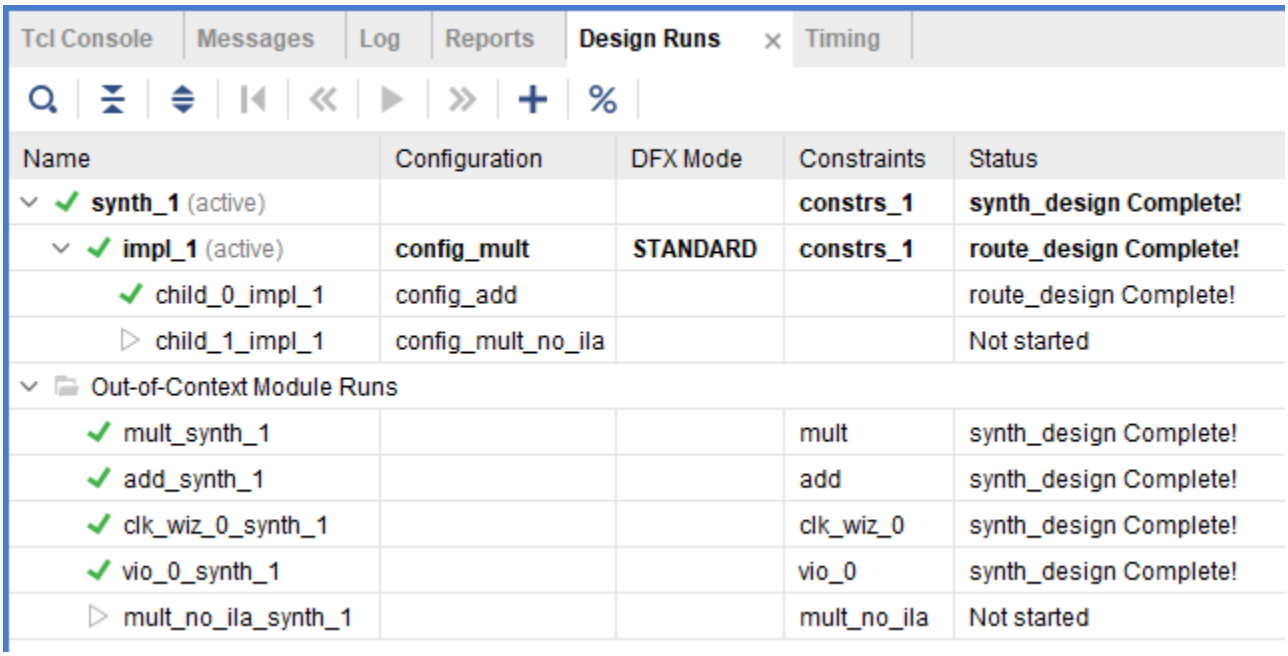
   Click **OK** to accept this new configuration.



This new configuration, as a child of the existing `impl_1`, reuses the static design implementation results, just like `config_add` did. Three runs now exist, with two as children of the initial parent. The green check marks indicate that two of the runs are currently complete.



6. Click **Next** then Finish to build this new configuration run.



Name	Configuration	DFX Mode	Constraints	Status
synth_1 (active)			**constrs_1**	**synth_design Complete!**
impl_1 (active)	**config_mult**	**STANDARD**	**constrs_1**	**route_design Complete!**
child_0_impl_1	config_add			route_design Complete!
child_1_impl_1	config_mult_no_ila			Not started
Out-of-Context Module Runs				
mult_synth_1			mult	synth_design Complete!
add_synth_1			add	synth_design Complete!
clk_wiz_0_synth_1			clk_wiz_0	synth_design Complete!
vio_0_synth_1			vio_0	synth_design Complete!
mult_no_ila_synth_1			mult_no_ila	Not started

7. Select this new child implementation run, right-click, and select **Launch Runs**. This runs OOC synthesis on the `mult_no_ila` module, then implements this module within the context of the locked static design.

> **CAUTION!** *Do not select Run Implementation from the Flow Navigator. It will rerun all the implementation runs, even those that have completed.*

8. Click **Cancel** on the dialog that opens after implementation completes.

   Right-click on `child_1_impl_1` and select **Open Run**. In the device view note two things:

   a. The static logic is locked and therefore appears orange.

   b. In the Design Runs tab, notice that the amount of logic in the RP Pblock is much smaller than for the other configurations.

9. Select **Tools → Schematic** (or selelct F4) to open the schematic view. Descend into the `math_rp` instance to see that all the BSCAN ports are tied to LUTs and no ILA or `Dbg_Hub` cores are inserted.

Step 10: Generating Bitstreams

At this point, there are two steps remaining: the first is running PR Verify to compare the two configurations to ensure consistency of the static part of the design images. This step is highly recommended and will occur automatically within the Vivado project. The second step is to generate the bitstreams themselves.

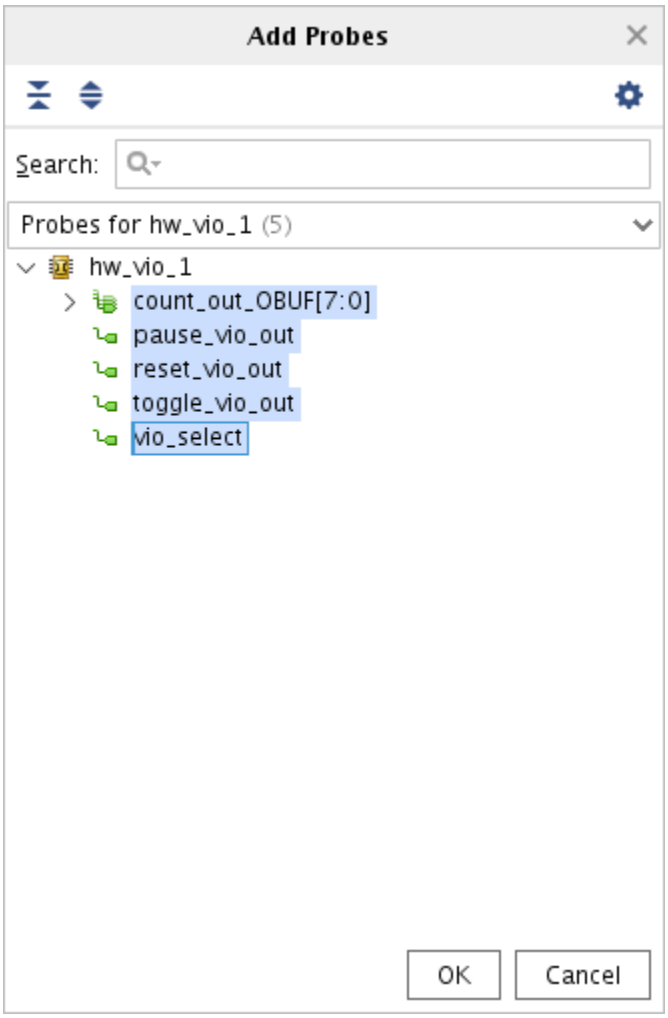
1. In the Flow Navigator, click **Generate Bitstream**. This action launches bitstream generation on the active parent run, and launches PR Verify and then bitstream generation on the implemented child runs.
2. This generates full and partial bitstreams for each configuration run.
3. When bitstream generation completes, select **Open Hardware Manager**.

Step 11: Connecting to the Board and Programming the FPGA

1. Open the hardware manager and connect to the target board.

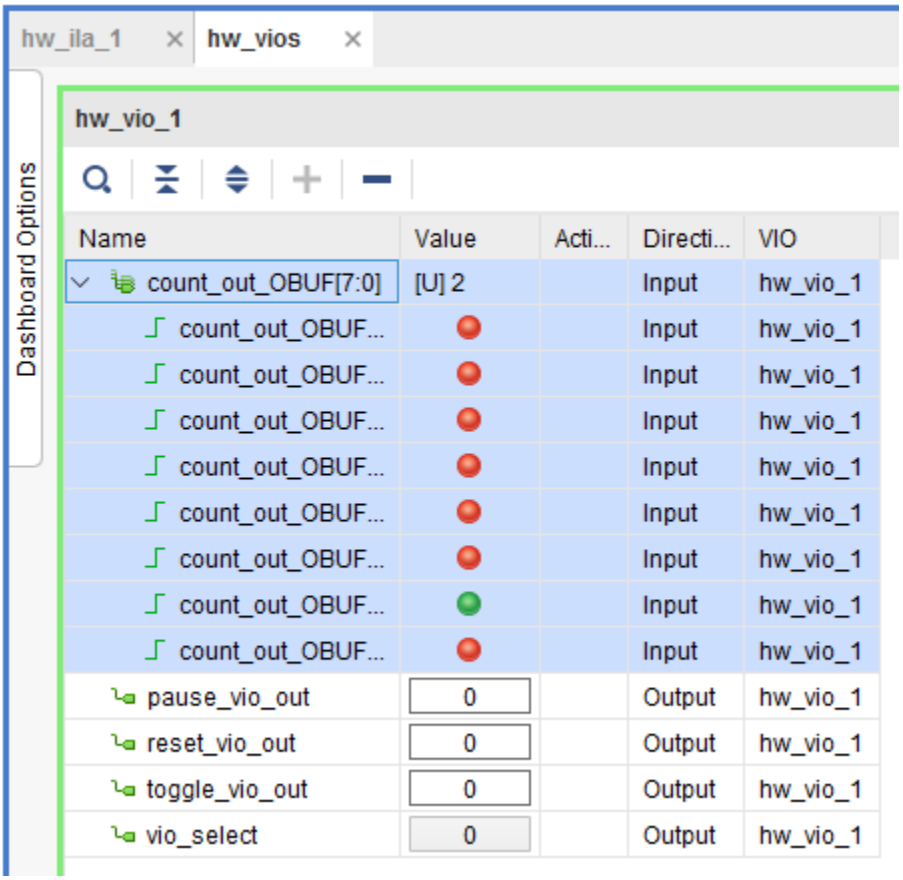
   This can be a local board or on a remote server. Exact details of how to accomplish this task depends on your setup. You can interact with this design remotely via the VIO and ILA debug cores.

2. Once you are connected to the hardware, right-click on the FPGA instance and select **Program Device**. The `top.bit` file should be selected by default from the `project_1.runs/impl_1` directory. If it is not, select `top.bit` from the `impl_1` project run directory. Note that the `top.ltx` probes file is automatically selected. This is a complete device bitstream that includes the multiplier RM.
3. Click on the hw_vio_1 dashboard tab. If it is not visible, open Dashboard Options and check the hw_vio_1 box.
4. Press the + button and select all the probes from the Add Probes dialog box, then click **OK**.

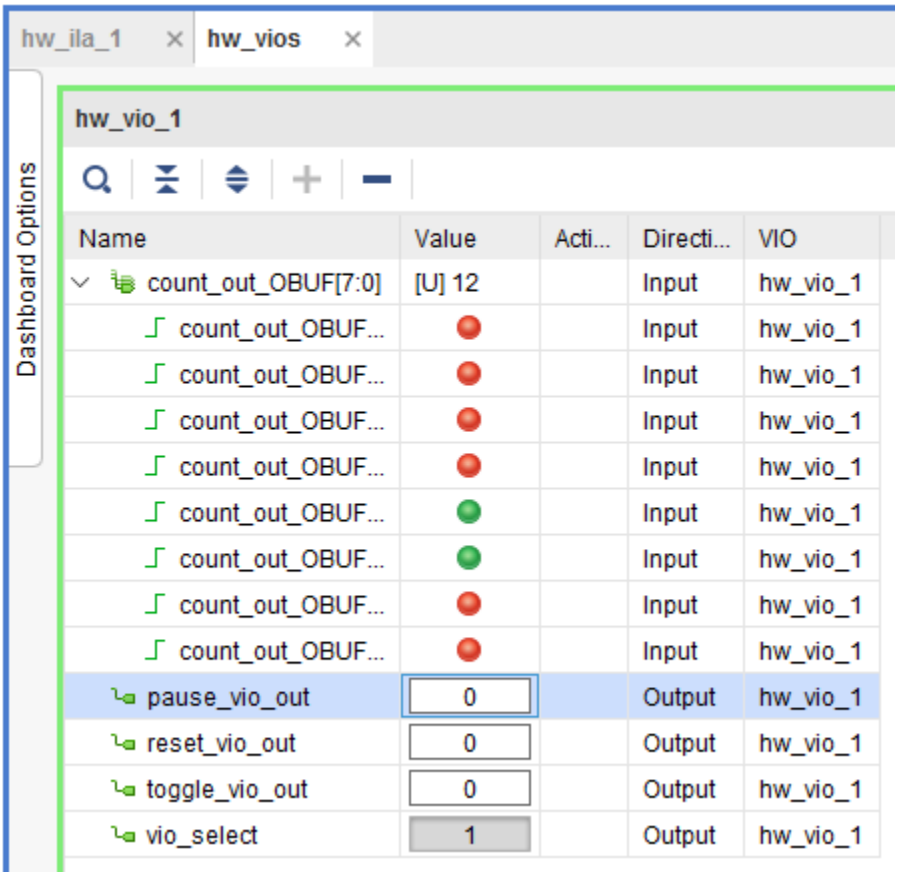


5. Right-click on the probes and set them up in the following manner:
   - **`count_out_OBUF[7:0] bus`**:
   - **`count_out_OBUF[7:0] individual bits`**: Radix: unsigned decimal
   - **`pause_vio_out`**: LED: low value `red`, high value `green`
   - **`reset_vio_out`**: Active `High` button
   - **`toggle_vio_out`**: Active `High` button
   - **`vio_select`**: Toggle button

   The resulting dashboard looks like this:

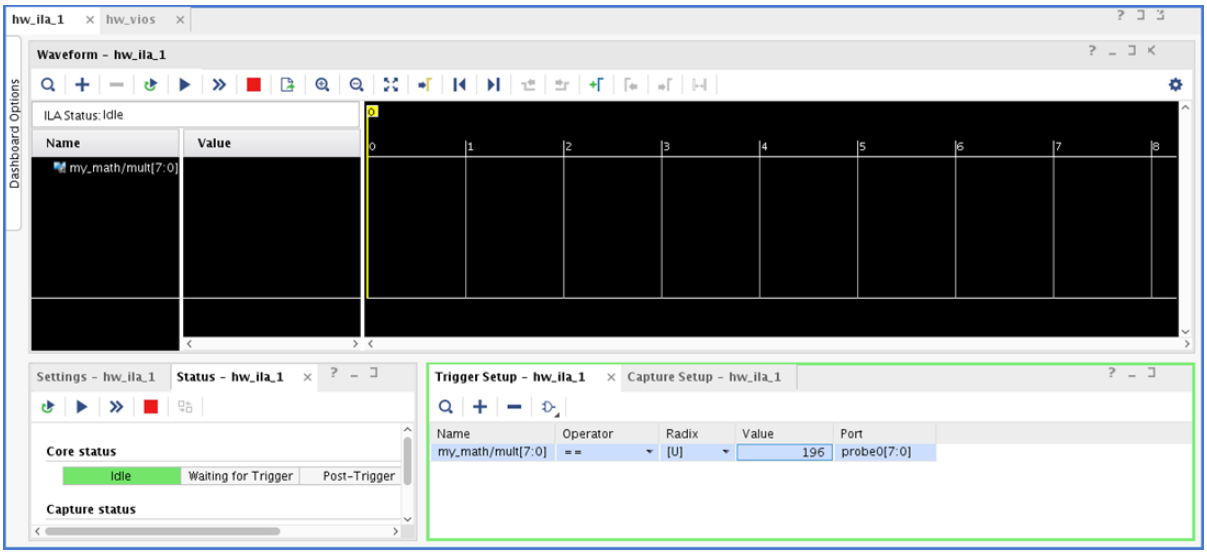


6. Change the `vio_select` value to a 1. This disables the buttons on the physical board and enables the pause, reset, and toggle buttons through the VIO.
7. Select the pause button by clicking on the Value field of `pause_vio_out`. The LED counter stop at a particular value. Take note of the unsigned binary value of the `count_out_OBUF`. In this screenshot, the value is 12.

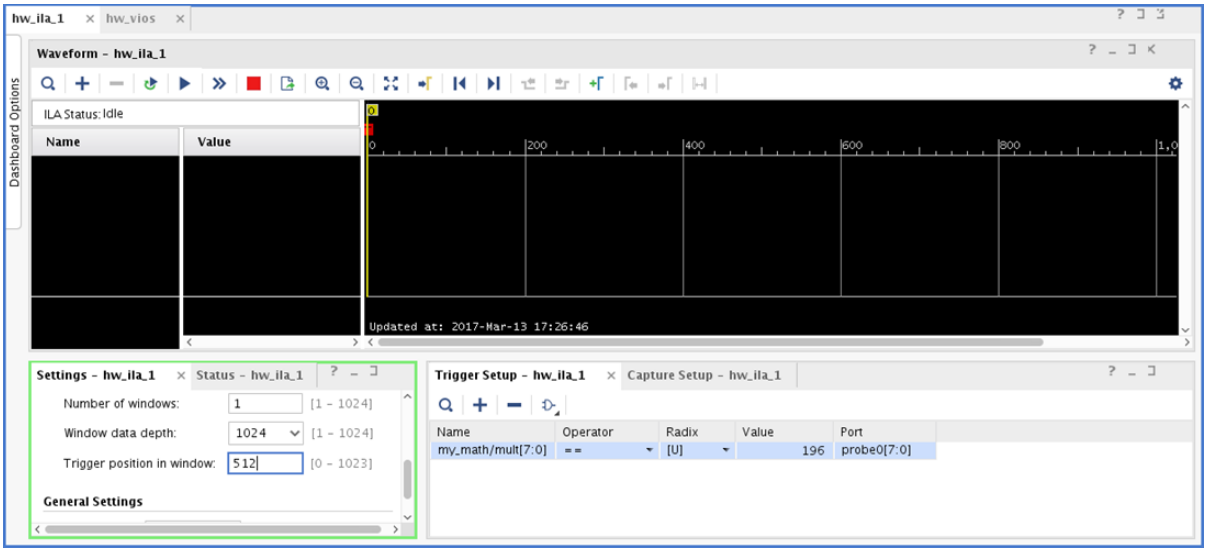


8. Press the `toggle_vio_out` button. The value of the `count_out_OBUF` bus is squared, as the current RM is a multiplier. In this case, 144.
9. Press the pause button again and the counter starts. The `count_out_OBUF` values now count by the square of 0 to 15. For example, 1, 4, 9, 16, 25, etc.

   Note: Given the relative frequencies of the internal clock and the sampling rate of the Hardware Manager, you might not see all values in the sequence.
10. Press the reset button to return the design to its default state. This count resumes to its initial 0 to 15 range.
11. Play with these buttons to understand the design. If you have a local board, you can toggle the `vio_select` and use the buttons on the board and the LEDs on the board to observe the same behavior.
12. Switch to the ILA dashboard. Up to this point you have used the VIO located in the static design. You can see the result of the multiplier, but if you want to observe the waveforms inside the RM, you can do this with the ILA located there.
13. In the Trigger Setup window, press the + button and add the `my_math/mult[7:0]` probes. Change the radix (in both the Trigger Setup and waveform windows) to unsigned decimal and set the value to 196 (for example 14x14).

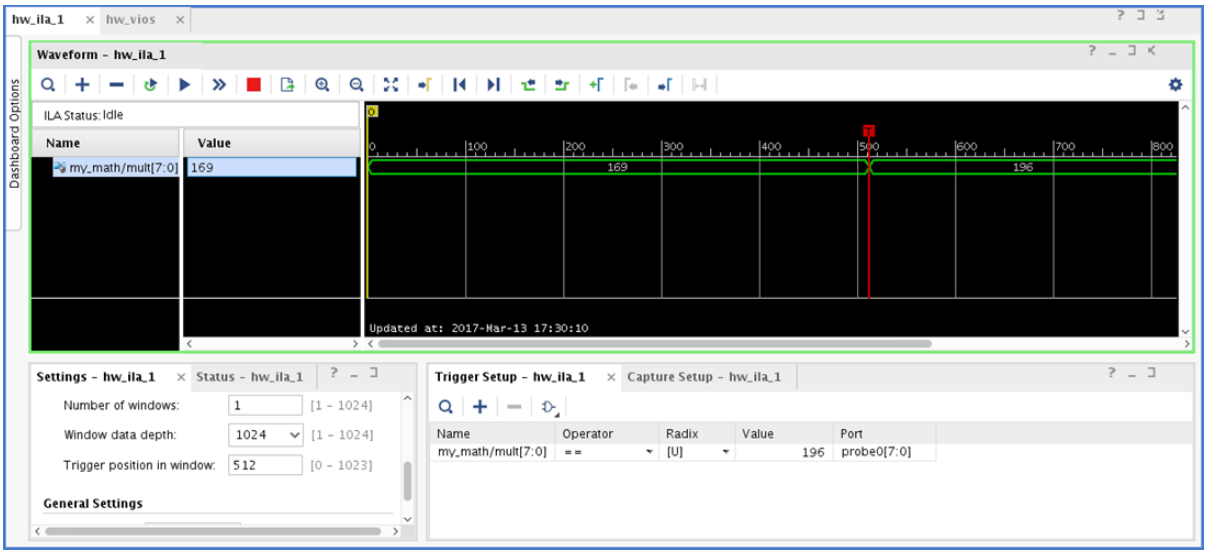


14. In the settings window for the ILA, change the trigger position in the window to 512.

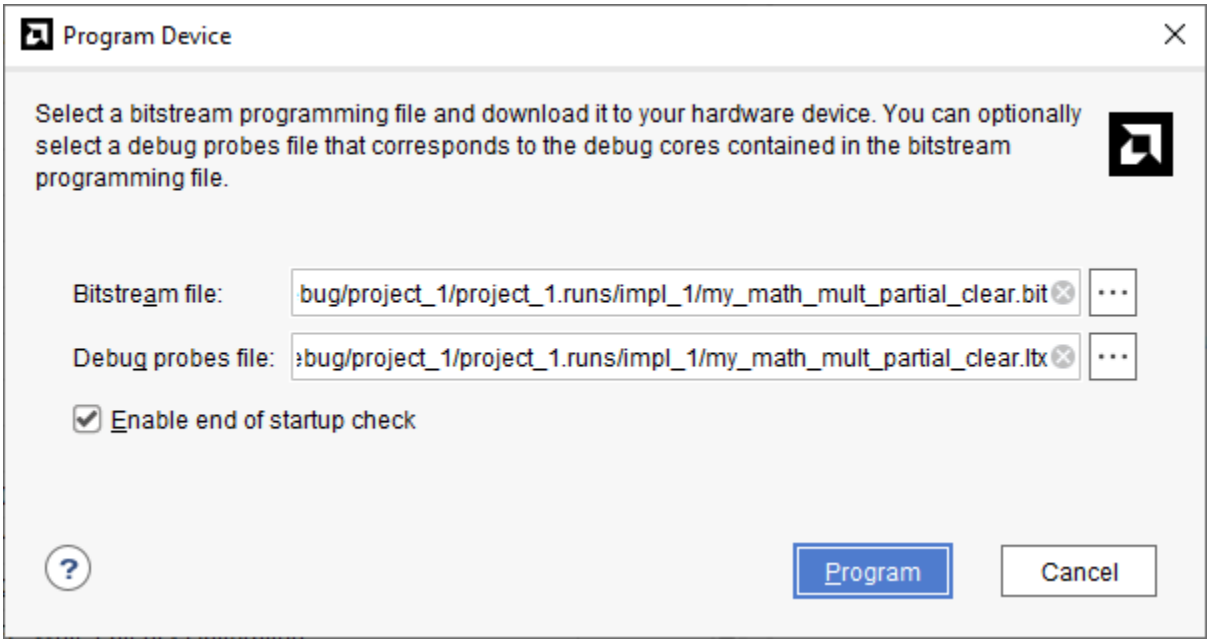


15. Click on the run trigger button in the waveform toolbar. You will see the transition of the waveform from 169 to 196 (for example 132 to 142).

    Note: Make sure the VIO does not have the design paused, or the trigger will not occur.



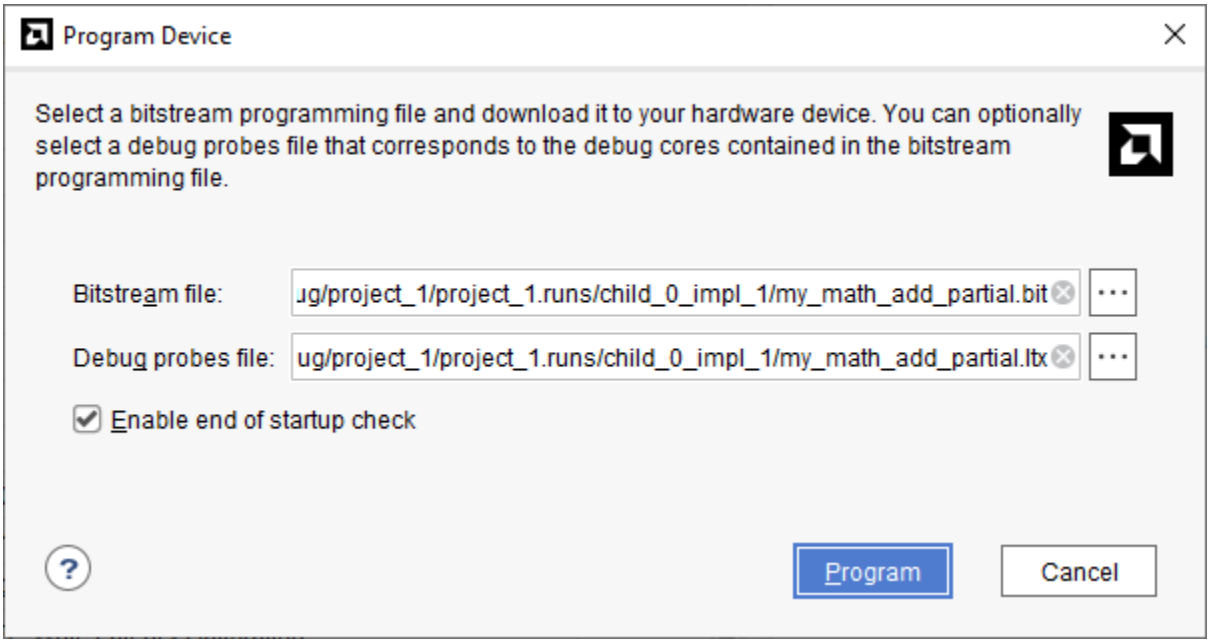
16. Now load the partial bitstream for the adder. Right-click on the target part in the Hardware view and choose **Program Device**.
17. If you are targeting an UltraScale part, you must first program the clearing bitstream to prepare the design for the next partial bitstream. For the bitstream file choose the multiplier clearing bitstream. Navigate to the `project_1.runs/impl1/` directory and choose the `my_math_mult_partial_clear.bit` file.



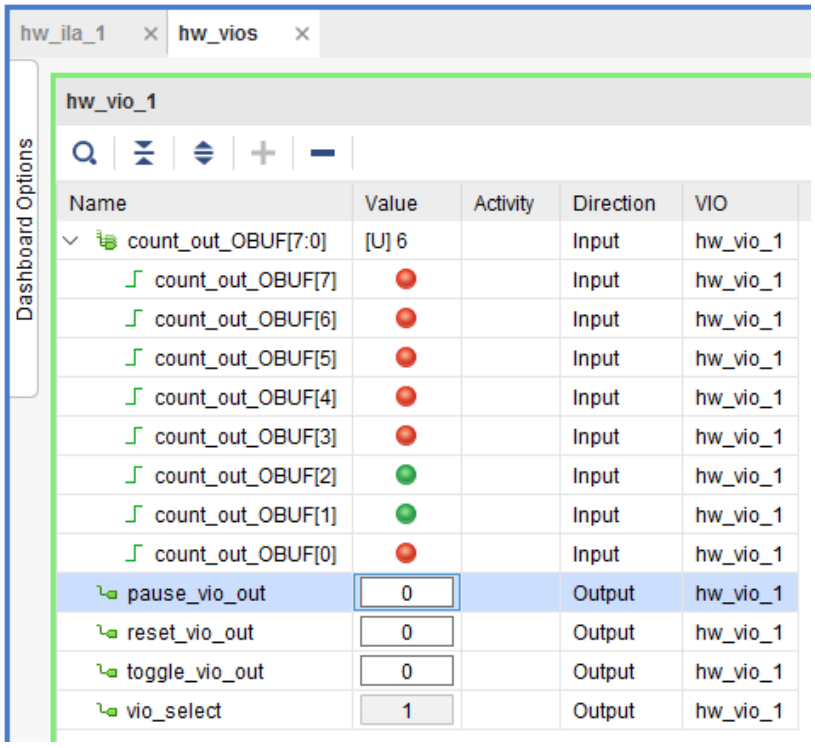
The paired LTX file will be picked up automatically. Click **Program**.

18. Switch to the VIO dashboard, and observe that the counter is still counting. If you press the toggle button to switch to the multiplier output, the value is held at 255. This is because the logic in the Reconfigurable Partition is currently disabled. Click the toggle button to switch back to the counter. Remember, vio_select must be set to a 1 to control remotely.

19. Right-click on the target part in the Hardware view and choose **Program Device**.
20. For the Bitstream file, navigate to `project_1.runs/child_0_impl_1/` and choose the `my_math_add_partial.bit` file.



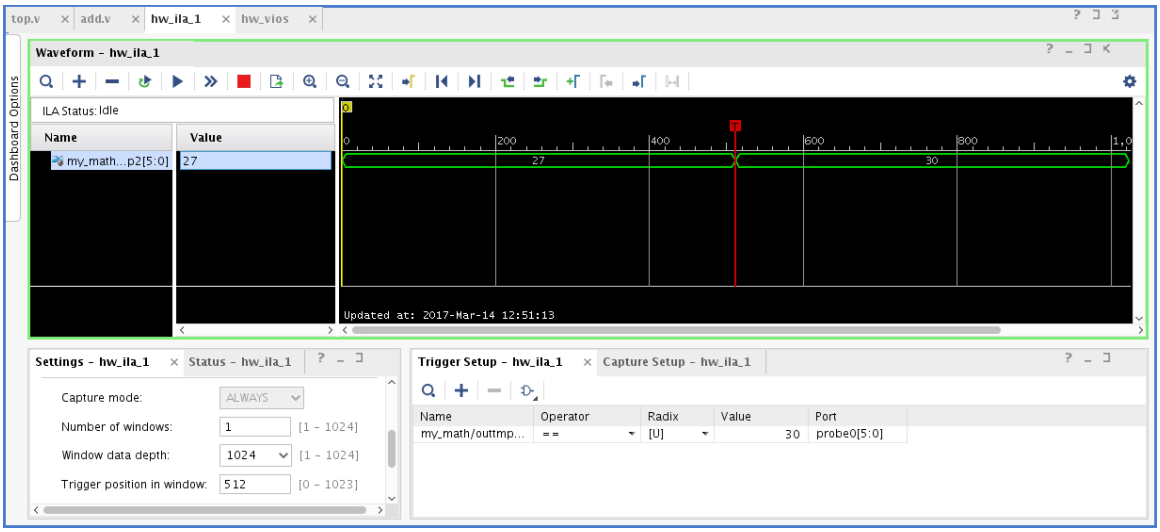
Once again, the matching LTX file will populate automatically. Click **Program**.



21. On the VIO dashboard, select pause. In this case, the value stopped at 6. After a toggle, the value is 18. The adder adds the same number 3 times.

22. Switch to the ILA dashboard. In the Trigger Setup window click + and add my_math/outtemp2[5:0] bus. Change the following settings for the trigger:
    - Radix = [U] (unsigned decimal)
    - Value = 30
23. In the ILA settings window change the trigger position to 512.
24. In the waveform window, click the + button and add the my_math/outtemp2[5:0] bus to the waveform. Right-click on the probe and change the radix to unsigned decimal.
25. In the waveform window, click the trigger button for the ILA. You will see the transition from 27 (9+9+9) to 30 (10+10+10).

    ***Note*:** Make sure the VIO does not have the design paused.



Close the Hardware Manager when you are satisfied that everything is functioning properly.

Lab 4 Conclusion

With the addition of RM-level debug, any part of a Dynamic Function eXchange design is debuggable. Users can easily switch between different RMs within the Hardware Manager to monitor design activity just as you would in a flat design.

Lab 5

DFX Controller IP for 7 Series Devices

Overview

This lab introduces the AMD Dynamic Function eXchange (DFX) Controller IP and its use in managing reconfiguration on 7 series devices. You customize the DFX Controller, defining virtual sockets and adding reconfigurable modules (RMs) to describe the reconfiguration needs of the target FPGA. You then compile the design, generate a PROM file, and set up the board. Finally, you operate the sample design in hardware, using push buttons to issue hardware triggers and the AXI4-Lite interface to query and modify the controller at runtime, including adding a new RM to an already deployed design. This lab targets the KC705, VC707, and VC709 development boards.

Step 1: Extract the Tutorial Design Files

1. Download the reference design files.
2. Extract the zip file contents to any write-accessible location.
3. In the extracted files, navigate to `\dfxc_7s`.

Step 2: Customizing the Dynamic Function eXchange (DFX) Controller IP

During customization of the DFX Controller IP, enter all required details. Identify information for each reconfigurable partition (RP) and reconfigurable module (RM) to create a fully populated controller. The controller uses this data to understand all reconfiguration needs for the target FPGA. Within this IP, reconfigurable portions of the design are called virtual sockets. A virtual socket includes the RP and all related static logic for managing it, such as decoupling or handshaking logic. You can customize core parameters during operation, but enter as much information as possible during this step. This improves the accuracy of the front-end design description compared to the final implemented design.

1. Open and access the IP:
   a. Open the Vivado IDE.
   b. In the Tasks section, click **Manage IP**.
   c. Select **New IP Location**.
   d. Enter the following details:
      - **Part:** Click the ellipses (...) to open the part selector. Click **Boards** to select your target. This lab supports the KC705, VC707, and VC709.
      - **IP Location:** `<Extract_Dir>/Sources/ip`
   e. Click **Finish**.
2. In the IP catalog, expand Foundational Elements. Expand DFX and double-click **DFX Controller**.



***Note*:** The DFX Controller IP GUI has four tabs on the left side that provide configuration feedback. Use the Validation tab to view errors that might arise as the core parameters are entered. The core does not compile with existing errors.

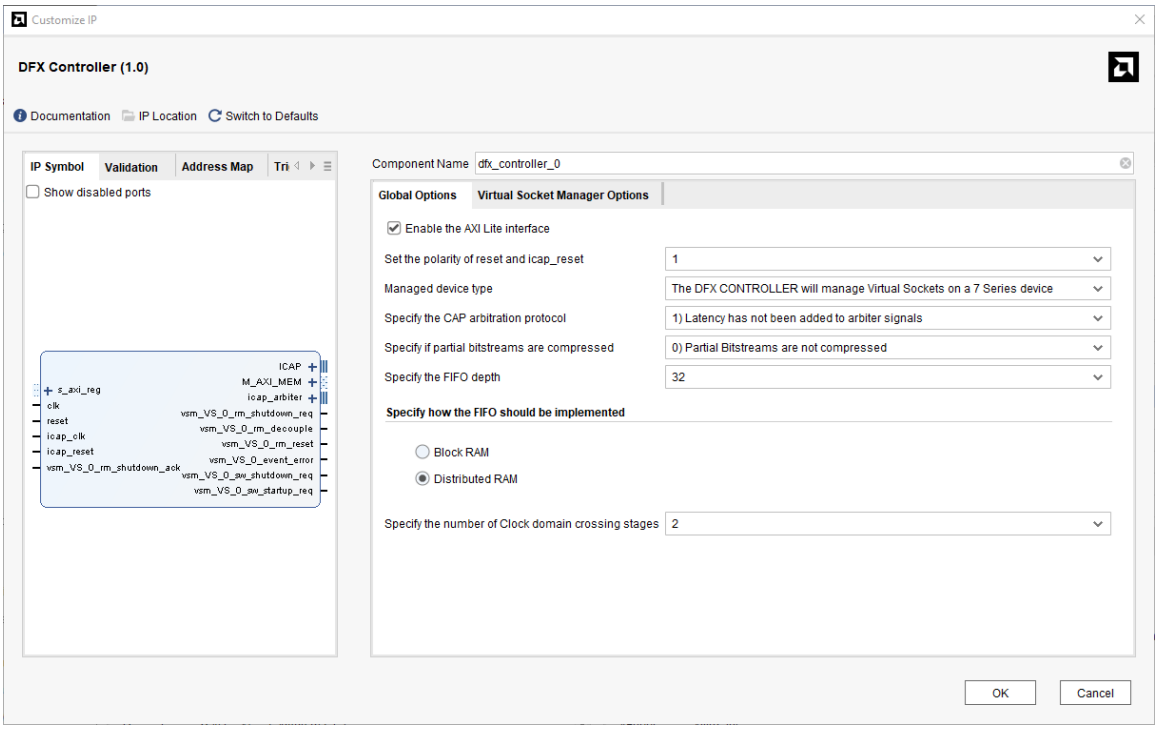
On the right, there are two tabs where all customization is done. Most of the information is entered on the Virtual Socket Manager Options tab.

3. Leave the component name as `dfx_controller_0`. The version of the DFX Controller used in the final design is automatically compiled in Step 3: Compiling the Design.
4. On the Global Options tab, make three changes:
   a. Change **Set the polarity of reset and icap_reset** to 1.

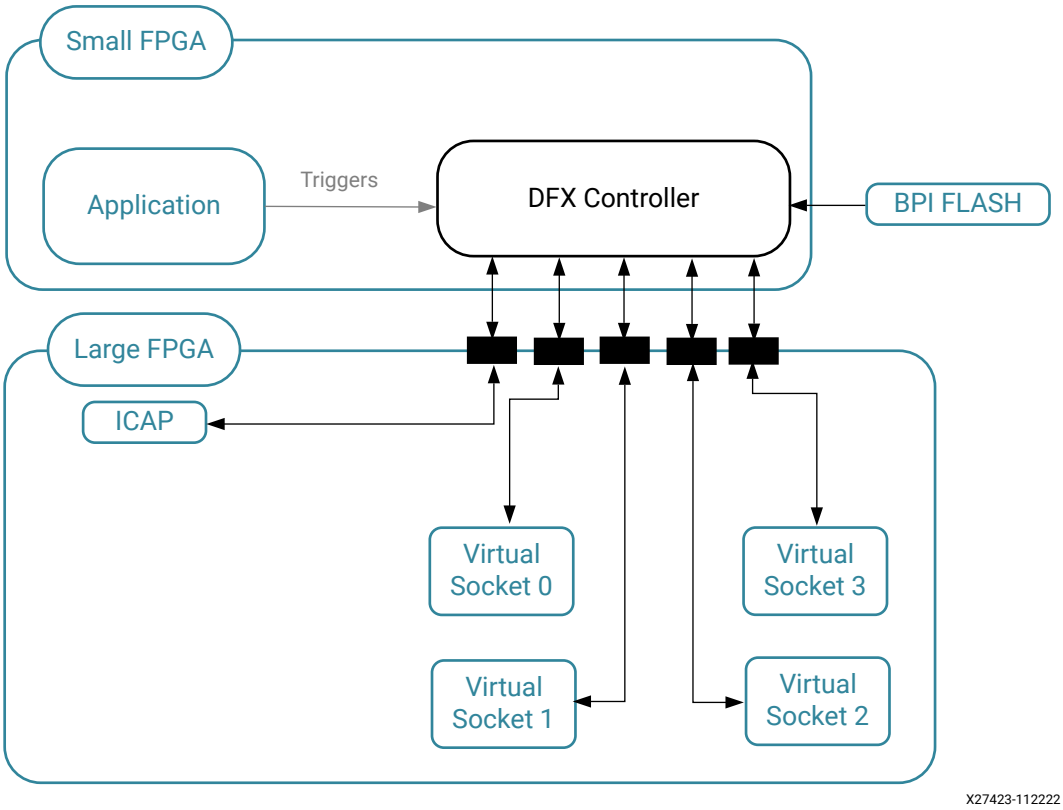
b. Change **Specify the CAP arbitration protocol** to 1) Latency has not been added to arbiter signals

c. Change **Specify the number of clock domain crossing stages** to 2.

Ensure Managed device type is set to 7 series. The DFX Controller Graphic User Interface should now look like this:



Note: This DFX Controller can manage virtual sockets on 7 series, UltraScale, and UltraScale+ devices. This IP is not limited to managing reconfiguration on the same device on which it resides. It can connect to an ICAP on another device to manage its reconfiguration.



5. Switch to the Virtual Socket Manager Options tab to define information about the Virtual Sockets and their Reconfigurable Modules.

   The DFX Controller IP is preloaded with one Virtual Socket with one Reconfigurable Module to get you started.

   First, define the virtual socket manager (VSM) for the shift functionality.

6. Rename the current VSM from **VS_0** to **vs_shift** in the Virtual Socket Name field.
7. Rename the current RM from **RM_0** to **rm_shift_left** in the Reconfigurable Module Name field.

   CAUTION!

   - To accept a new value in any field in the GUI, simply click in any other field in the GUI or press the Tab key. Do NOT press Enter, as this triggers compilation of the IP.
   - Underscores are not visible in the Virtual Socket Manager to configure and Reconfigurable Module to configure pull-down menus. The Virtual Socket Name and Reconfigurable Module Name fields below the pull-down menus show this more accurately.

8. Click the **New Reconfigurable Module** button to create a new RM for this VSM. Notice the form has changed. The new, generic reconfigurable module name is `RM_1`. Name it `rm_shift_right`.

   TIP: *Up to 128 reconfigurable modules can be managed by a single virtual socket manager.*

9. Configure the `vs_shift` VSM to have the following properties:
   - Has Status Channel = checked
   - Has PoR RM = checked, value set to **rm_shift_right**
   - Number of RMs allocated = **4**

     The PoR RM indicates which RM is contained within the initial full-design configuration file, so the VSM knows which triggers and events are appropriate upon startup of the FPGA. The VSM tracks the current active Reconfigurable Module in its socket.

     Even through you have only defined two RMs for this Virtual Socket, you have set aside space for four in total. This allows for expansion later on. Additional Reconfigurable Modules can be identified using the AXI4-Lite interface, but only if spaces have been reserved for them.
10. For each of these RMs, enter the following values. Use the **Reconfigurable Module to configure** pull-down to switch between the two RMs.
    - For rm_shift_left:
      - Reset type = **Active High**
      - Duration of Reset = **3**
    - For rm_shift_right:
      - Reset type = **Active High**
      - Duration of Reset = **10**

    ***Note*:** The different reset durations are given to show that these can be independently assigned, as each RM might have different requirements. Reset durations are measured in clock cycles.
11. For each RM, assign a bitstream size and location to identify where it resides in the BPI flash device. These values differ based on the target board.
    - When targeting the KC705:
      - For rm_shift_left:
        - Bitstream 0 address = **0x00AEA000**
        - Bitstream 0 size (bytes) = **482828**
      - For rm_shift_right:
        - Bitstream 0 address = **0x00B60000**
        - Bitstream 0 size (bytes) = **482828**
    - When targeting the VC707:
      - For rm_shift_left:
        - Bitstream 0 address = **0x01355C00**

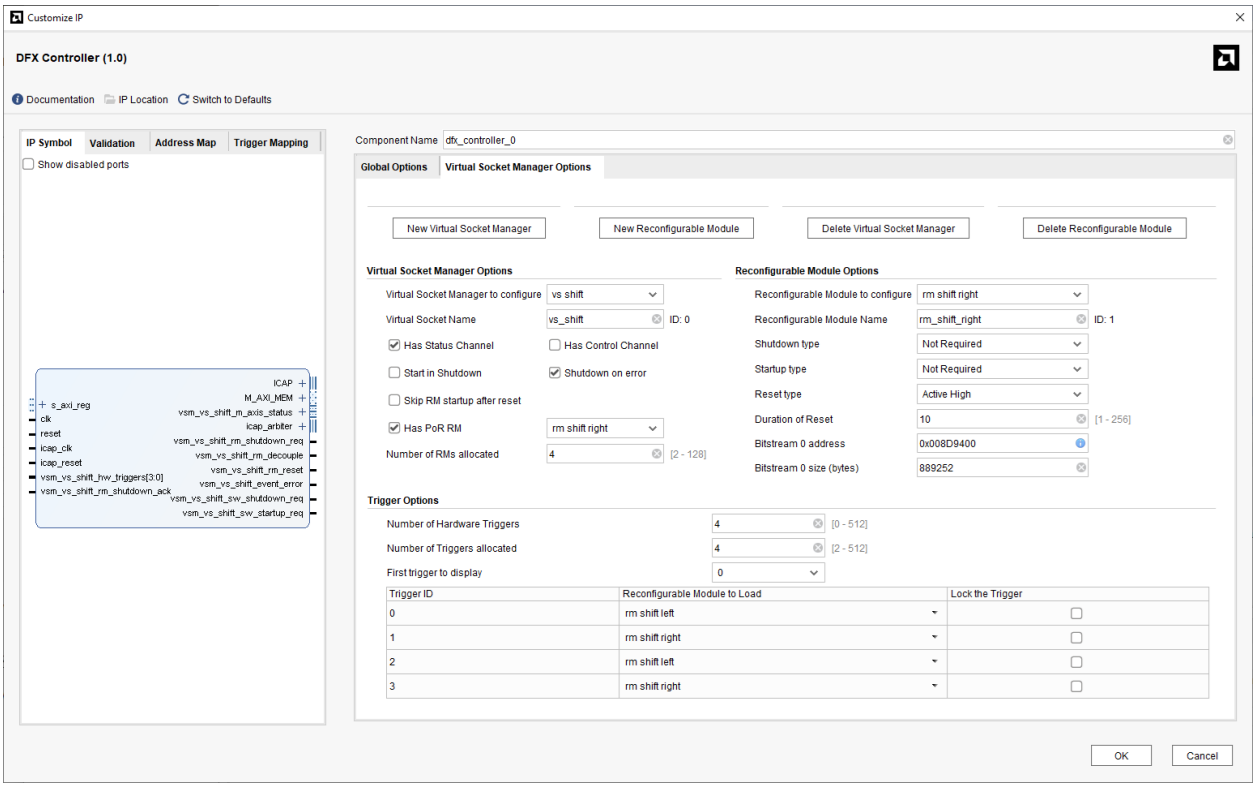
    - Bitstream 0 size (bytes) = **708260**
  - For rm_shift_right:
    - Bitstream 0 address = **0x01402C00**
    - Bitstream 0 size (bytes) = **708260**
- When targeting the VC709:
  - For rm_shift_left:
    - Bitstream 0 address = **0x00800000**
    - Bitstream 0 size (bytes) = **889252**
  - For rm_shift_right:
    - Bitstream 0 address = **0x008D9400**
    - Bitstream 0 size (bytes) = **889252**

  This information is typically not known early in design cycles, as bitstream size is based on the size and composition of the Reconfigurable Partition Pblock, and bitstream address is based on storage details. Until the design is to be tested on silicon, these can be set to 0. As the design settles and hardware testing with the DFX Controller is set to begin, this information can be added. The bitstream address information must match the information passed during PROM file generation. Certain bitstream generation options, most notably bitstream compression, can lead to variations in the final bitstream size for different configurations, even for the same Reconfigurable Partition.

12. Define the Trigger Options for the shift functionality:
    - Number of Hardware Triggers = **4**
    - Number of Triggers allocated = **4**

    The four trigger assignments are done automatically. These can be modified during device operation using AXI4-Lite, which is especially useful when you have added a new RM through the same mechanism during a field system upgrade.

    At this point, the IP GUI should look like this (showing rm_shift_right for the VC709 here):



Next, you create and populate the Count Virtual Socket following the same basic steps, with slightly different options.

13. Click the **New Virtual Socket Manager** button to create a new VSM.
14. Select the **New Reconfigurable Module** button to add two RMs with these names and properties:
    - Reconfigurable Module Name = **rm_count_up**
        - Reset type = **Active High**
        - Duration of Reset = **12**
    - Reconfigurable Module Name = **rm_count_down**
        - Reset type = **Active High**
        - Duration of Reset = **16**

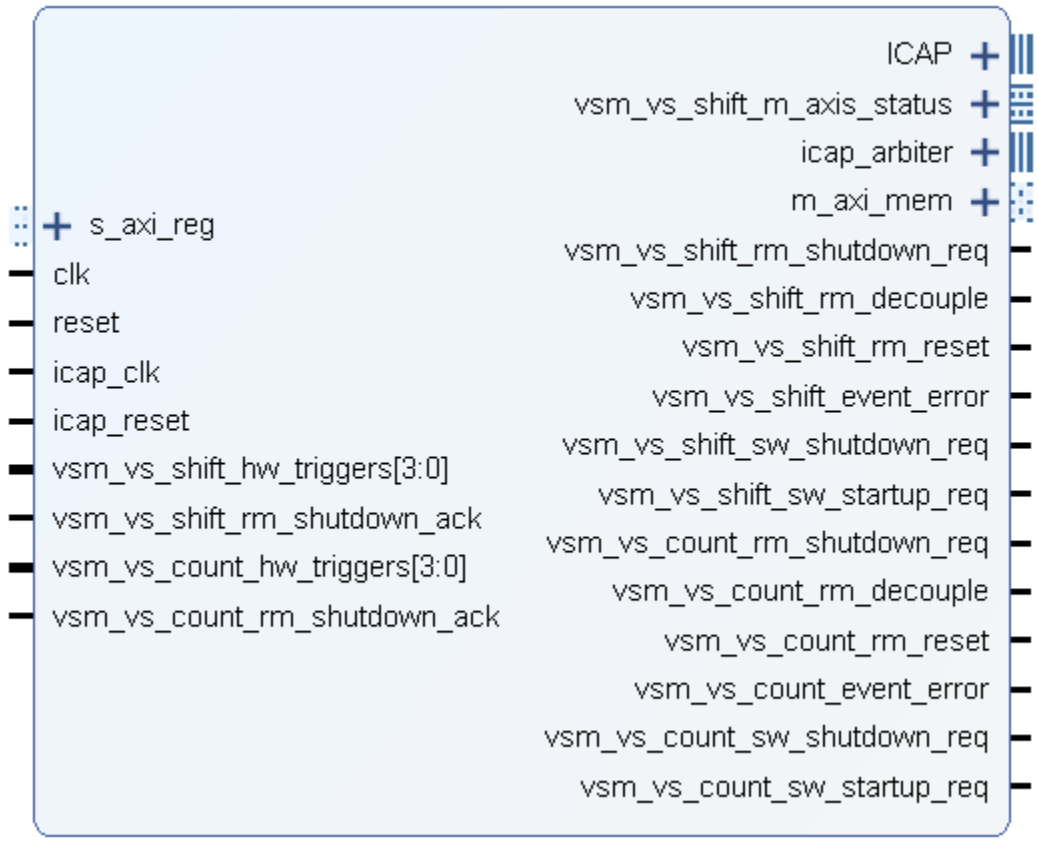
    For this Virtual Socket, leave the bitstream address and size information at the default of 0. In addition to being defined here, bitstream size information can be added to a routed configuration checkpoint via the DFX Controller Tcl API, or can be added in an active design using the AXI4-Lite interface. For the Count Virtual Socket, the bitstream address and size information is added using the Tcl commands after place and route, but before bitstream generation.

***Note*:** For more information about how to use the DFX Controller Tcl API, see the *Dynamic Function eXchange Controller IP LogiCORE IP Product Guide* (PG374).

Examine the Tcl scripts in the `<Extract_Dir>/Sources/scripts` directory. The `update_dfxc_<board>.tcl` uses the DFX Controller Tcl API to update the bitstream address and bitstream size, which is stored in `dfx_info_<board>.tcl`. This file is sourced later in the lab from `<Extract_Dir>/design.tcl`.

15. On the Virtual Socket Manager Options tab, modify these VSM settings from their default values:
    - Virtual Socket Name = **vs_count**
    - Start in Shutdown = checked
    - Shutdown on error = unchecked
    - Has PoR RM = checked, value set to **rm_count_up**
16. Define the Trigger Options for the Count functionality:
    - Number of Hardware Triggers = **4**
    - Number of Triggers allocated = **4**

    This completes the planned customization of the DFX Controller IP for this tutorial.
17. To begin core compilation and out-of-context synthesis, click **OK**, and then click **Generate**.



Step 3: Compiling the Design

The DFX Controller IP is created, but the design is not yet compiled. In order to create the PROM image with all the necessary full and partial images, source the following scripts in Tcl mode using the commands below.

IMPORTANT! *Before running this Tcl script, open it to set the value of the xboard variable. KC705 is the default, but VC707 or VC709 can be selected.*

- `vivado -mode tcl -source design.tcl`:

  Sourcing design.tcl generates all the necessary IP (including the DFX Controller), synthesizes and implements the entire design (three configurations), updates the vs_count VSM using the DFX Controller Tcl API, and generates bitstreams.

Note that the customization of the IP is scripted. Examine the `gen_ip_<board>.tcl` script in `<Extract_Dir>/Sources/scripts` to see all these parameters defined for automated IP creation, the DFX Controller, and others. The DFX Controller instance you create using the IP GUI is not actually used for the full design processing, so you do not have to complete Step 2 to compile the entire design.

- `vivado -mode tcl -source create_prom_file_<board>.tcl`:

  Sourcing the board-specific create_prom_file.tcl creates the PROM image for the target board. This script contains hard-coded values for bitstream address for the entire project. If this design is modified in such a way that changes bitstream sizes, full or partial, then these values must also change. Changes that affect bitstream sizes include changing the target device, changing the size or shape of the Pblocks, or introducing bitstream options such as compression or per-frame CRC.

  This script defines PROM file options by setting properties and then making calls to write_cfgmem. The DFX Controller works in byte addresses because the data is stored in bytes in AXI. This linear flash PROM uses half word addresses because it stores data in half words (16 bits). Divide the ROM address by 2 to get the AXI address. For example, the shift_left address for the KC705 is given as 00AEA000 in during DFX Controller customization and 00575000 (half that value) for write_cfgmem. Note that the starting addresses are always multiples of 1024 (0x0400) to ensure that each bitstream starts on a byte address boundary. Also note that the initial configuration file for the VC709 is compressed in order to fit in the linear flash; the address for the first partial bitstream that follows is padded to allow expansion of that initial configuration file.

  Supplied in the lab directory is a file called dfxc_bitstream_sizes_lab5.xlsx. In this file, bitstream sizes are entered by the user based in the yellow highlighted fields. It calculates the starting address in hex for each partial bitstream at the next byte boundary. Values in blue are to be supplied for DFX Controller IP customization, in either the DFX Controller IP GUI, in the `gen_ip_<board>.tcl` script, or in `dfx_info_<board>.tcl` which is used for post-route API modification. The values in green are addresses divided by two to be used in PROM file generation in the `create_prom_file_<board>.tcl` script.

Step 4: Setting Up the Board

Once the partially reconfigurable design is in operation, you can connect to and communicate with the core to check status, deliver triggers and make modifications.

1. Prepare the target board for programming.
   a. Connect the JTAG port to your computer via the micro-USB connection.
      - For the KC705: U59

- For the VC707 or VC709: U26

b. Set the configuration mode to 010 (BPI) by setting the Address DIP Switch (SW13) to 00010 (bit 4 is high).

c. Turn on the power to the board.

2. Open the Vivado IDE.
3. Select **Flow → Open Hardware Manager**
4. Click on the **Open Target** link and select **Auto Connect** for the device to be recognized.
5. To program the BPI configuration flash, right-click the device (e.g., xc7k325t_0) and select Add Configuration Memory Device.
6. From the list shown, select the appropriate linear BPI flash and click OK twice.
   - For the KC705, select **28f00ap30t**
   - For the VC707 or VC709, select **28f00ag18f**
7. In the Configuration file field, search the tutorial directory for dfx_prom.mcs found in the bitstreams subdirectory. Click **OK** to select this file, and then click **OK** to program the flash.

   At this point, the board is ready to operate with the tutorial design. Any power-cycle or hard reset automatically programs the AMD FPGA with this sample design.

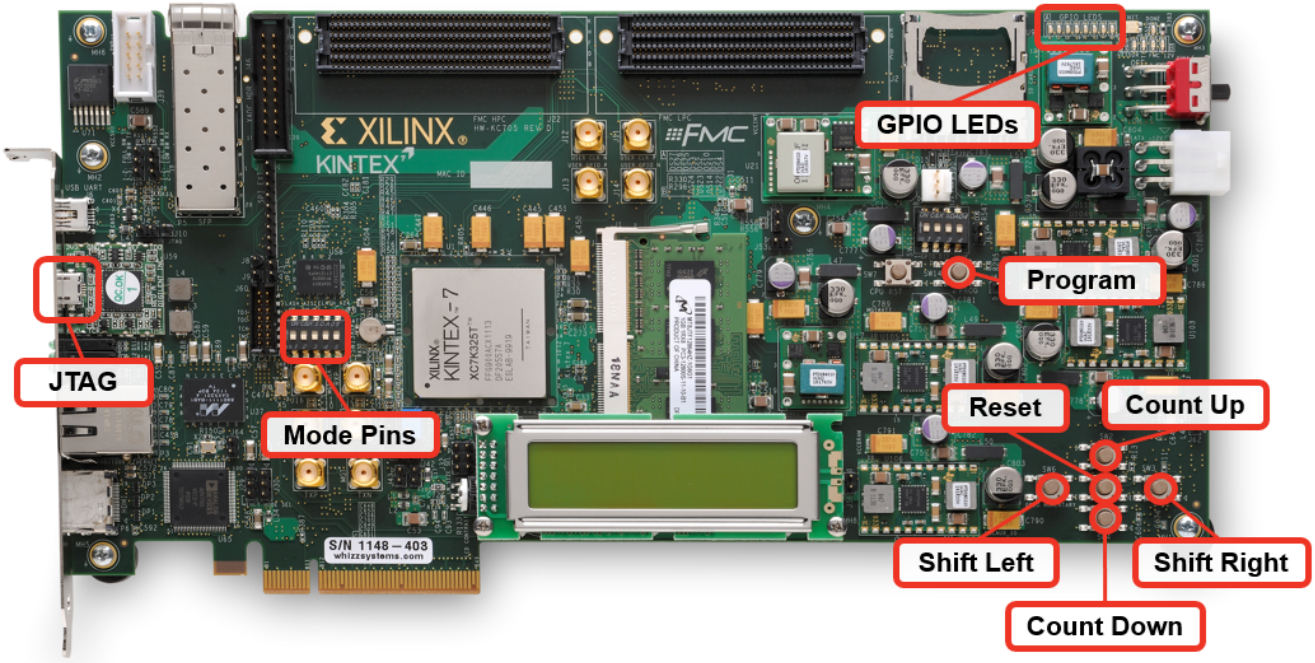
Step 5: Operating the Sample Design

Position the board so that the text is readable. The LCD screen is on the side closest to you, with the power connection on the right and the JTAG connection on the left. The buttons of interest are the five user push buttons in the lower right corner, plus the PROG push button in the middle right.

Their functions for the KC705 are as follows:

- PROG (SW14) – program the device from the BPI flash
- North (SW2) – load the Count Up partial bit file
- South (SW4) – load the Count Down partial bit file
- East (SW3) – load the Shift Right partial bit file
- West (SW6) – load the Shift Left partial bit file
- Center (SW5) – reset the design

Figure 2: **Push Buttons, Switches, and Connections on the KC705 Demonstration Board**



Their functions for the VC707 and VC709 are as follows:

- PROG (SW9) – program the device from the BPI flash
- North (SW3) – load the Count Up partial bit file
- South (SW5) – load the Count Down partial bit file
- East (SW4) – load the Shift Right partial bit file
- West (SW7) – load the Shift Left partial bit file
- Center (SW6) – reset the design

Figure 3: **Push Buttons, Switches, and Connections on the VC707 Demonstration Board**

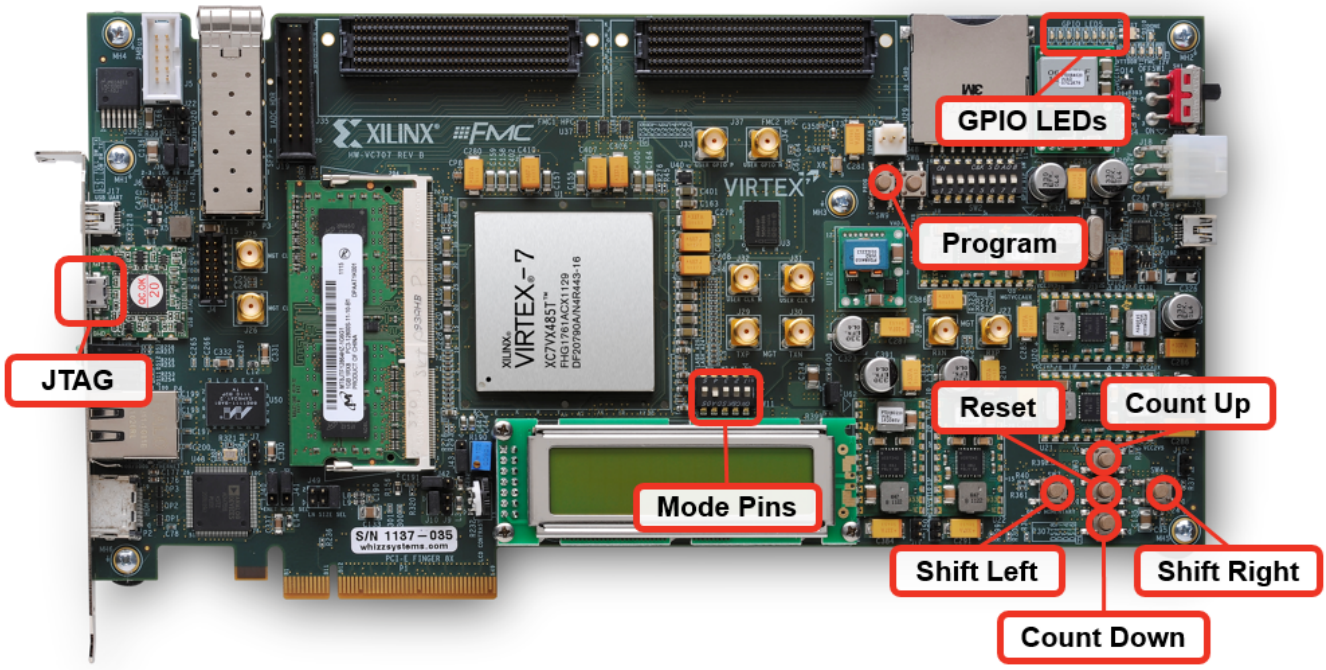
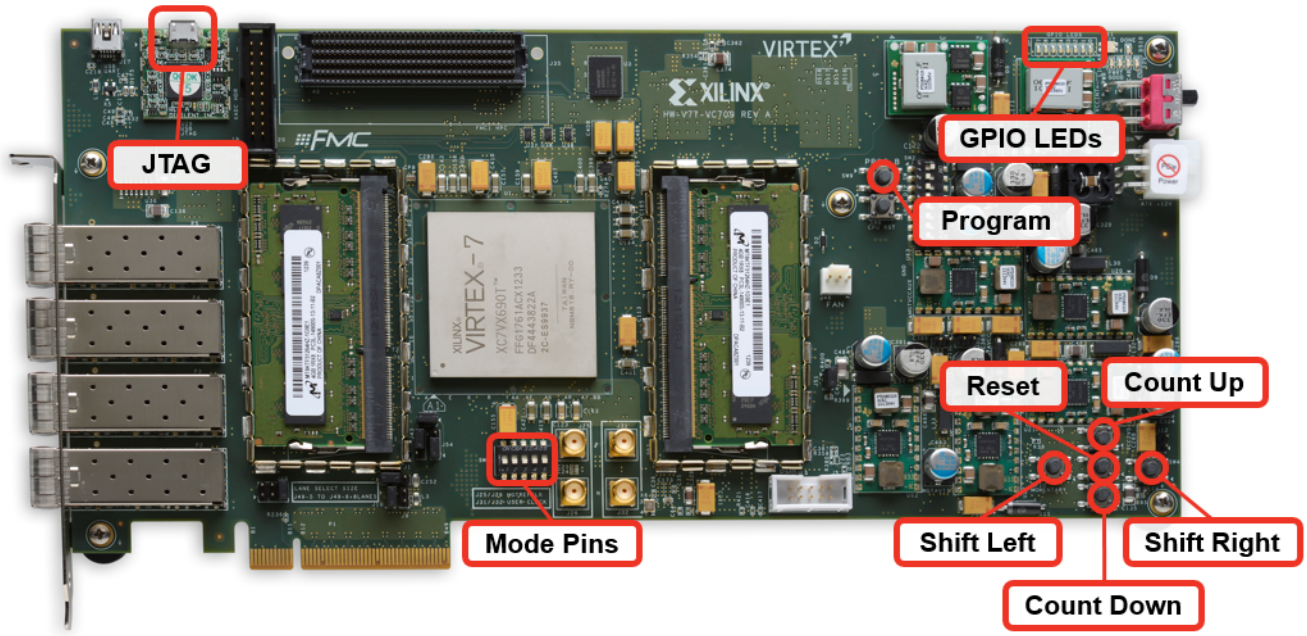


Figure 4: **Push Buttons, Switches, and Connections on the VC709 Demonstration Board**



1. Program the FPGA by pressing the **PROG** pushbutton. The 8 GPIO LEDs in the upper-right corner will start operation after the DONE LED goes high.

   At this point, the four bits on the left of the GPIO bank are counting up, and the four bits on the right are shifting to the right.

2. Press the **Shift Left** and **Shift Right** buttons alternately.

   With each push, a partial bit file is pulled from the BPI flash by the DFX Controller and delivered to the ICAP, changing the functionality in that Reconfigurable Partition. When this happens, the LED shift direction changes, depending on the button pushed.

3. Press the **Count Down** and **Count Up** buttons alternately.

   With each push, nothing happens. When configuring the DFX Controller, the Counter Virtual Socket was programmed to begin in Shutdown mode. It does not respond to any hardware or software triggers until it is moved to Active mode.

Step 6: Querying the DFX Controller in the FPGA

In this step, you interact with the core via JTAG from the Hardware Manager to understand the status of the core and issue software triggers.

In the Vivado Hardware Manager, you might need to select Refresh Device to establish the link to the device over the JTAG connection. Notice the XADC as well as 6 ILA cores and the hw_axi link shown under the device in the Hardware view.

1. In the Tcl Console, cd into the DFX Controller tutorial directory then source the AXI4-Lite command Tcl script.

   ```
   source ./Sources/scripts/axi_lite_procs.tcl
   ```

   This enables a set of procedures that make the subsequent interaction with the DFX Controller easier. Examine this file to see how these procedures are defined. Note that these are written explicitly (hard-coded) for this design, the references to Virtual Sockets in any other design will need to be modified. For more information on this topic, consult the *Dynamic Function eXchange Controller IP LogiCORE IP Product Guide* (PG374).

2. Source the procedure to establish communication with the DFX Controller.

   ```
   dfxc_jtag_setup
   ```

3. Check the state of each Virtual Socket to see if they are in Shutdown or not.

   ```
   is_vsm_in_shutdown vs_shift
   is_vsm_in_shutdown vs_count
   ```

You should see that the Shift Virtual Socket is in Active mode (value = 0), and the Count Virtual Socket is in Shutdown mode (value = 1).

4. Examine the status of each Virtual Socket.

```
dfxc_decode_status vs_shift
dfxc_decode_status vs_count
```

Before examining the data returned, reference Table 9 in the *Dynamic Function eXchange Controller IP LogiCORE IP Product Guide* (PG374). The table in that section defines the values in the STATUS register. While this is a 32-bit register, you only need to pay attention to the lowest 24 bits, as the upper 8 bits are used for Virtual Socket Managers (VSM) in UltraScale devices.

The status of vs_shift is 263, which is 0000_0000_0000_0001_0000_0111 in binary. The status for vs_shift can also be 7, where the only difference is that RM_ID is now 0.

- RM_ID (bits 23:8) = 1. This means RM 1 is loaded (rm_shift_right). It might also appear as RM_ID (bits 24:8) = 0. This means RM 0 is loaded (rm_shift_left).
- SHUTDOWN (bit 7) = 0. This VSM is not in the shutdown state.
- ERROR (bits 6:3) = 0000. There are no errors.
- STATE (bits 2:0) = 111. The Virtual Socket is full.

The status of vs_count is 129, which is 0000_0000_0000_0000_1000_0001 in binary.

- RM_ID (bits 23:8) = 0. This means RM 0 is loaded (rm_count_up).
- SHUTDOWN (bit 7) = 1. This VSM is in the shutdown state.
- ERROR (bits 6:3) = 0000. There are no errors.
- STATE (bits 2:0) = 001. RM_SHUTDOWN_ACK is 1, as this VSM is executing the hardware shutdown step.

  These explicit details are reported in the breakdown of the status register in the return value from this Tcl proc.

5. Send a software trigger to the Shift Virtual Socket.

```
dfxc_send_sw_trigger vs_shift 1
dfxc_send_sw_trigger vs_shift 0
```

Remember that values of 0 and 2 correspond to shift left, and values of 1 and 3 correspond to shift right, as defined during DFX Controller customization.

6. Check the configurations of the RMs for the Count Virtual Socket.

```
dfxc_show_rm_configuration vs_count 1
dfxc_show_rm_configuration vs_count 0
```

The values for the bitstream sizes and address are reported here. These values could then be modified to account for necessary adjustments to the size or location of the bitstream. Different indices can be added to insert new RMs. Note that this query cannot be done for vs_shift, as the vs_shift VSM is not in the shutdown state.

7. Move the Count Virtual Socket Manager into active mode.

```
dfxc_restart_vsm_no_status vs_count
```

The Count Up and Count Down pushbuttons can now be used to load these partial bitstreams using the DFX Controller.

Step 7: Modifying the DFX Controller in the FPGA

In the final step, you add a new Reconfigurable Module to the Shifter VSM. In the create_prom.tcl script, you can see that two black box modules have already been generated. These represent two new RMs that may have been created after the static design was deployed to the field. You modify the DFX Controller settings to access one of these RMs by assigning the size, address, properties and trigger conditions.

1. Shut down the Shift VSM so it can be modified.

```
dfxc_shutdown_vsm vs_shift
```

Currently, RM ID 2 has the same mapping as the partial bit file for RM ID 0, so the same shift left partial bitstream would be loaded. This is the behavior as requested when the initial trigger mapping was done during core customization.

2. Check the status of the first three RM IDs to see their register bank assignments.

```
dfxc_show_rm_configuration vs_shift 0
```

```
dfxc_show_rm_configuration vs_shift 1
```

```
dfxc_show_rm_configuration vs_shift 2
```

3. When the MCS file is created for the prom, it adds additional blanking RMs that are already loaded into the BPI flash. Use this sequence of commands to reassign the trigger mapping for slot 2 to point to the blanking Reconfigurable Module for vs_shift.

```
dfxc_write_register vs_shift_rm_control2 0
```

This defines the settings for the RM_CONTROL register for slot 2. No shutdown, startup, or reset are required. Note how for the other two slots, the differing reset durations lead to different control values.

```
dfxc_write_register vs_shift_rm_bs_index2 2
```

This assigns a new bitstream reference for this RM ID.

```
dfxc_write_register vs_shift_trigger2 2
```

This assigns the trigger mapping such that trigger index 2 retrieves RM 2.

```
dfxc_show_rm_configuration vs_shift 2
```

This shows the current state of RM ID 2. Note the changes from the prior call to this command.

4. Complete the RM ID 2 customization by setting the bitstream details.

   For the KC705:

   ```
   dfxc_write_register vs_shift_bs_size2 482828
   ```

   ```
   dfxc_write_register vs_shift_bs_address2 13496320
   ```

   For the VC707:

   ```
   dfxc_write_register vs_shift_bs_size2 708260
   ```

   ```
   dfxc_write_register vs_shift_bs_address2 23108608
   ```

   For the VC709:

   ```
   dfxc_write_register vs_shift_bs_size2 889252
   ```

   ```
   dfxc_write_register vs_shift_bs_address2 11960320
   ```

5. Restart the VSM and then issue trigger events to it using software, as there is no pushbutton assigned for slot 2.

   ```
   dfxc_restart_vsm_no_status vs_shift
   ```

   ```
   dfxc_send_sw_trigger vs_shift 2
   ```

   Switch between values of 0, 1 and 2 to reload different partial bitstreams. The blanking bitstream in slot 2 removes the shifter function, so no activity on the LEDs is seen.

   Note that this same sequence of events could not be performed for the Count VSM as it is currently configured, even knowing that the PROM image has a Count black box partial bitstream sitting at (for the KC705) address 13979648 with a size of 541812. During DFX Controller customization, this VSM was selected to have only 2 RMs allocated, so expansion is not permitted.

Lab 5 Conclusion

This concludes lab 5. In this lab, you:

- Customized the Dynamic Function eXchange (DFX) Controller IP.
- Created a Virtual Sockets and added RMs to them.
- Complied the design and created a PROM file.
- Programmed the linear flash on the KC705, VC707 or VC709 board.
- Used pushbuttons to issue hardware triggers.
- Used the AXI4-Lite interface to check the core status and issue software triggers.
- Added a new RM to an already deployed design.

Lab 6

DFX Controller IP for UltraScale Devices

Overview

This lab demonstrates the AMD Dynamic Function eXchange (DFX) Controller IP for AMD UltraScale™ devices. As in the 7 series flow, you customize the DFX Controller by defining virtual sockets and their reconfigurable modules (RMs), then compile the design and generate a PROM file. You program the linear (BPI) flash on the board, operate the sample design, and use push buttons to issue hardware triggers and the AXI4-Lite interface to check core status and issue software triggers, including adding a new RM to an already deployed design. This lab targets the VCU108 development board. The KCU105 is not supported, because its QSPI boot flash is not a supported configuration scheme for DFX on UltraScale devices.

Step 1: Extract the Tutorial Design Files

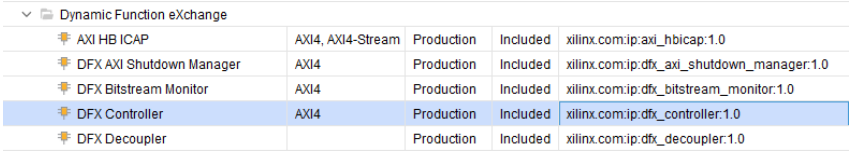
1. Download the reference design files.
2. Extract the zip file contents to any write-accessible location.
3. In the extracted files, navigate to `\dfxc_us`.

Step 2: Customizing the Dynamic Function eXchange (DFX) Controller IP

The DFX Controller IP requires a few details to be entered during the customization process. Identifying all information regarding each reconfigurable partition (RP) and reconfigurable module (RM) creates a fully populated controller that understands the entirety of the reconfiguration needs of the target FPGA. Within this IP, reconfigurable portions of the design are referred to as Virtual Sockets, which encompasses the RP along with all associated static logic used to manage it, such as decoupling or handshaking logic. While the core parameters are customizable during operation, the more that can be entered during this step, the better. This allows the front-end design description to more accurately match the final implemented design.

1. Open the AMD Vivado™ IDE and in the Tasks section click the **Manage IP** task, then**New IP Location**. Enter the following details and click **Finish**:
   - Part: Click the ellipses (...) to get to the part selector. Click **Boards** to select the VCU108
   - IP Location: <Extract_Dir>/Sources/ip

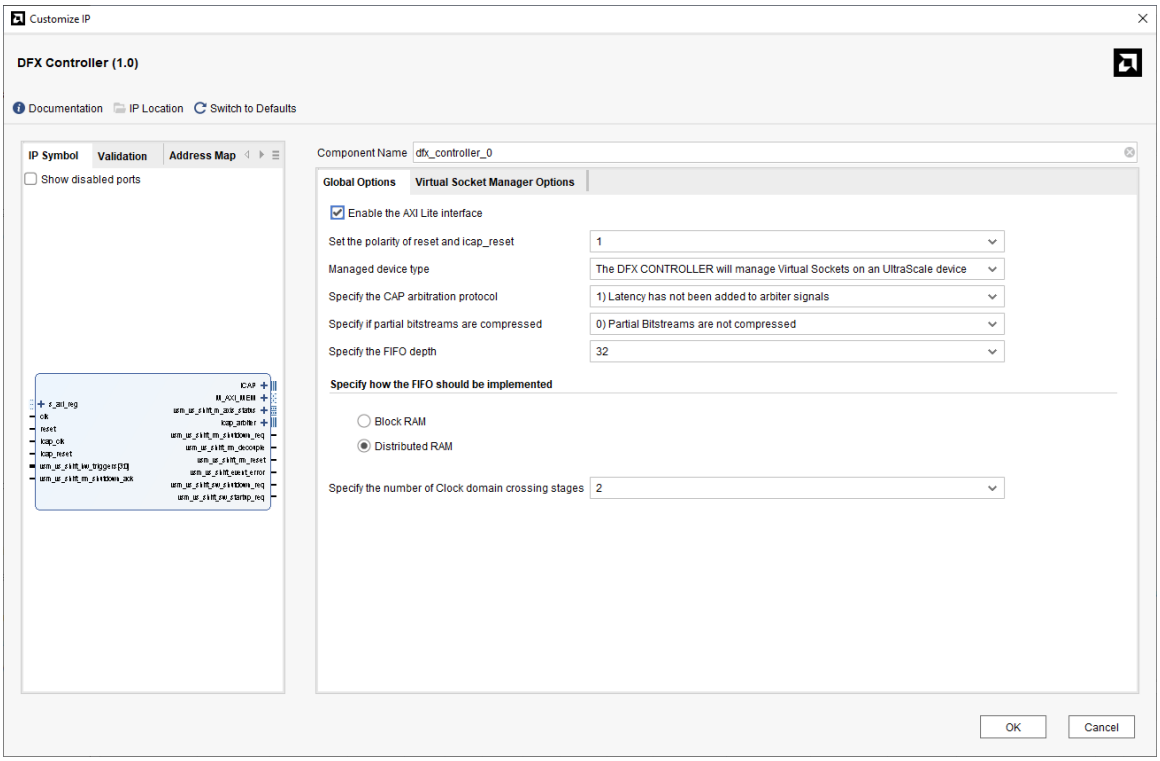
   ***Note*:** The KCU105 development board is not supported, as the boot flash on this board is a QSPI device. QSPI and sync-mode BPI configuration schemes are not supported for Dynamic Function eXchange on AMD UltraScale™ devices. See this link in *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909).
2. In the IP catalog, expand the Foundational Elements category, then expand DFX to double-click the DFX Controller IP.



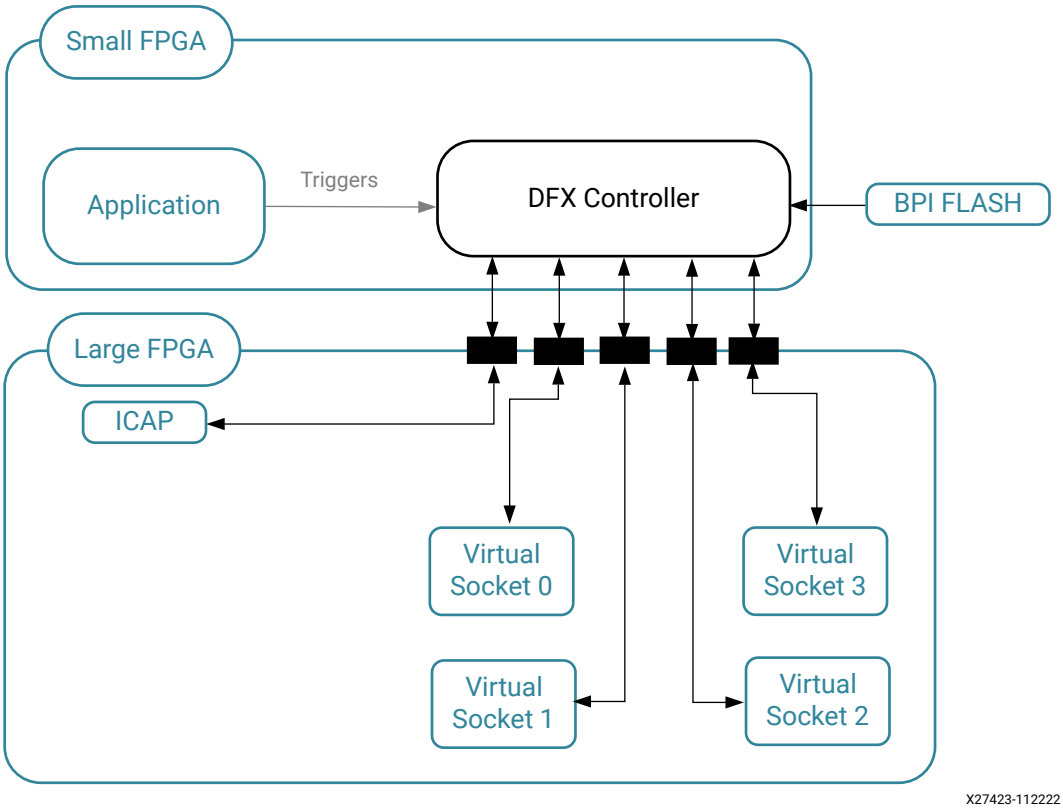
   The DFX Controller IP GUI has four tabs on the left side, providing feedback on the current configuration of the IP. The Validation tab shows any errors that might arise as the core parameters are entered. The core does not compile if errors exist.

   There are two tabs on the right side of the GUI where all customization is done. Most of the information is entered on the Virtual Socket Manager Options tab.
3. Leave the component name as dfx_controller_0. The version of the DFX Controller used in the final design will be automatically compiled in Step 3: Compiling the Design.
4. On the Global Options tab, make three changes:
   a. Set the Polarity of reset and icap_reset = **1**
   b. Specify the CAP arbitration protocol = **1) Latency has not been added to arbiter signals**
   c. Set the number of Clock domain crossing stages = **2**

Make sure that the Managed device type is set to The DFX CONTROLLER will manage Virtual Sockets on an UltraScale device. The DFX Controller GUI should now look like this:



***Note*:** This DFX Controller can manage Virtual Sockets on 7 series, UltraScale, or UltraScale+ devices. This IP is not limited to managing reconfiguration on the same device on which it resides. It can connect to an ICAP on another device to manage its reconfiguration. An example of a multi-chip solution using the DFX Controller follows.



5. Switch to the Virtual Socket Manager Options tab to define information about the Virtual Sockets and their Reconfigurable Modules.

   The DFX Controller IP is preloaded with one Virtual Socket with one Reconfigurable Module to get you started.

   First, define the Virtual Socket Manager (VSM) for the Shift functionality.

6. Rename the current VSM from **VS_0** to **vs_shift**.
7. Rename the current RM from **RM_0** to **rm_shift_left**.

CAUTION!

- To accept a new value in any field in the GUI, simply click in any other field in the GUI or press the Tab key. Do NOT press Enter, as this will trigger compilation of the IP.
- Underscores are not visible in the Virtual Socket Manager to configure and Reconfigurable Module to configure pull-down menus. The Virtual Socket Name and Reconfigurable Module Name fields below the pull-down menus show this more accurately.

8. Click the **New Reconfigurable Module** button to create a new RM for this VSM. Name it **rm_shift_right** in the **Reconfigurable Module Name** field.

TIP: *Up to 128 Reconfigurable Modules can be managed by a single Virtual Socket Manager.*

9. Configure the vs_shift VSM to have the following properties:
   - Has Status Channel = checked
   - Has PoR RM = checked, value set to **rm_shift_right**
   - Number of RMs allocated = **4**

   The PoR RM indicates which RM is contained within the initial full-design configuration file, so the VSM knows which triggers and events are appropriate upon startup of the FPGA. The VSM tracks the current active Reconfigurable Module in its socket.

   Even through you have only defined two RMs for this Virtual Socket, you have set aside space for four in total. This allows for expansion later on. Additional Reconfigurable Modules can be identified using the AXI4-Lite interface, but only if spaces have been reserved for them.
10. For each of these RMs, enter the following values. Use the **Reconfigurable Module to configure** pull-down to switch between the two RMs.
    - For rm_shift_left:
      - Reset type = **Active High**
      - Duration of Reset = **3**
    - For rm_shift_right:
      - Reset type = **Active High**
      - Duration of Reset = **10**

    ***Note*:** The different reset durations are given to show that these can be independently assigned, as each RM may have different requirements. Reset durations are measured in clock cycles.
11. For each RM, assign a bitstream size and location to identify where it resides in the BPI flash device.
    - For rm_shift_left:
      - Bitstream 0 address = **0x00B00000**
      - Bitstream 0 size (bytes) = **375996**
      - Bitstream 0 is a clearing bitstream = unchecked
      - Bitstream 1 address = **0x00B5C000**
      - Bitstream 1 size (bytes) = **26036**
      - Bitstream 1 is a clearing bitstream = checked
    - For rm_shift_right:
      - Bitstream 0 address = **0x00B62800**
      - Bitstream 0 size (bytes) = **375996**

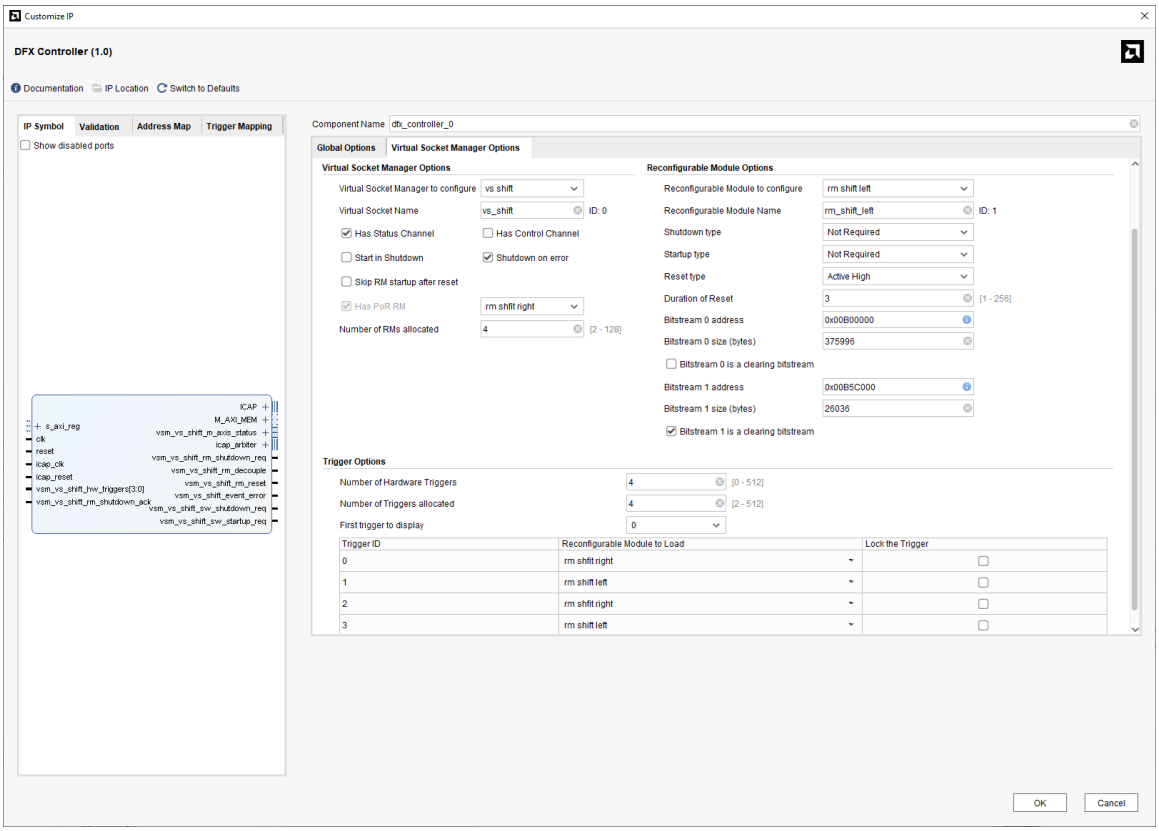
- Bitstream 0 is a clearing bitstream = unchecked
- Bitstream 1 address = **0x00BBE800**
- Bitstream 1 size (bytes) = **26036**
- Bitstream 1 is a clearing bitstream = checked

This information is typically not known early in design cycles, as bitstream size is based on the size and composition of the Reconfigurable Partition Pblock, and bitstream address is based on storage details. Until the design is to be tested on silicon, these can be set to 0. As the design settles and hardware testing with the DFX Controller is set to begin, this information can be added. The bitstream address information must match the information passed during PROM file generation. Certain bitstream generation options, most notably bitstream compression, can lead to variations in the final bitstream size for different configurations, even for the same Reconfigurable Partition.

12. Define the Trigger Options for the Shift functionality:
    - Number of Hardware Triggers = **4**
    - Number of Triggers allocated = **4**

    The four trigger assignments are done automatically. These can be modified during device operation using AXI4-Lite, which is especially useful when you have added a new RM through the same mechanism during a field system upgrade.

    At this point, the IP GUI should look like this (showing rm_shift_left here):



Next, create and populate the Count Virtual Socket following the same basic steps, with slightly different options here and there.

13. Click the **New Virtual Socket Manager** button to create a new VSM.
14. Add two RMs with these names and properties:
    - Reconfigurable Module Name = **rm_count_up**
        - Reset type = **Active High**
        - Duration of Reset = **12**
    - Reconfigurable Module Name = **rm_count_down**
        - Reset type = **Active-High**
        - Duration of Reset = **16**

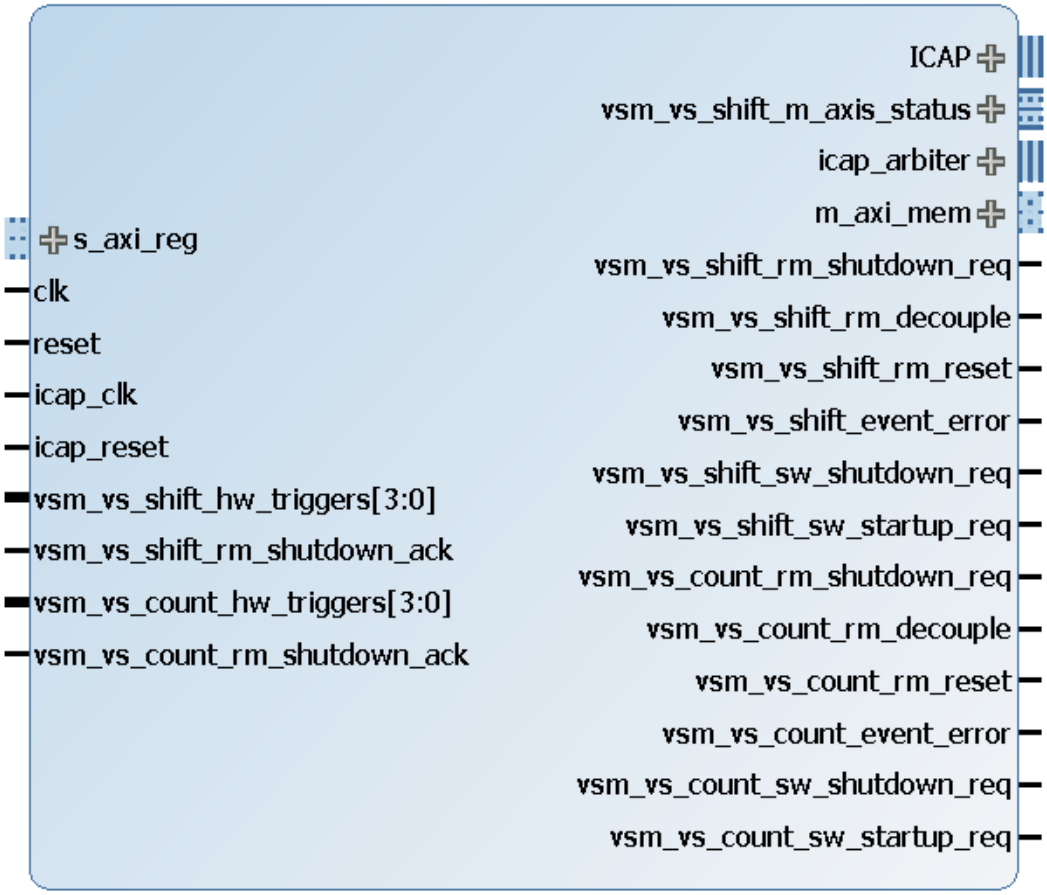
    For this Virtual Socket, leave the bitstream address and size information at the default of 0, but set bitstream 1 to be a clearing bitstream. In addition to being defined here, bitstream size information can be added to a routed configuration checkpoint via the DFX Controller Tcl API, or can be added in an active design using the AXI4-Lite interface. For the Count Virtual Socket, the bitstream address and size information is added using the Tcl commands after place and route, but before bitstream generation.

> **TIP:** *For more information about how to use the DFX Controller Tcl API, see the Customizing the Core Post Implementation topic in the Dynamic Function eXchange Controller IP LogiCORE IP Product Guide (PG374).*

Examine the Tcl scripts in the `<Extract_Dir>/Sources/scripts` directory. The `update_dfxc_vcu108.tcl` uses the DFX Controller Tcl API to update the bitstream address and bitstream size, which is stored in `dfx_info_vcu108.tcl`. This file is sourced later in the lab from `<Extract_Dir>/design.tcl`.

15. On the Virtual Socket Manager Options tab, modify these VSM settings from their default values:
    - Virtual Socket Name = **vs_count**
    - Start in Shutdown = checked
    - Shutdown on error = unchecked
    - Has PoR RM = checked, value set to **rm_count_up**
16. Define the Trigger Options for the Count functionality:
    - Number of Hardware Triggers = 4
    - Number of Triggers allocated = 4

    This completes the planned customization of the DFX Controller IP for this tutorial.
17. Click **OK** and then **Generate** to begin core compilation and out-of-context synthesis.



Step 3: Compiling the Design

The DFX Controller IP is created, but the design is not yet compiled. In order to create the PROM image with all the necessary full and partial images, source the following scripts in Tcl mode using the commands below.

```
vivado -mode tcl -source design.tcl
```

Sourcing `design.tcl` generates all the necessary IP (including the DFX Controller), synthesizes and implements the entire design (three configurations), updates the vs_count VSM using the DFX Controller Tcl API, and generates bitstreams.

Note: The customization of the IP is scripted. Examine the gen_ip_vcu108.tcl script (in <Extract_Dir>/Sources/scripts) to see all these parameters defined for automated IP creation, the DFX Controller, and others. The DFX Controller instance you create using the IP GUI is not actually used for the full design processing, so you do not have to complete Step 2 to compile the entire design.

```
vivado -mode tcl -source create_prom_file_vcu108.tcl
```

Sourcing `create_prom_file_vcu108.tcl` creates the PROM image for the VCU108 target. This script contains hard-coded values for bitstream address for the entire project. If this design is modified in such a way that changes bitstream sizes, full or partial, then these values must also change. Changes that affect bitstream sizes include changing the target device, changing the size or shape of the Pblocks, or introducing bitstream options such as compression or per-frame CRC.

This script defines PROM file options by setting properties and then making calls to write_cfgmem. The DFX Controller works in byte addresses because the data is stored in bytes in AXI. This linear flash PROM uses half word addresses because it stores data in half words (16 bits). Divide the ROM address by 2 to get the AXI address. For example, the shift_left address is given as 00B00000 during DFX Controller customization and 00580000 (half that value) for write_cfgmem. Note that the starting addresses are always multiples of 1024 (0x0400) to ensure that each bitstream starts on a byte address boundary.

Supplied in the lab directory is a file called `dfxc_bitstream_sizes_lab6.xlsx`. In this file, bitstream sizes are entered by the user based in the yellow highlighted fields. It calculates the starting address in hex for each partial bitstream at the next byte boundary. Values in blue are to be supplied for DFX Controller IP customization, in either the DFX Controller IP GUI, in the `gen_ip_vcu108.tcl` script, or in `dfx_info_vcu108.tcl` which is used for post-route API modification. The values in green are addresses divided by two to be used in PROM file generation in the `create_prom_file_vcu108.tcl` script.

Step 4: Setting up the Board

Once the partially reconfigurable design is in operation, you can connect to and communicate with the core to check status, deliver triggers, and make modifications.

1. Prepare the VCU108 board for programming.
   a. Connect the JTAG port (J106) to your computer via the micro-USB connection.
   b. Set the configuration mode to 010 (BPI) by setting the Address DIP Switch (SW16) to 00010 (bit 4 is high).
   c. Turn on the power to the board.
2. Open the AMD Vivado™ IDE.
3. Select **Flow → Open Hardware Manager**.

4. Click on the **Open Target** link and select **Auto Connect**. The AMD Virtex™ UltraScale™ VU095 device will be recognized.
5. To program the BPI configuration flash, right-click the device (xcvu095_0) and select **Add Configuration Memory Device**.
6. From the list shown, select the Micron flash **28f00ag18f** and click **OK** twice.
7. In the Configuration file field, search the tutorial directory for dfx_prom.mcs found in the bitstreams subdirectory. Click **OK** to select this file, and then click **OK** to program the flash.

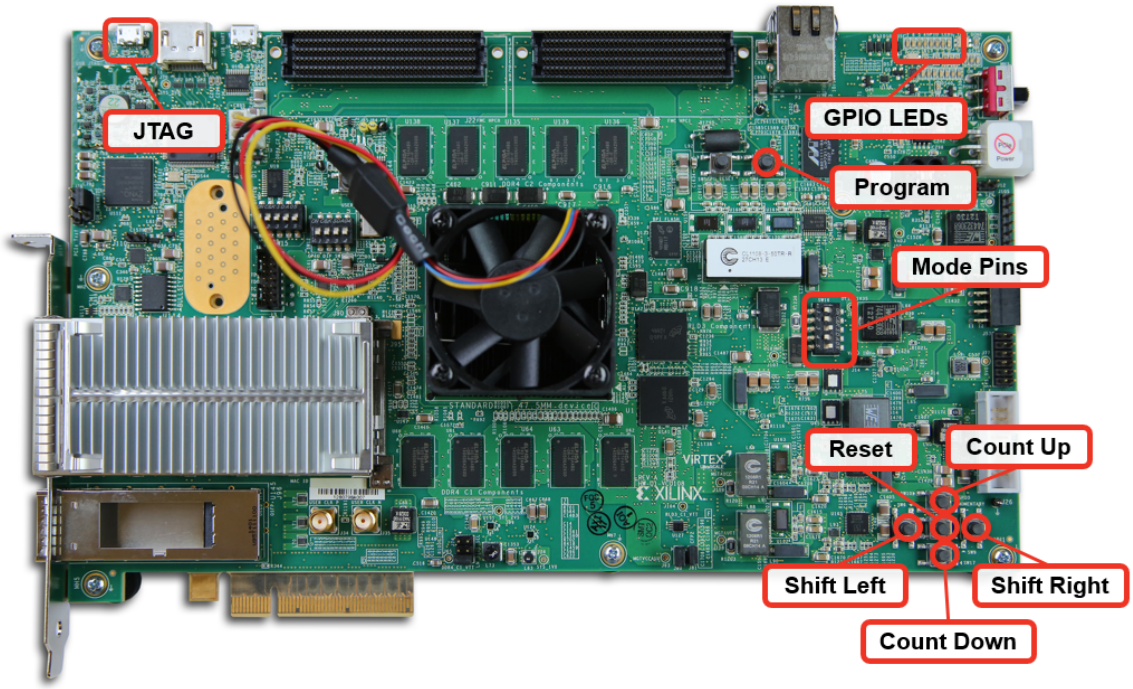
   At this point, the board is ready to operate with the tutorial design. Any power-cycle or hard reset automatically programs the Virtex UltraScale FPGA with this sample design.

Step 5: Operating the Sample Design

Position the board so that the text is readable. The LCD screen is on the side closest to you, with the power connection on the right and the JTAG connection on the left. The buttons of interest are the five user push buttons in the lower right corner, plus the PROG push button in the middle right. Their functions are as follows:

- PROG (SW4) – program the device from the BPI flash
- North (SW10) – load the Count Up partial bit file
- South (SW8) – load the Count Down partial bit file
- East (SW9) – load the Shift Right partial bit file
- West (SW6) – load the Shift Left partial bit file
- Center (SW7) – reset the design

Figure 5: **Push Buttons, Switches, and Connections on the VCU108 Demonstration Board**



1. Program the FPGA by pressing the PROG pushbutton. The 8 GPIO LEDs in the upper-right corner will start operation after the DONE LED goes high.

   At this point, the four bits on the left of the GPIO bank are counting up, and the four bits on the right are shifting to the right.

2. Press the Shift Left and Shift Right buttons alternately.

   With each push, a partial bit file is pulled from the BPI flash by the DFX Controller and delivered to the ICAP, changing the functionality in that Reconfigurable Partition. When this happens, the LED shift direction changes, depending on the button pushed.

3. Press the Count Down and Count Up buttons alternately.

   With each push, nothing happens. When configuring the DFX Controller, the Counter Virtual Socket was programmed to begin in Shutdown mode. It does not respond to any hardware or software triggers until it is moved to Active mode.

Step 6: Querying the DFX Controller in the FPGA

In this step, you interact with the core via JTAG from the Hardware Manager to understand the status of the core and issue software triggers.

In the Vivado Hardware Manager, you might need to select **Refresh Device** to establish the link to the device over the JTAG connection. Notice the XADC as well as 6 ILA cores and the hw_axi link shown under the device in the Hardware view.

1. In the Tcl Console, cd into the DFX Controller tutorial directory then source the AXI4-Lite command Tcl script.

   ```
   source ./Sources/scripts/axi_lite_procs_us.tcl
   ```

   This enables a set of procedures that make the subsequent interaction with the DFX Controller easier. Examine this file to see how these procedures are defined. These are written explicitly (hard-coded) for this design, the references to Virtual Sockets in any other design will need to be modified. For more information on this topic, consult the *Dynamic Function eXchange Controller IP LogiCORE IP Product Guide* (PG374).

2. Source the procedure to establish communication with the DFX Controller.

   ```
   dfxc_jtag_setup
   ```

3. Check the state of each Virtual Socket to see if they are in Shutdown or not.

   ```
   is_vsm_in_shutdown vs_shift
   is_vsm_in_shutdown vs_count
   ```

   You should see that the Shift Virtual Socket is in Active mode (value = 0), and the Count Virtual Socket is in Shutdown mode (value = 1).

4. Examine the status of each Virtual Socket.

   ```
   dfxc_decode_status vs_shift
   dfxc_decode_status vs_count
   ```

   Before examining the data returned, reference the table in the STATUS Register section of the *Dynamic Function eXchange Controller IP LogiCORE IP Product Guide* (PG374). The table in that section defines the values in the STATUS register. While this is a 32-bit register, you only need to pay attention to the lowest 24 bits, as the upper 8 bits are used for Virtual Socket Managers (VSM) in UltraScale devices.

   The status of vs_shift is 263, which is 0000_0000_0000_0001_0000_0111 in binary. The status for vs_shift can also be 7, where the only difference is that RM_ID is now 0.

   - RM_ID (bits 23:8) = 1. This means RM 1 is loaded (rm_shift_right). It can also appear as RM_ID (bits 24:8) = 0. This means RM 0 is loaded (rm_shift_left).

- SHUTDOWN (bit 7) = 0. This VSM is not in the shutdown state.
- ERROR (bits 6:3) = 0000. There are no errors.
- STATE (bits 2:0) = 111. The Virtual Socket is full.

The status of vs_count is 129, which is 0000_0000_0000_0000_1000_0001 in binary.

- RM_ID (bits 23:8) = 0. This means RM 0 is loaded (rm_count_up).
- SHUTDOWN (bit 7) = 1. This VSM is in the shutdown state.
- ERROR (bits 6:3) = 0000. There are no errors.
- STATE (bits 2:0) = 001. RM_SHUTDOWN_ACK is 1, as this VSM is executing the hardware shutdown step.

These explicit details are reported in the breakdown of the status register in the return value from this Tcl proc.

5. Send a software trigger to the Shift Virtual Socket.

```
dfxc_send_sw_trigger vs_shift 0
dfxc_send_sw_trigger vs_shift 1
```

Remember that values of 0 and 2 correspond to shift left, and values of 1 and 3 correspond to shift right, as defined during DFX Controller customization.

6. Check the configurations of the RMs for the Count Virtual Socket.

```
dfxc_show_rm_configuration vs_count 0
dfxc_show_rm_configuration vs_count 1
```

The values for the bitstream sizes and address for both clearing and partial bitstreams are reported here. These values could then be modified to account for necessary adjustments to the size or location of the bitstream. Different indices can be added to insert new RMs. Note that this query cannot be done for vs_shift, as the vs_shift VSM is not in the shutdown state.

7. Move the Count Virtual Socket Manager into active mode.

```
dfxc_restart_vsm_no_status vs_count
```

The Count Up and Count Down pushbuttons can now be used to load these partial bitstreams using the DFX Controller.

Step 7: Modifying the DFX Controller in the FPGA

In the final step, you add a new Reconfigurable Module to the Shifter VSM. In the `create_prom.tcl` script, you can see that two black box modules have already been generated. These represent two new RMs that may have been created after the static design was deployed to the field. You modify the DFX Controller settings to access one of these RMs by assigning the size, address, properties and trigger conditions.

1. Shut down the Shift VSM so it can be modified.

   ```
   dfxc_shutdown_vsm vs_shift
   ```

2. Check the status of the first three RM IDs to see their register bank assignments.

   ```
   dfxc_show_rm_configuration vs_shift 0
   dfxc_show_rm_configuration vs_shift 1
   dfxc_show_rm_configuration vs_shift 2
   ```

   Currently, RM ID 2 is not assigned to any partial bitstreams. This is the behavior as requested when the initial trigger mapping was done during core customization.

3. When the MCS file is created for the prom, it adds additional blanking RMs that are already loaded into the BPI flash. Use this sequence of commands to reassign the trigger mapping for slot 2 to point to the blanking Reconfigurable Module for vs_shift.

   ```
   dfxc_write_register vs_shift_rm_control2 0
   ```

   This defines the settings for the RM_CONTROL register for slot 2. No shutdown, startup, or reset are required. Note how for the other two slots, the differing reset durations lead to different control values.

   ```
   dfxc_write_register vs_shift_rm_bs_index2 327684
   ```

   This assigns a new bitstream reference for this RM ID.

   ```
   dfxc_write_register vs_shift_trigger2 2
   ```

   This assigns the trigger mapping such that trigger index 2 retrieves RM 2. The RM_BS_INDEX register within the DFX Controller is 32 bits but is broken into two fields. UltraScale devices require clearing and partial bitstreams. These bitstreams are identified separately with unique IDs, but referenced together in this field.

   This value of 327684 converts to `0000000000000101_0000000000000100` in binary. Or more simply, ID 5 for the upper 16 bits for the CLEAR_BS_INDEX and ID 4 for the lower 16 bits for the BS_INDEX. This assignment sets the clearing and partial bitstream identifiers at the same time.

   ```
   dfxc_show_rm_configuration vs_shift 2
   ```

This shows the current state of RM ID 2. Note the changes from the prior call to this command.

4. Complete the RM ID 2 customization by setting the bitstream details.

```
dfxc_write_register vs_shift_bs_size4 375996
dfxc_write_register vs_shift_bs_address4 12935168
dfxc_write_register vs_shift_bs_size5 26036
dfxc_write_register vs_shift_bs_address5 13312000
```

5. Restart the VSM and then issue trigger events to it using software, as there is no pushbutton assigned for slot 2.

```
dfxc_restart_vsm_no_status vs_shift
dfxc_send_sw_trigger vs_shift 2
```

Switch between values of 0,1, and 2 to reload different partial bitstreams. The blanking bitstream in slot 2 removes the shifter function, so no activity on the LEDs is seen.

Note that this same sequence of events could not be performed for the Count VSM as it is currently configured, even knowing that the PROM image has a Count black box partial bitstream sitting at address 13338624 with a size of 274104. During DFX Controller customization, this VSM was selected to have only 2 RMs allocated, so expansion is not permitted.

Lab 6 Conclusion

This concludes lab 6. In this lab, you:

- Customized the Dynamic Function eXchange (DFX) Controller IP.
- Created Virtual Sockets and added RMs to them.
- Compiled the design and created a PROM file.
- Programmed the linear flash on the VCU108 board.
- Used pushbuttons to issue hardware triggers.
- Used the AXI4-Lite interface to check the core status and issue software triggers.
- Added a new RM to an already deployed design.

Lab 7

DFX Controller IP for UltraScale+ Devices

Overview

This lab focuses on runtime functionality and software management of Dynamic Function eXchange (DFX) on AMD UltraScale+™ devices, rather than on the implementation flow covered in earlier labs. Using a scripted MicroBlaze™-based design, you process and export the hardware, then build software in the AMD Vitis™ environment to manage reconfiguration. You program the QSPI flash, drive DFX events over a UART interface from QSPI or DDR4 memory, monitor reconfiguration through debug cores, and insert bitstream delivery errors to observe the CRC-checking response. This lab targets the VCU118 development board.

Step 1: Extract the Tutorial Design Files

1. Download the reference design files.
2. Extract the zip file contents to any write-accessible location.
3. In the extracted files, navigate to `\dfxc_usp\dfxc_vcu118`.

Step 2: Processing the Tutorial Design

The focus of this design is end runtime functionality and software management, not design processing. This section does not cover implementation flow details extensively. Review the Dynamic Function eXchange (DFX) design flow in earlier labs of this document.

1. Launch AMD Vivado™ from the command shell in Tcl mode to source the project creation script.

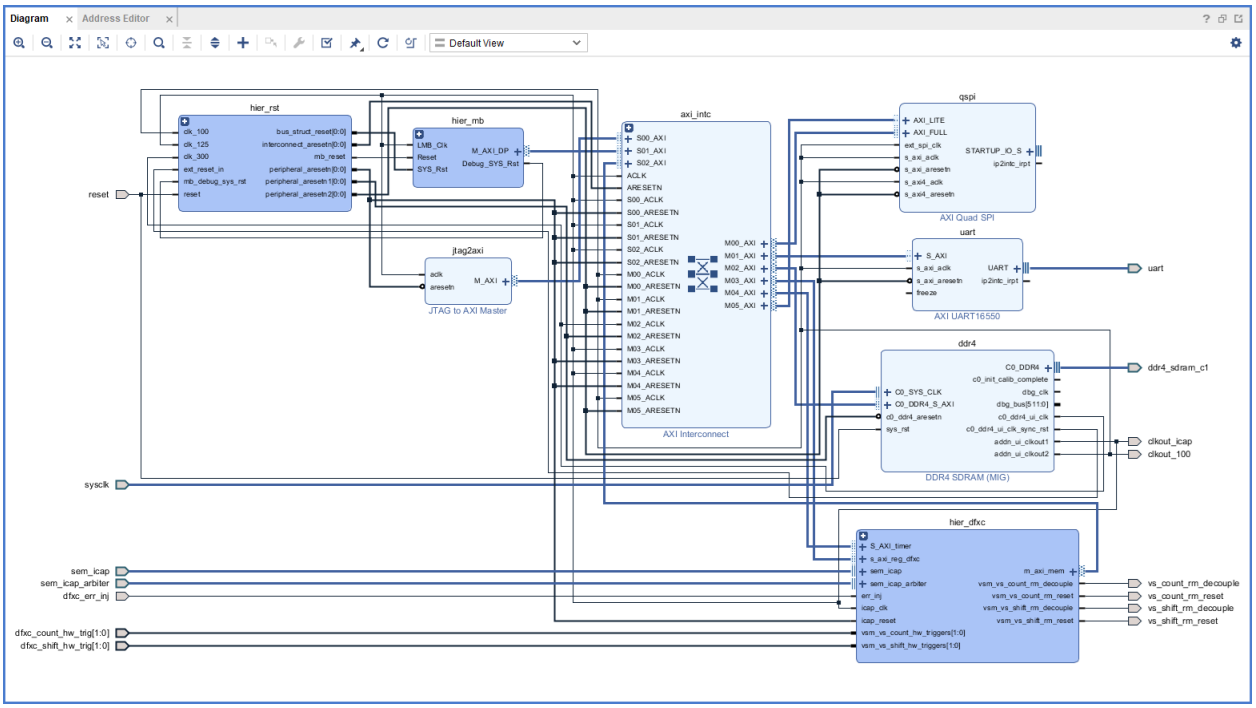
```
vivado -mode tcl -source project_dfxc_vcu118.tcl
```

2. When the script completes, open the Vivado IDE:

```
start_gui
```

3. Check if the IP needs to be updated:
   a. Select **Reports → Report IP Status** and update any out-of-date IP.
   b. If you use a newer version of Vivado, expect minor changes such as to the ILA core. Align your Vivado version with that of this document for consistent results. When updating IP, keep the default setting with the core container disabled. Skip synthesis of the IP modules; that is handled in the next step.
4. Generate the block design:
   a. In the Flow Navigator, select **IP INTEGRATOR → Generate Block Design** to prepare the design.
   b. Keep the Out of context per IP Synthesis option selected and click **Generate**.

   Vivado creates all IP modules in the block design and synthesizes them. This static infrastructure does not trigger synthesis for RTL submodules such as the shift and count reconfigurable modules (RMs).
5. In the Flow Navigator, click **Run Implementation** to process the design through synthesis and implementation.
6. Review the design during place and route. While place and route runs, explore the design:
   - Use the LED-Shift-Count top-level design from other labs, with a block diagram (`mb_dfxc`) inside an AXI subsystem.
   - The MicroBlaze™ manages the design and interacts with a user interface through UART.
   - The DFX Controller IP triggers reconfiguration events, supported by an ILA core and a timer for visibility.
   - Include DDR4 and QSPI interfaces.
   - Instantiate shift and count RMs, the SEM IP, and multiple FRAME_ECC components for SSI support.



7. Export the hardware:
   a. When implementation finishes, skip bitstream generation. Click **Cancel** in the dialog.
   b. Select **File → Export → Export Hardware**, then click **Next**.
   c. Keep Output set to Pre-synthesis, then click **Next**.
   d. Keep the default *top* name of the XSA file.
   e. Click **Next** and then **Finish** to create a design image for the AMD Vitis™ software platform.
8. Before you launch the Vitis Unified tools, complete extra steps for this tutorial. This design uses a PRC driver with a known issue. Apply the provided Vitis SDT patch.
   a. Back in the command shell, navigate to the Vivado project directory, dfxc_vcu118/project_dfxc_vcu118. Clone the device tree Git repository:

```
git clone https://github.com/Xilinx/system-device-tree-xlnx
```

   b. Navigate to the repository and check out the 2025.2 repository:

```
cd system-device-tree-xlnx
git checkout xlnx_rel_v2025.2
```

   c. Apply patch delivered in the design archive:

```
git apply ../../sources/dfxc_demo/sdtgen_patches/0001_fix_prt_driver.patch
```

9. With the patch applied, launch Vitis.
   a. In the Vivado IDE, select **Tools → Launch Vitis IDE**.

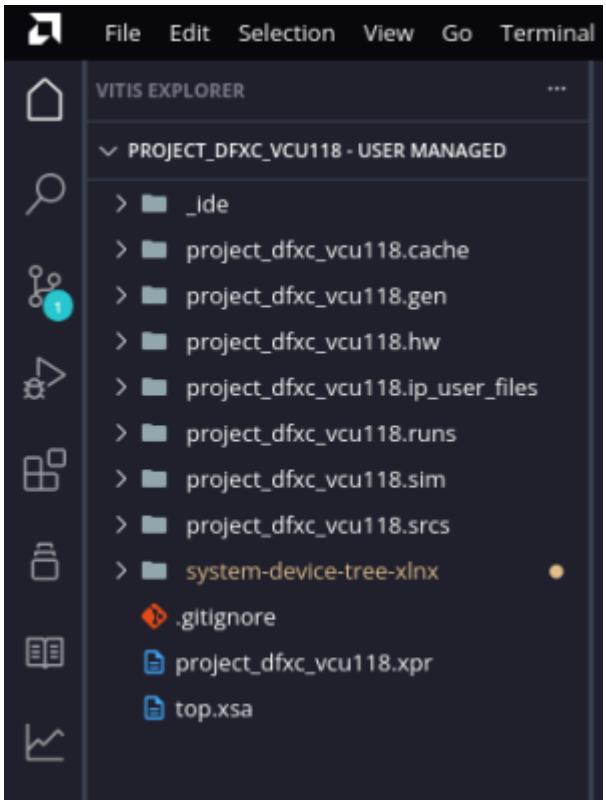
b. Or, from a terminal type `vitis`.

10. Set and verify the workspace in Vitis:

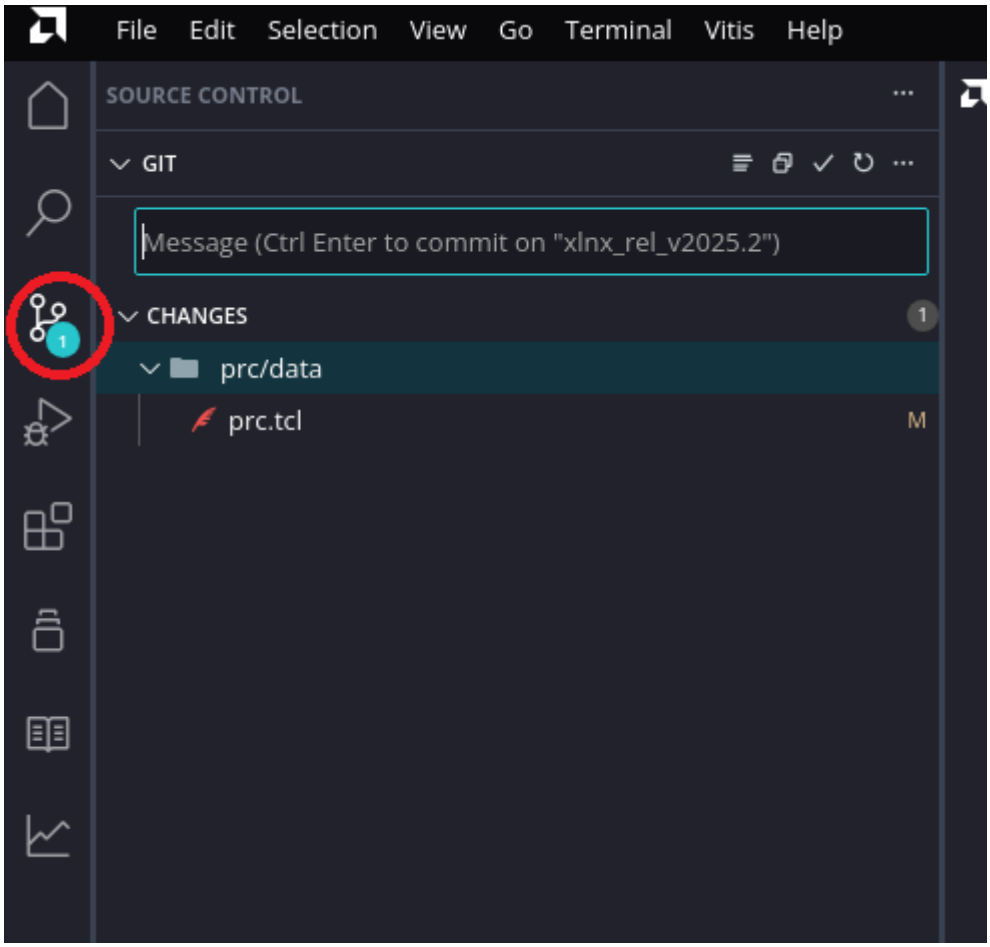
    a. In the Vitis explorer, set the workspace to the `project_dfxc_vcu118` project directory.

       - If prompted about an unrecognized workspace version, select **Update**.

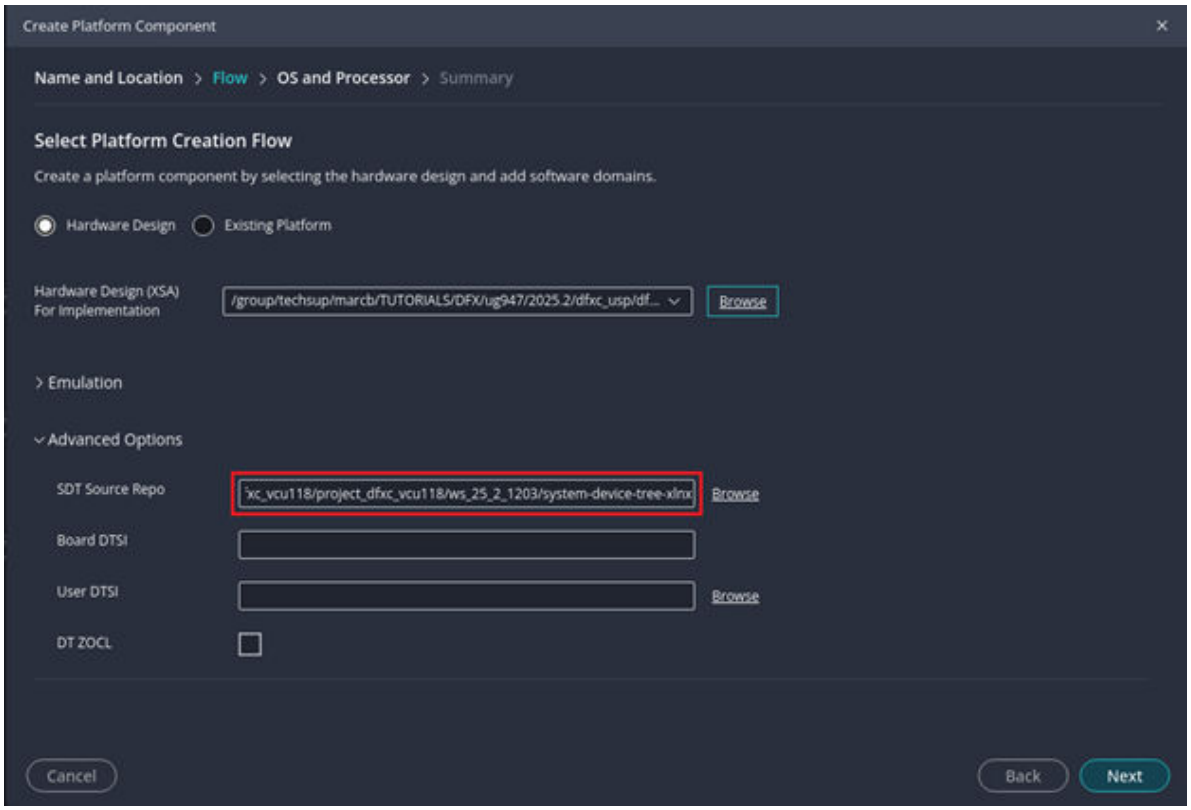
    b. Verify that the workspace maps to the `system-device-tree-xlnx` repository and confirm that the patch is in place. Look for a dot alongside `system-device-tree-xlnx` indicating changes.



    c. Click **Source control** (with the teal circle). Check that `prc.tcl` shows an orange M, indicating a modified source file.



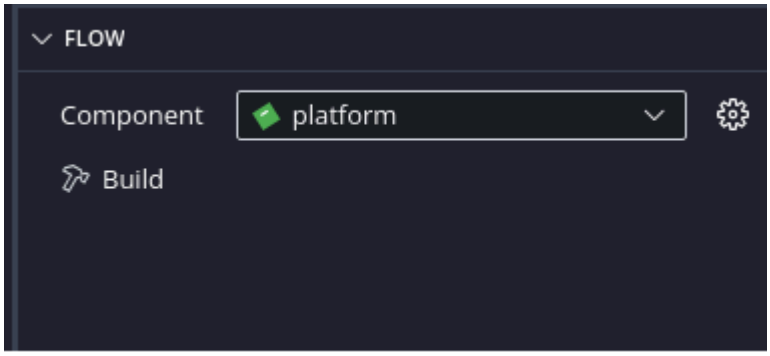
11. Create the platform:
    a. Select **File → New Component → Platform**.
    b. Leave the Component name as *platform*, and keep the Component location as is. Click **Next** to open the Flow tab.
    c. In Hardware Design (XSA) For Implementation, import the `top.xsa` file exported from Vivado (click **Browse**, select the file, then click **Open**.
    d. Open **Advanced Options** and set **SDT Source Repo** to the directory to where the cloned system device tree repository is located: `<export_dir>/dfxc_usp/dfxc_vcu118/project_dfxc_vcu118/system-device-tree-xlnx`, then click **Next**.



e. Confirm the Operating system is standalone and the processor is `hier_mb_mb`.

f. Click **Next** and then **Finish**.

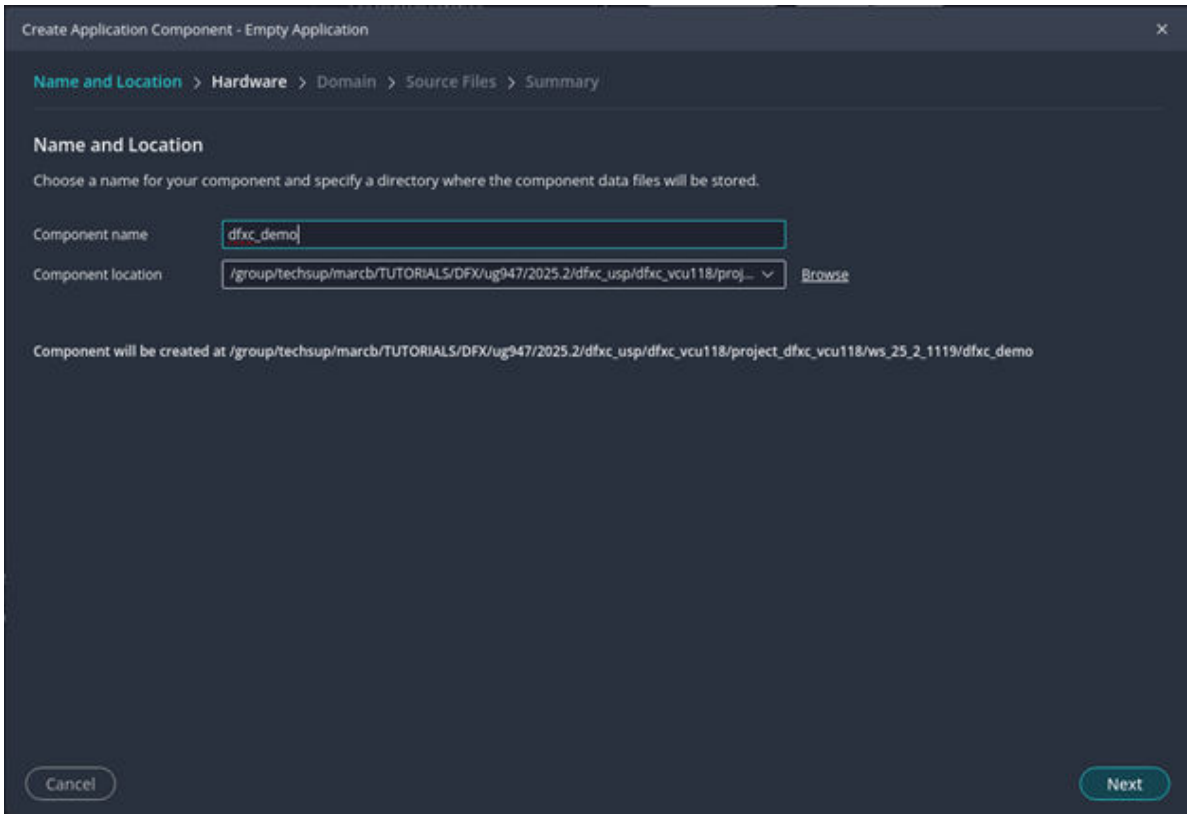
g. To build the platform, click **Build** (the button with the hammer icon) from the Vitis Explorer view.



12. Create the application:

a. Select **File → New Component → Application**.

b. In the Component name field, enter *dfxc_demo*, then click **Next**.



c. On the Hardware page, select the newly created platform (*platform*) from the list.

d. Click **Next** twice, skipping the Domain page.

13. Add the source files.

    a. On the Source Files page, click the **Add Files** icon.

    b. Browse to the `sources/dfxc_demo/src` directory.

    c. Select all six `.c` and `.h` source files in the folder, then click **Open**.

    d. Click **Next**, then **Finish** on the Summary page to complete application creation.

14. Build the application:

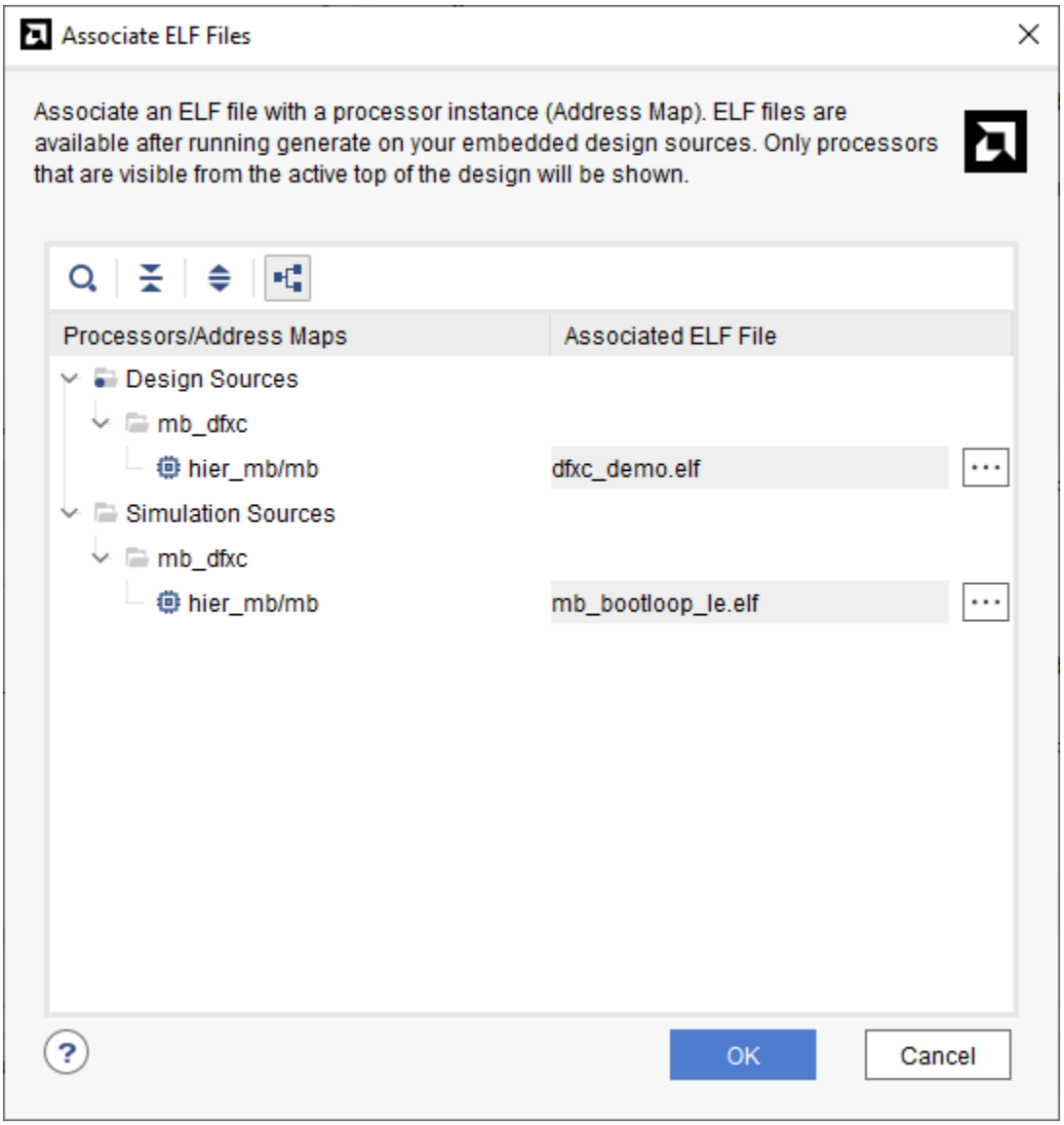
    a. In the Flow window, select the `dfxc_demo` application from the Component drop-down list.

    b. Click the **Build** to compile the application.



Note: If prompted to set the Application Platform Build Dependency, select **Always build platform with application** and click the **Save in Workspace Preference** button.

15. Review the source code:
    a. Expand the Sources directory and open `dfxc_demo.c`.
    b. Review the main software code that you see later.
    c. Find partial bitstream locations and sizes in `dfxc_demo.h`.
    d. See calculations in `dfxc_bitstream_sizes_lab7.xlsx`
16. Exit Vitis and return to Vivado.
17. Add and associate the ELF file:
    a. In the Vivado IDE project, select **File → Add Sources**.
    b. Select **Add or create design sources**, then click **Next**.
    c. Add `dfxc_demo.elf` from the `project_dfxc_vcu118/dfxc_demo/build` folder.
    d. Deselect the **Copy Sources into project** option, then click **Finish**.
    e. In the Sources window, right-click `dfxc_demo.elf` in the ELF section.
    f. Select **Associate ELF Files** then select **Skip Generate** because the design is compiled.
    g. In the window that appears, set the top reference for Design Sources to the `dfxc_demo.elf` file you added.
    h. Click **OK**.



18. Generate the full bitstream:
    a. In the Design Runs window, right-click `impl_1`
    b. Select **Generate Bitstream**, then click **OK**.

    This creates the full bitstream containing the MicroBlaze code in the .elf file for the Shift Right – Count Up configuration. Partial bitstreams for each RM are also generated. Use only the full configuration bitstream from the parent run here. Do not generate bitstreams from the `child_0_impl_1` run. Check the log for reported partial bitstream sizes in bits.
19. Create all bitstream files:
    a. In the Tcl Console, change to the directory above `project_dfxc_vcu118` where the script exists.

b. Source the script:

```
source create_all_bitstreams_vcu118.tcl
```

Settings for full versus partial bitstreams must be different to account for the configuration modes and options in this design. This is done within a Tcl script that copies the full bitstream created in the prior step, and then creates all the partial bitstreams that are necessary for all RMs.

> **IMPORTANT!** *You cannot currently set different options for full versus partial bitstreams in project mode in the Vivado IDE. A future Vivado release might add this feature as a new page in the DFX Wizard.*

Examining this script shows that the CONFIG_MODE must change from the default of SPIx4 for the initial configuration to SELECTMAP32 for the partial bitstreams, which are delivered to the ICAP.

20. Review the partial bitstream script:
    - Examine the script to confirm `CONFIG_MODE` changes from the default of SPIx4 for the initial configuration to SELECTMAP32 for the partial bitstreams. Partial bitstreams deliver to the ICAP.
    - The script generates two versions of partial bitstreams: with and without the per-frame CRC feature enabled.
21. Source this script to create PROM images for the target board:

```
source create_prom_file_vcu118.tcl
```

This creates a QSPI boot image containing the full bitstream followed by all partial bitstreams. The script calculates addresses based on the size of each partial bitstream. The log reports partial bitstream sizes in bytes. See the `dfxc_bitstream_sizes_lab7.xlsx` for calculation details.

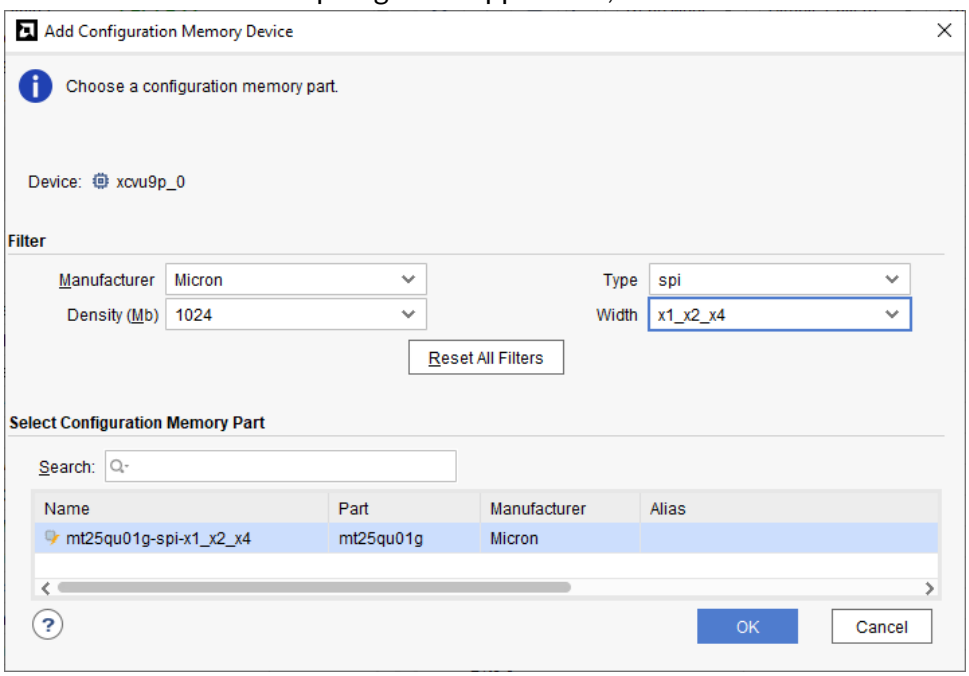
Step 3: Running the Tutorial Design

Once all the bitstreams and prom images have been created, you can run the design on hardware. There are many different features that can be demonstrated. There is no specific order in which these demonstrations must be done after the device has been programmed.

Program the QSPI Flash Device

1. Connect the target board to your computer and power the board on. Connect to both micro-USB ports for JTAG and UART connections.

2. Program the configuration memory device (QSPI).
   a. Open the Vivado Hardware Manager and connect to the target board.
   b. Right-click on the device and select **Add Configuration Memory Device**.
   c. Select the Micron mt25qu01g that supports x1, x2 and x4 modes.



   d. When prompted, add a programming file. The target file is the `dfx_prom.mcs` from the Bitstreams folder in the project directory.

3. After the PROM has been programmed, use the PROG button to reconfigure the FPGA from this boot flash.

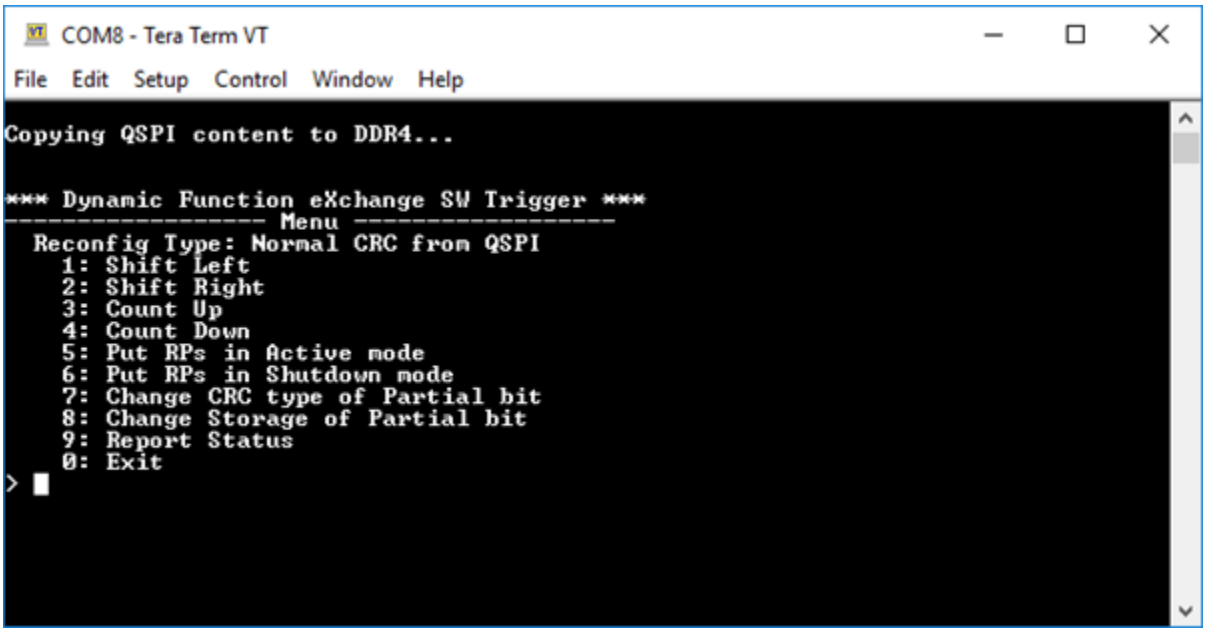
   The push buttons can control actions in the FPGA design. The left and right buttons load shift left and shift right partial bitstreams, respectively. The up and down buttons load count up and count down partials, respectively. Do not push the center button yet.

Manage Reconfiguration via Software

1. Open a UART terminal to communicate with the software running in MicroBlaze.
   a. Set the COM port to an appropriate value for your computer.
   b. Set the Baud Rate to 115200.
   c. Press the **PROG** button on the board to restart the design with the UART terminal open.

      Note: If you need a USB to UART driver for your terminal, see the *Silicon Labs CP210x USB-to-UART Installation Guide* (UG1033).

   All the partial bitstreams are copied from the QSPI flash over to the DDR4 memory. Then the software menu appears:



The menu options allow you to:

- **1-4:** These are the four trigger options to load partial bitstreams to the ICAP via the DFX Controller. These mimic the pushbuttons on the board.
- **5-6:** These toggle the DFX Controller status between active and shutdown mode for both RPs. When the RPs are in shutdown mode, triggers (software or hardware) are ignored.
- **7:** This toggles the partial bitstreams used between standard partials and those instrumented with per-frame CRC checks.
- **8:** This toggles the partial bitstream source between QSPI and DDR4 memory storage.
- **9:** Reports the current status of each Virtual Socket.

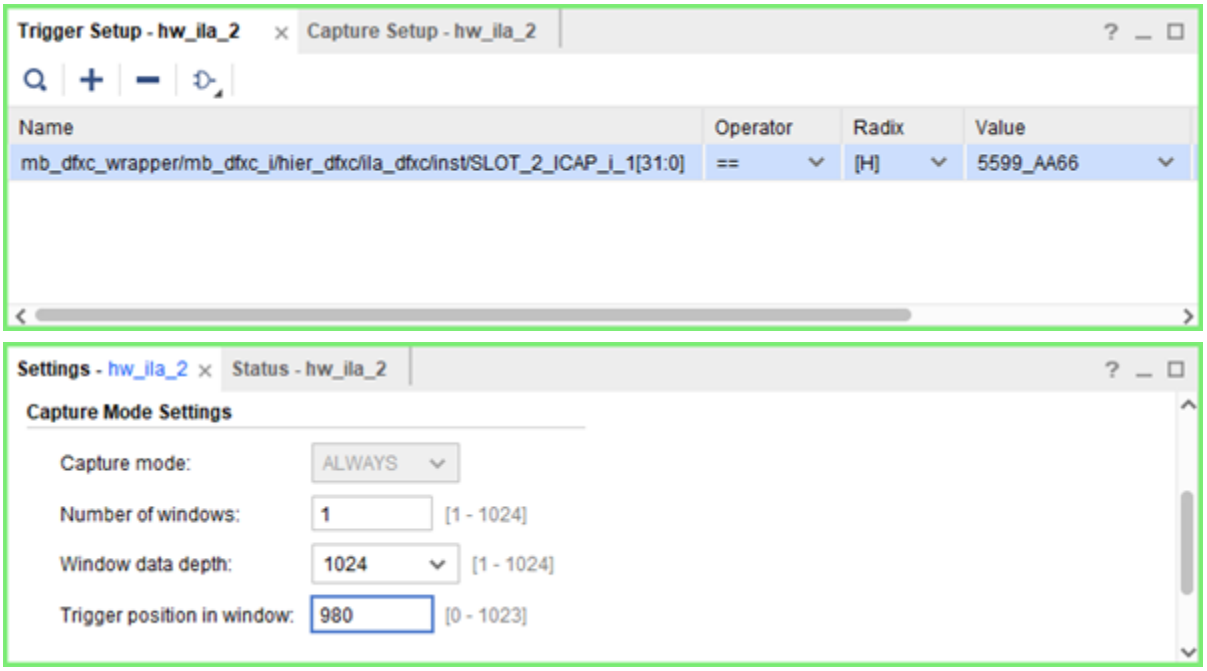
As you walk through these different features, the software provides feedback. Reconfiguration time is reported each time reconfiguration is executed from within the software.

Monitor Dynamic Function eXchange via Debug Cores

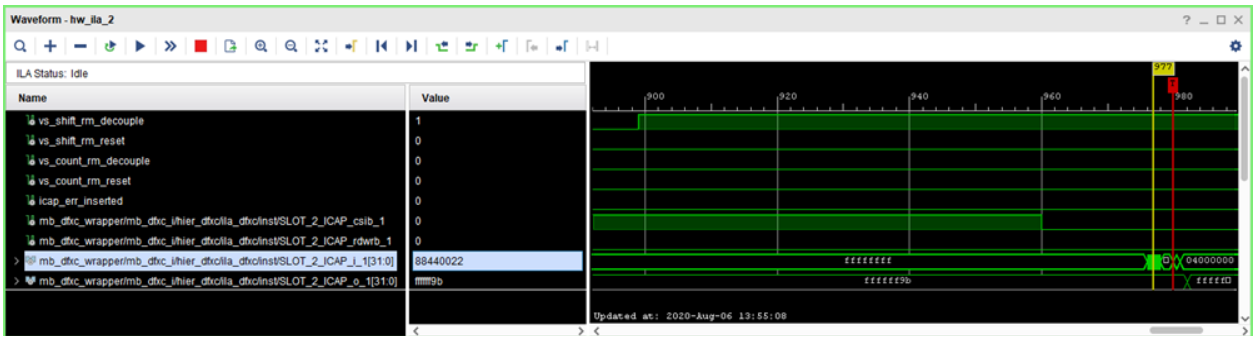
Vivado debug cores have been inserted in this design, allowing you to monitor activity during Dynamic Function eXchange events. A key detail for configuring via the ICAP is to have prepared the partial bitstreams with the right bit and byte ordering for each .bin file.

1. In the Vivado Hardware Manager, refresh the device to find all the Vivado Debug cores. There are three ILA cores, one VIO core and one MIG core in this design.
2. Right-click the part in Hardware Menu, and select **Hardware Device Properties**. In the General tab, point the probes file to `Bitstreams/top_count_up_shift_right.ltx`.

3. In hw_ila_2, click the + to add probes in the Trigger Setup window.
4. Select **SLOT_2_ICAP_i_1[31:0]** and click **OK**. Change the Radix to **[H]** for hexidecimal.
5. Set the trigger in the Trigger Settings window to a value of 5599_AA66, which is the configuration sync word, bit-swapped.
6. Set the trigger position in the Settings window to 980.



7. Select the Run Trigger button in the Hardware Manager GUI.
8. On the board, push one of the pushbuttons other than the center to trigger reconfiguration of the shifter or counter. Or, perform this action via the UART terminal.
9. In the resulting captured waveform, note a few things:
   - Far to the left, one of the `rm_decouple` signals (depending on which Reconfigurable Partition you have chosen to reconfigure) has transitioned from low to high. This isolation is initiated in the design prior to partial bitstream delivery.
   - The sync word is preceded by 000000dd and then 88440022, which are the bit-swapped bus width detection.
   - The ICAP output transitions from ffffff9b (no sync, no error) to ffffffdb (sync, no error). This transition shows recognition of the sync word, and the configuration engine is now expecting bitstream data.
   - PRDONE transitions high to low much further to the right, out of the range of this captured waveform.



***Note*:** Bitstreams will have multiple sync-desync pairs, as they are constructed via multiple segments. Multi-SLR devices, for example, have more due to bitstream formatting per SLR.

10. Change the Value of the ICAP_i port to 0000_00B0, which is the desync word, bit-swapped.
11. Set the trigger position in window to 512.
12. Arm the trigger again and issue a reconfiguration.

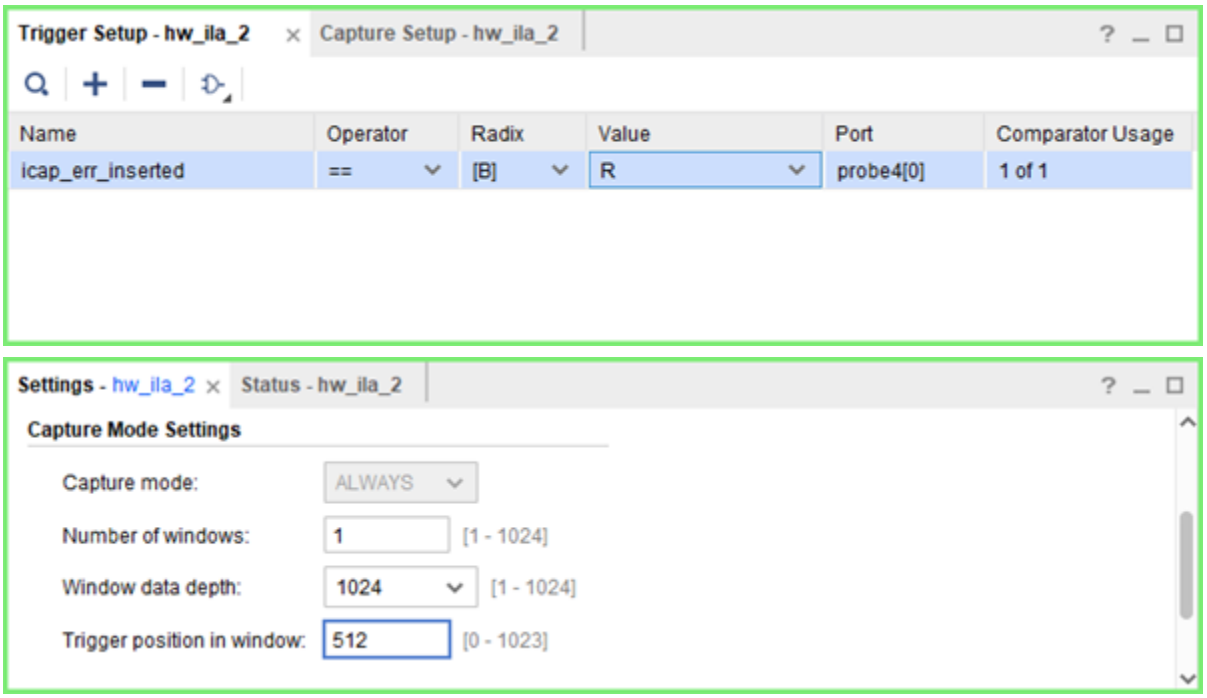
    The resulting waveform shows the end of this part of the reconfiguration sequence, and shows PRDONE going high a few clock cycles after the desync word was seen.

Insert CRC Failures and View the FPGA Response

Partial bitstreams with and without per-frame CRC checking were created and loaded into the QPSI flash as part of the PROM file. The design can insert CRC failures by swapping some bits in the CRC value just prior to loading the file into the ICAP. Any uncompressed – using bitstream generation property, not the DFX Controller feature – partial bitstream can have an error injected in this manner to see how the device responds. This is controlled via the center push button.

1. If not done in the prior section: In the Vivado Hardware Manager, refresh the device to find all the Vivado Debug cores.

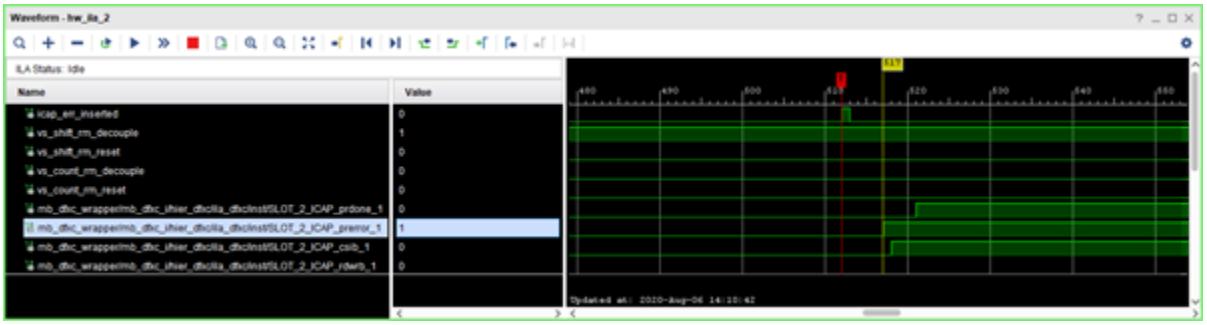
   There are three ILA cores, one VIO core and one MIG core in this design.
2. If not done in the prior section: In one of the ILA core windows, click the **Specify the probes file** links to find `Bitstreams/top_count_up_shift_right.ltx`.
3. Click **OK** then **Refresh**.
4. In hw_ila_2, click the + to add probes in the Trigger Setup window.
5. Select **icap_err_inserted** and click **OK**.
6. Set the trigger in the Trigger Settings window to rising edge of icap_error_inserted.
7. Set the trigger position in the Settings window to 512.



8. Push the **Run Trigger** button in Hardware Manager GUI.
9. In the UART terminal, set Reconfig Type as Normal CRC from DDR4 (option 7). This is the default setting.
10. Push the center pushbutton – this inserts the CRC error – and then reconfigure the module you would like via pushbutton or the UART terminal. The CRC value at the end of the partial bitstream is swapped to cause a CRC error. As a result, INIT_B goes low (the INIT_B LED turns red), indicating a CRC error. Notice on the board that the function not reconfigured is still operating.

   ***Note*:** On the KCU116 board, the INIT_B LED is on the underside of the board.

11. In ILA Waveform, the trigger location marks where icap_err_inserted is asserted. After the error insertion, you will see PRERROR then PRDONE goes high. Also, the `rm_decouple` signal is high throughout, indicating the Reconfigurable Partition is still isolated.



***Note*:** The dfxc_vsm_vs_*_event_error signal is low, but it will be pulsed outside of captured waveform because of the latency in Dynamic Function eXchange (DFX) Controller.

If the CRC error is found using the standard CRC, which only occurs at the end of the partial bitstream, the incorrect bitstream has already been loaded into the device. There is no way to know where any incorrect bits exist, or if they will disrupt the reconfigurable or static design. The only way to be sure of a full recovery from this condition is to perform a full reconfiguration of the device. In this tutorial, only the CRC value is swapped, so you can be assured that the error has not affected the static design.

12. When CRC error occurred, the DFX Controller entered shutdown mode. In the UART terminal, Report Status (option 9) shows the RP is in shutdown mode and it reports a BS ERROR. To recover from this error status, return the RP to active mode by selecting Put RPs in Active Mode (option 5) from the terminal, and reconfigure with a correct partial bit file. Then INIT_B returns high (LED turns green) and the design is now back to normal operation.

    Next, try with per-frame CRC values inserted.

13. In the terminal, set the Reconfig Type to per Frame CRC (option 7).
14. In the Hardware Manager, re-arm the trigger by clicking the Run Trigger button. Then perform Dynamic Function eXchange from the terminal or via pushbutton.
15. In the ILA waveform, you can see the error has been inserted in the first frame of partial bitstream – you can see that reconfiguration starts soon after vs_rm_*_decouple goes High. However, when using per-frame CRC, the error inserted frame has not been loaded into the device yet, so there is no need for a reconfiguration of the full design, just reconfigure the incomplete Reconfigurable Partition with a valid partial bitstream. To recover from the error status, repeat the procedure from step 8.

Lab 7 Conclusion

This concludes lab 7. In this lab, you:

- Implemented an UltraScale+ version of a design with the Dynamic Function eXchange (DFX) Controller.
- Compiled a MicroBlaze core with software that manages Dynamic Function eXchange events.
- Programmed the QSPI on the VCU118 board.
- Used a UART interface to manage Dynamic Function eXchange from QSPI or DDR4 memory.
- Inserted bitstream delivery errors to see CRC checking capabilities.

***Note*:** Although an SEM core exists within the design, it is not exercised during hardware testing in this lab. If the SEM core is running on hardware to detect upset events, it must be paused before performing Dynamic Function eXchange.

Lab 8

Nested Dynamic Function eXchange

Overview

This lab covers a simple example of Nested Dynamic Function eXchange (Nested DFX) targeting one of four AMD UltraScale™ or AMD UltraScale+™ demo boards. Nested DFX is the concept of placing one or more dynamic regions within a dynamic region, subdividing a device to permit more granular reconfiguration. With this feature, you can segment a Reconfigurable Partition (RP) into smaller regions, each of which is partially reconfigurable.

The design in this lab is a modified version of the LED-Shift-Count design used in other labs in this document. Instead of simply swapping different shifters or different counters, an additional reconfigurable layer has been inserted that enables you to have two shifters or two counters in the current design. Each of these shifters or counters are then individually partially reconfigurable.

The design flow uses the Tcl scripted solution used in Lab 2, as Nested DFX is not yet supported in project mode. You may follow the explicit instructions as shown in segmented Tcl scripts within this tutorial, or use a single full script that runs the full front-to-back processing of the complete design.

Step 1: Extracting the Tutorial Design Files

1. Download the reference design files.
2. Navigate to `\nested_dfx` in the extracted files. The `nested_dfx` data directory is referred to in this lab as the `<Extract_Dir>`.

Step 2: Examining the Scripts

Start by reviewing the Tcl scripts provided in the design archive.

The Synthesis Scripts

The files run_synth.tcl, design_settings.tcl and advanced_settings.tcl are located at the root level. The run_synth.tcl script contains the minimum required settings to run the synthesis portion of this Dynamic Function eXchange design. The design_settings.tcl script selects the target device and board and sets relative paths for the project. The advanced_settings.tcl contains default flow settings and should only be modified by experienced users.

In `run_synth.tcl` for this specific lab, under flow control, you can control which modules are synthesized. In the tutorial, as the name implies, only synthesis is run by this script; implementation, verification, and bitstream generation are run interactively. The full DFX scripts from Labs 1 and 2 are not currently set up to run Nested DFX.

In `design_settings.tcl`, under Define target demo board, you can select one of four demonstration boards supported for this design. The script is delivered targeting the VCU118, so if you wish to target a different board, make the edit here. This lab currently targets the following AMD development platforms:

- KCU116 (AMD Kintex™ UltraScale+™)
- VCU118 (AMD Virtex™ UltraScale+™)
- KCU105 (AMD Kintex™ UltraScale™)
- VCU108 (AMD Virtex™ UltraScale™)

The Nested DFX Scripts

During this lab, after synthesis, you will walk through the Nested DFX flow step-by-step using individual Tcl commands. This lab is designed to show the unique details required for inserting the second layer of reconfigurability and therefore highlights the new steps required to achieve this, but the entire solution can be scripted. Specific sections are grouped into scripts that can be run to compile a subsection of the flow. Explicit names and paths are used in these scripts, but they can certainly be modified for use in new designs. These scripts include:

- `implement_parent_config.tcl`: This implements the top-level static design and establishes the first-order Reconfigurable Partition (inst_RP) that will be later subdivided.
- `subdivide_shifters.tcl`: This subdivides the first-order Reconfigurable Partition into two second-order shift functions, each partially reconfigurable.
- `subdivide_counters.tcl`: This subdivides the first-order Reconfigurable Partition into two second-order count functions, each partially reconfigurable.
- `implement_sub_shifters.tcl`: This implements shift_right and shift_left Reconfigurable Modules in the two second-order RPs below inst_RP.
- `implement_sub_counters.tcl`: This implements count_up and count_down Reconfigurable Modules in the two second-order RPs below inst_RP.

- `verify_configurations.tcl`: This runs pr_verify on pairs of design checkpoints to confirm compatibility of Reconfigurable Modules contained within.
- `generate_all_bitstreams.tcl`: This opens checkpoints one-by-one to create all possible partial bitstreams for this overall design.
- `run_all.tcl`: This runs all scripts above, in order, from synthesis to bitstream generation, to compile the complete tutorial design.

The default board is the VCU118, but three other boards are available for selection. The board selection can be made just once within design_settings.tcl each of the implementation scripts will pick up the value, so even if a new Vivado session is launched, the design settings will be understood.

Step 3: Synthesizing the Design

The run_synth.tcl script automates the synthesis phase of this tutorial. Seven iterations of synthesis are called, one for the static top-level design, two for the first-order Reconfigurable Modules, and four for the second-order Reconfigurable Modules.

1. Open the AMD Vivado™ Tcl shell:

   On Windows, select the AMD Vivado desktop icon or **Start → All Programs → Xilinx Design Tools → Vivado 2026.1 → Vivado 2026.1 Tcl Shell**.

   On Linux, type: `vivado -mode tcl`.
2. In the shell, navigate to the <Extract_Dir> directory.
3. Confirm the target board is selected by the xboard variable in run_synth.tcl.
4. Run the run_synth.tcl script by entering:

```
source run_synth.tcl -notrace
```

After all the seven passes through Vivado Synthesis have completed, the Vivado Tcl shell is left open. You can find log and report files for each module, alongside the final checkpoints, under each named folder in the Synth subdirectory.



TIP: *In the <Extract_Dir> directory, multiple log files have been created:*

1. run.log shows the summary as posted in the Tcl shell window
2. command.log echoes all the individual steps run by the script
3. critical.log reports all critical warnings produced during the run

***Note*:** The `command.log` file is itself a Tcl run script. This file can be modified if desired and sources as an input to reproduce the same results as an alternative to the more complex and parameterized Tcl_HD scripts.

Step 4: Assembling and Implementing the Design

Now that the synthesized checkpoints for each module, plus top, are available, you can assemble the design. You will run all flow steps from the Tcl Console, but you can use features within the IDE (such as the floorplanning tool) for interactive events.

Implementation Design Flow

The steps in this lab are managed by a set of Tcl scripts that walk through the commands used to implement each configuration of the Nested DFX design. Examine each script before running to see what each does. Most commands (link_design, route_design, pr_verify, write_bitstream, etc.) will look familiar, and others (pr_subdivide, pr_recombine) are new. The key detail is the order in which they are run, as later scripts are dependent on earlier ones.

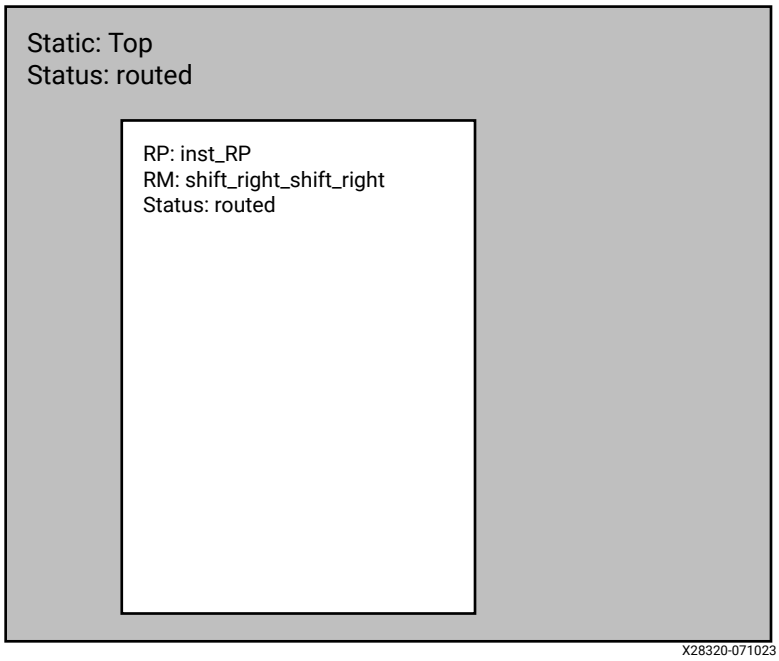
After scripts have completed, open checkpoints (for example top_route_design.dcp or top_count_up_up_route_design.dcp) to examine the results, noting the implications of the DFX attributes and commands used to create them.

The first implementation design run establishes the static design and the first-order Reconfigurable Partition. At this point, the flow is no different than a standard DFX design flow – “inst_RP” is the lone RP in the design, and the shifter modules that reside below that level are implemented with the rest of the inst_RP logic. Second-order Reconfigurable Partitions do not exist yet.

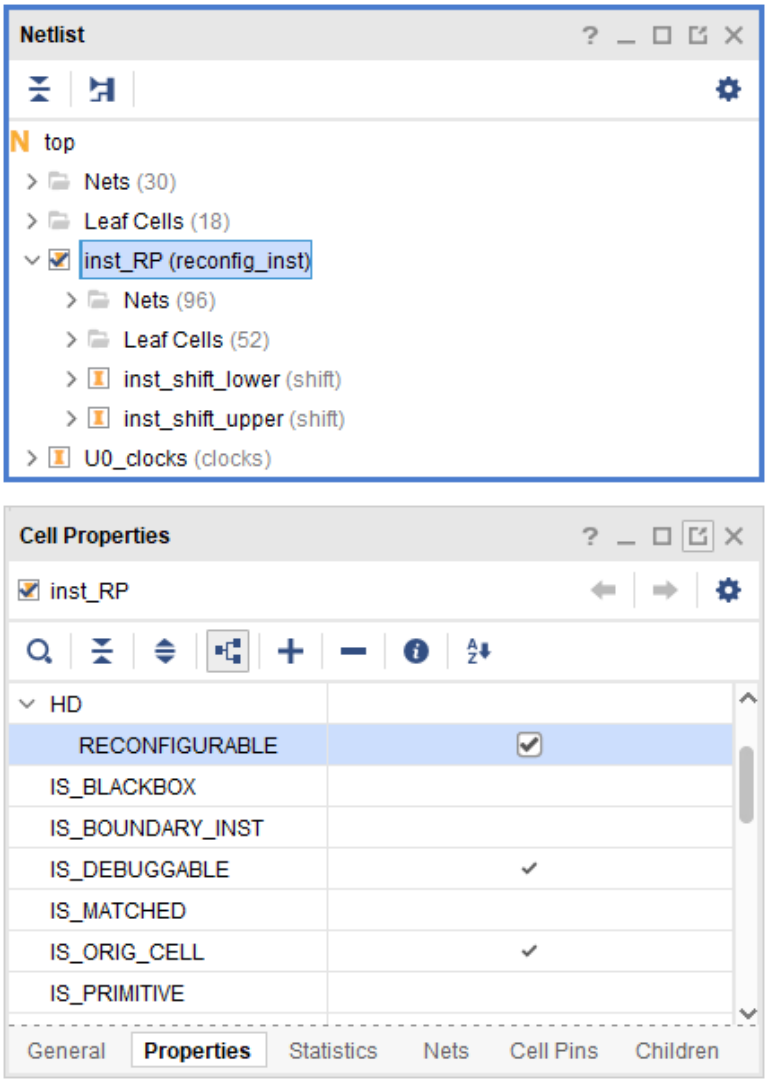
1. Implement the parent configuration by sourcing its run script:

   ```
   source implement_parent_config.tcl -notrace
   ```

   The resulting checkpoint (top_route_design.dcp) is a full design image with the single RP. No additional DFX steps such as carving the RP into a black box or locking the static design have been done at this point. This checkpoint will only be used to establish the locked static design image which is common to all design iterations that follow.



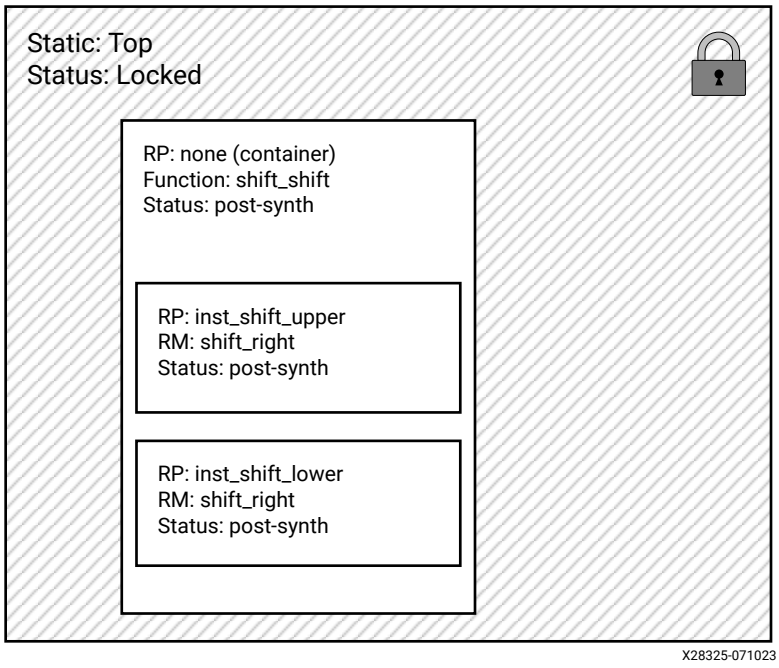
Open top_route_design.dcp that has been written to the Implement/top_static folder to see that this is a standard DFX design. inst_RP has the HD.RECONFIGURABLE property and an associated Pblock for the Reconfigurable Partition.

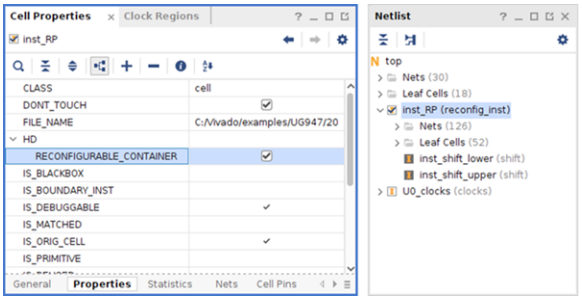


2. Create the second-order Reconfigurable Partitions by sourcing this script:

```
source subdivide_shifters.tcl
```

This script subdivides the inst_rp module into second-order Reconfigurable Partitions. The pr_subdivide command removes the HD.RECONFIGURABLE property from inst_RP and applies it to both inst_shift_upper and inst_shift_lower. inst_RP is then tagged with the HD.RECONFIGURABLE_CONTAINER property, noting that it was once an RP. This can be seen by examining the top_static_shifters.dcp checkpoint.





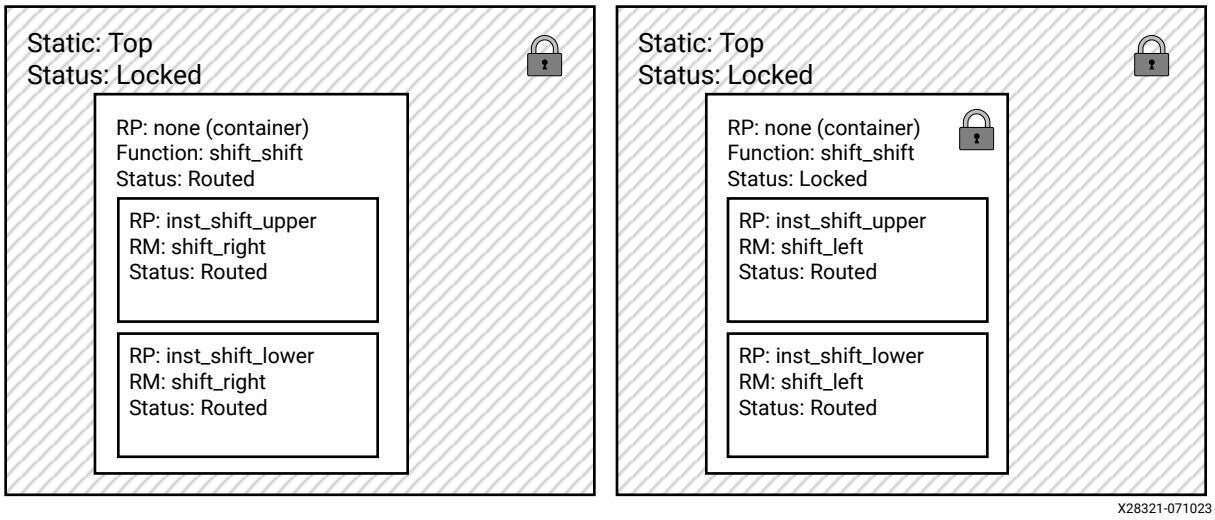
The HD.RECONFIGURABLE_CONTAINER property can also be queried directly by checking the property on the inst_RP hierarchical instance. The following Tcl command will return a value of 1.

```
get_property HD.RECONFIGURABLE_CONTAINER [get_cells inst_RP]
```

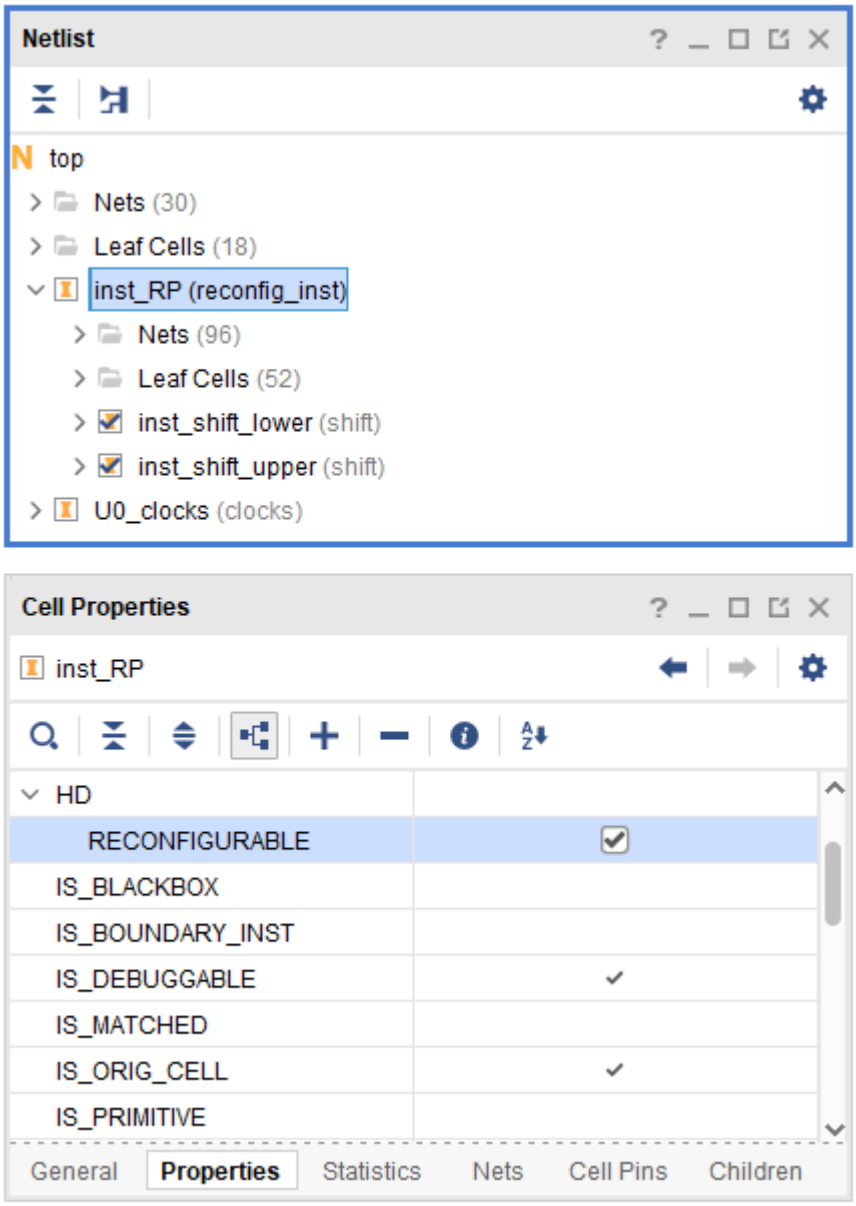
3. Implement the shifter submodules in second order Reconfigurable Partitions.

```
source implement_sub_shifters.tcl -notrace
```

This walks through two implementation flows to place and route shift_right and shift_left functions in the second-order RPs. The commands used here are identical to a standard DFX flow with one difference: The starting point of the first configuration includes the locked top-level static design. Implementation treats the logical design in the inst_RP level (reconfig_shifters) as static; this is the level of hierarchy that is locked by the `lock_design -level routing` command after the first configuration completes.



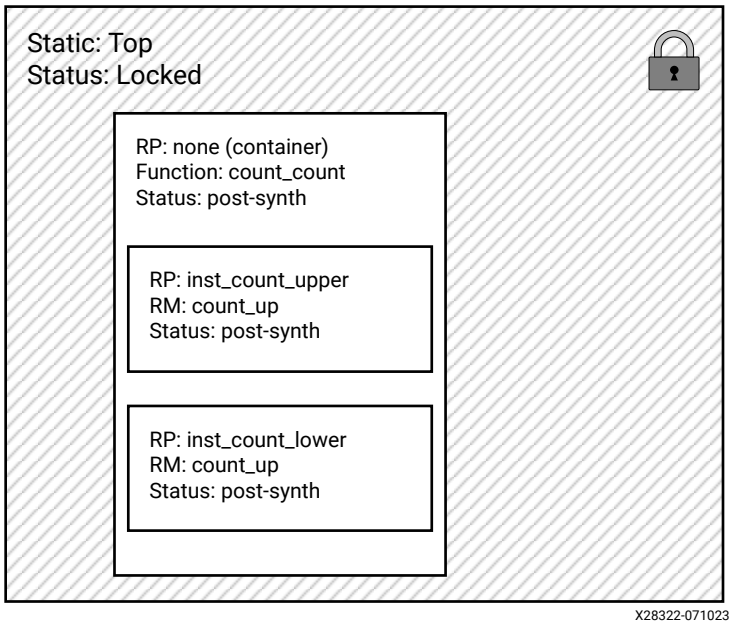
This script finishes with a call to pr_recombine to create a routed design checkpoint of the shift_right-shift_right combination, moving the HD.RECONFIGURABLE property back to the inst_RP level. Examining the hierarchy of `top_shift_right_right_recombined.dcp` you can see this property has returned to the inst_RP instance.



4. Create another set of second-order Reconfigurable Partitions sourcing this script:

```
source subdivide_counters.tcl
```

Just like the first subdivide script, this starts with the initial configuration (top_route_design.dcp) and subdivides the inst_RP level, but this time into two counter functions. The top-level static for this design version is identical to the version used for the shifters.

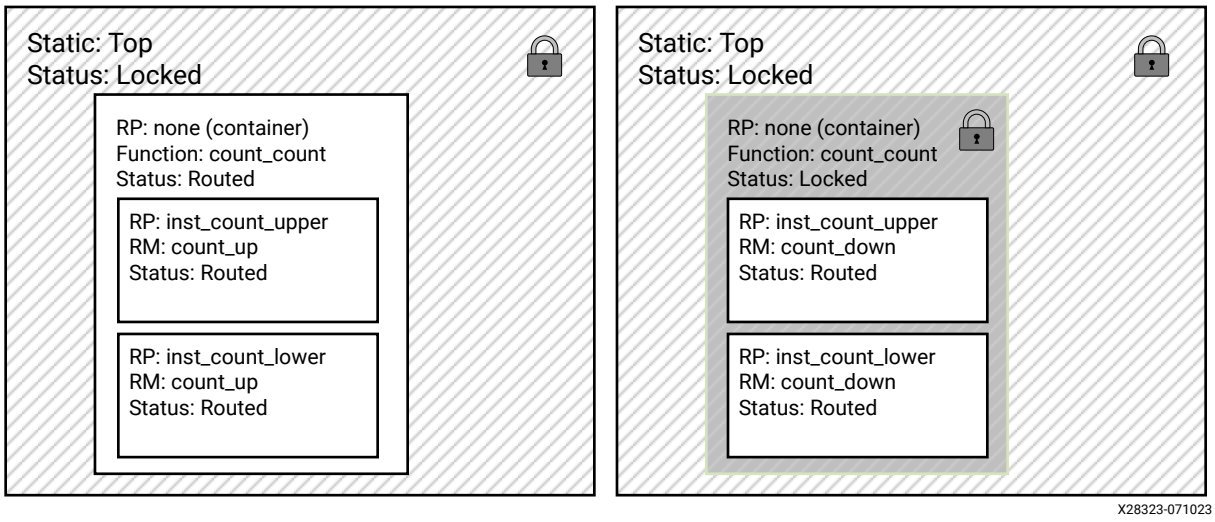


5. Implement the counter submodules in second order Reconfigurable Partitions.

```
source implement_sub_counters.tcl -notrace
```

Again, like with the shifters path, a standard DFX flow is used to process two counter modules (count_up, count_down) in each second-order Reconfigurable Partition. Also like the shifters, a recombined design checkpoint is created from the first pass through the second-order flow.

***Note*:** The two second-order implementation scripts (as well as the subdivide scripts that precede them) can be run in parallel in two unique Vivado sessions. Both rely on the same locked top static design but are unique from the inst_RP level on down. This version has slightly different floorplans for the second-order RPs, but could also vary in number.



Static Design Updates

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

For example, if there is a design change for the top-level static, all existing results must be considered out-of-date and everything must be recompiled. If there is an update to one of the first-order RMs (reconfig_shifters or reconfig_counters), all results dependent on the modified module must be recompiled. Any of these individual scripts can be called on their own to update the results as needed.

Verification Passes

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

```
source verify_configurations.tcl
```

This script compares three pairs of routed designs. Each does a pairwise comparison of checkpoints with static logic expected to be identical. This section describes the comparisons done and the compatible bitstreams that will be created in the next step.

1. The first call to `pr_verify` compares the two recombined checkpoints. These should each have identical static implementation results top only, with a single Reconfigurable Partition, inst_RP. These checkpoints represent standard DFX designs with no nesting, even though each could receive appropriate second-order partial bitstreams.

   If other checkpoints are created with second-order modules (for example `shift_left`, `count_down`) and then recombined, they could be compared via `pr_verify` and have their "inst_RP" partial bitstreams added to this compatibility list. This would also be true for any other RMs for inst_RP even without any subdivided second-order RPs.

2. The second call to pr_verify compares the shift_right and shift_left second level checkpoints. These have static locked down to the upper and lower submodules, so the comparison is between this static logic for the top and reconfig_shifters levels of hierarchy.

3. Much like the second, the third call to pr_verify compares the count_up and count down second level checkpoints. These have static locked for top and reconfig_counters, so the comparison is between this static logic down to the upper and lower Reconfigurable Partitions.

Bitstream Creation

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

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

- The `-cell` option will generate ONLY a partial bitstream for the requested cell.
- The `-no_partial_bitfile` option will generate ONLY a standard full device bitstream.

Run the following script to create a collection of full and partial bitstreams for existing configurations that have been implemented. To save time and space, only a single full device bitstream is created.

```
source generate_all_bitstreams.tcl
```

This script opens each checkpoint, one by one, and writes specific full or partial bitstreams. The bitstreams are placed in subfolders based on compatibility. Each bitstream is created with either the `-no_partial_bitfile` option (the first bitstream listed below) or the `-cell` option (every other bitstream). The use of the latter means that partial bit file names can be anything you desire; use names that clearly indicate function, version and compatibility. Here are the eleven bitstreams generated in their respective folders:

- Bitstreams
  - top_shift_right_right.bit
- Bitstreams/inst_RP
  - inst_RP_shift_right_right_recombined_partial.bit
  - inst_RP_count_up_up_recombined_partial.bit
- Bitstreams/inst_shift
  - shift_right_upper_partial.bit
  - shift_right_lower_partial.bit
  - shift_left_upper_partial.bit
  - shift_left_lower_partial.bit
- Bitstreams/inst_count
  - count_up_upper_partial.bit

- count_up_lower_partial.bit
- count_down_upper_partial.bit
- count_down_lower_partial.bit

In addition to these, clearing bit files are created for UltraScale devices, one for each partial bit file listed above. The base names are the same but end in "_clear." More information about how these are to be used is given in the next section of this lab.

Commands are also included to create, implement, and generate partial bitstreams for grey box configurations. These are not required for the solution but can be used to "turn off" activity within a particular RP. Set the grey parameter to "true" before sourcing the generate_all_bitstreams.tcl script to create these optional partial bitstreams.

Grey box partial (and clearing) bitstreams are established for each second-order RP (4 in total) and the first-order RP (inst_RP), as well as the case of each inst_RP RM instance (reconfig_shifters, reconfig_counters) with grey boxes for the second-order RPs within them.

- Bitstreams/inst_RP
  - inst_RP_grey_partial.bit
  - reconfig_shifters_grey_grey_partial.bit
  - reconfig_counters_grey_grey_partial.bit
- Bitstreams/inst_shift
  - shift_upper_grey_partial.bit
  - shift_lower_grey_partial.bit
- Bitstreams/inst_count
  - count_upper_grey_partial.bit
  - count_lower_grey_partial.bit

Note that no bitstreams have been created from the initial top_route_design.dcp checkpoint. This is because there is no need – the top-level static image for this design is identical to all others, and the shift_right-shift_right function is logically the same to the first subdivided run. The implementation results for the latter are different because of the new RPs introduced, but if you were to load a shift_right-shift_right partial image from before the subdivide, you could not individually swap out the second-order shifters.

For this lab a full device bitstream is created only for this shift_right-shift_right version of the design, but a full device bitstream could be generated for any legal combination of first- and second-order Reconfigurable Modules. It simply depends on how you would like the device to initially behave. You could create a count_up-count_down version, or a shifter version with grey boxes for each second-order RP, all by linking routed module checkpoints with the locked top static, then calling write_bitstream.

In summary, build the design results from the top down, locking each relative static layer using the pr_subdivide function. Then, to return to higher-level Reconfigurable Partitions, use pr_recombine to create checkpoints for generating partial bitstreams at that level.

Step 5: Test the Design in Hardware

With a set of full and partial bitstreams created, the design can be tested on one of the four demonstration boards. The current design supports the KCU105, VCU108, KCU116, and VCU118 boards, revisions Rev 1.0 and newer.

Configuring the Device with a Full Image

1. Connect the board to your computer using the Platform Cable USB and power on the board.
2. From the main Vivado IDE, select **Flow > Open Hardware Manager**.
3. Select Open target on the green banner. Follow the steps in the wizard to establish communication with the board.
4. Right-click the **Xilinx device** (e.g., xcku040_0) and select Program Device.
5. Navigate to the Bitstreams folder to select `top_shift_right_right.bit`, then click **Program** to program the device.

You should now see the bank of GPIO LEDs performing two identical tasks: two sets of four LEDs are shifting to the right. Note the amount of time it took to configure the full device.

The currently operating device contains top (static), first-order RM reconfig_shifters, and second-order RMs shift_right (upper) and shift_right (lower).

Partially Reconfiguring the Device

At this point, you can partially reconfigure the active device with any of the partial bitstreams that you have created, but only if it is compatible with the currently loaded design. For UltraScale devices, you must always first start with the appropriate clearing bitstream(s). Because of this, the following section is split by family, as UltraScale+ instructions are simpler. The instructions below have been separated by family for clarity.

Load partial bitstreams for UltraScale+ devices

If you are targeting the VCU118 or KCU116, follow these instructions. UltraScale device instructions are further on in this section.

First, reconfigure the “upper” shifter location.

1. Select **Program device** on the green banner (or right-click on the target device and select **Program device**). Navigate to the Bitstreams/inst_shift folder to select `shift_left_upper_partial.bit`, then click **Program** to program the device. The upper shift portion is now shifting left, while the lower portion is still shifting right. DONE has also returned high (on).

   In order to transition to counter functions, the first-order reconfig_counters RM must first be loaded. Loading any second-order count_up or count_down partial bitstreams at this point would not function properly, as these functions would not connect to the top-level static design.

2. Select **Program device** on the green banner again. Navigate to the Bitstreams/inst_RP folder to select inst_RP_count_up_up_recombined_partial.bit, then click **Program** to program the device. The two sections of LEDs are now counting up.

   Now that the reconfig_counters first-order function is established, Reconfigurable Partitions below that hierarchy can be partially reconfigured.

3. Select **Program device** on the green banner one last time. Navigate to the Bitstreams/inst_count folder to select count_down_lower_partial.bit, then click **OK** to program the device. The upper shift portion is still counting up, while the lower portion is now counting down.

   This concludes the lab instructions for UltraScale+ devices.

Load partial bitstreams for UltraScale devices

If you are targeting the VCU108 or KCU105, follow these instructions. First, reconfigure the “upper” shifter location.

1. Select **Program device** on the green banner (or right-click on the target device and select **Program device**). Navigate to the Bitstreams/inst_shift folder to select shift_right_upper_partial_clear.bit, then click **Program** to program the device. The upper shift portion of the LEDs stopped, but the lower shift portion kept shifting, unaffected by the reconfiguration. Note the much shorter configuration time, as well as the fact that the DONE LED has turned off.

2. Select **Program device** on the green banner again. Navigate to the Bitstreams/inst_shift folder to select shift_left_upper_partial.bit, then click **Program** to program the device. The upper shift portion is now shifting left, while the lower portion is still shifting right. DONE has also returned high (on).

In order to transition to counter functions, the first-order reconfig_counters RM must first be loaded. Loading any second-order count_up or count_down partial bitstreams at this point would not function properly, as these functions would not connect to the top-level static design.

> **IMPORTANT!** *Moreover, for UltraScale devices, clearing bitstreams must be applied, from the bottom up, before a new first-order partial bitstream can be delivered. Each clearing bitstream must match the currently loaded function at that level of hierarchy. The order in which the second-order clearing bitstreams does not matter, but they must precede the first-order clearing bitstream.*

3. Select **Program device** on the green banner and program the device using these clearing bitstreams, one at a time.
   - inst_shift/shift_left_upper_partial_clear.bit
   - inst_shift/shift_right_lower_partial_clear.bit
   - inst_RP/inst_RP_shift_right_right_recombined_partial_clear.bit

   Following these actions, all LED activity has stopped, as the functionality of the shifters and the connectivity to the top static have been removed from the active design.
4. Select **Program device** on the green banner again. Navigate to the Bitstreams/inst_RP folder to select inst_RP_count_up_up_recombined_partial.bit, then click **Program** to program the device. The two sections of LEDs are now counting up.

   Now that the reconfig_counters first-order function is established, Reconfigurable Partitions below that hierarchy can be partially reconfigured.
5. Select **Program device** on the green banner. Navigate to the Bitstreams/inst_count folder to select count_up_lower_partial_clear.bit then click **Program** to program the device. This stops the counter that is driving the count function in the lower set of LEDs.
6. Select **Program device** on the green banner one last time. Navigate to the Bitstreams folder to select count_down_lower_partial.bit, then click **OK** to program the device. The upper shift portion is still counting up, while the lower portion is now counting down.

   This concludes the lab instructions for UltraScale devices.

Lab 8 Conclusion

This concludes lab 8. In this lab, you:

- Synthesized a design bottom-up to prepare for Nested Dynamic Function eXchange implementation
- Applied pr_subdivide and pr_recombine to create nested levels of Reconfigurable Partitions
- Implemented multiple configurations via scripts.
- Compared checkpoints pairwise for static design consistency.

- Examined framesets and verified the two configurations.
- Configured and partially reconfigured an FPGA with first-order and second-order partial images

Lab 9

Abstract Shell for Dynamic Function eXchange

Overview

The AMD Vivado™ software tool flow lets you compile DFX designs using an in-context methodology. This solution requires multiple passes through place and route. The first pass establishes the static design implementation result (along with the first Reconfigurable Module (RM) for each Reconfigurable Partition (RP)). Then all subsequent place and runs are done in context with that initial static image. A fully routed and locked static design database, containing netlist, placement, and routing information for the entire static region, must be loaded into Vivado before implementing any RMs beyond the first.

The Abstract Shell solution reduces the requirements for this in-context flow. Because the static design is locked, it cannot (and must not) be modified when new RMs are implemented. The context is still critical, so the path through the tools does not change. However, instead of loading a full static design image, an Abstract Shell checkpoint is used. This Abstract Shell contains only a minimal logical and physical database necessary to implement a new RM within a specific RP to validate timing and pass PR Verify, and then generate a partial bitstream for that RM.

This lab uses the DFX Controller IP design shown in Lab 7. The first pass through place and route is identical to the run completed in Lab 7, but then all child runs to implement new RMs are done within Abstract Shells. The end result is a collection of design checkpoints that can be used to program the VCU118 in the same way that was done in Lab 7, but compilation time for producing the child RMs is reduced.

Step 1: Extracting the Tutorial Design Files

1. Download the reference design files.
2. Extract the zip file contents to any write-accessible location.
3. In the extracted files hierarchy, navigate to `\abstract_shell`.

Step 2: Processing the Tutorial Design

The purpose of this design is to review the design flow process using Abstract Shell, so the details of the implementation flow and hardware operation are not extensively covered here. For a review of the Dynamic Function eXchange design flow, refer to earlier labs in this document. For details on the DFX Controller IP and hardware operation, review Lab 7 in this document.

1. Extract the tutorial design archive.
2. From a command shell, launch Vivado with the example design project creation script. This must be launched from the directory where the script is located.

   ```
   vivado -mode tcl -source project_dfxc_vcu118.tcl
   ```

3. When the script completes, open the Vivado IDE by typing the following: `start_gui`.
4. Check to see if IP needs to be updated. Run **Reports → Report IP Status** and update any out-of-date IP.

   A few minor revision changes, such as for ILA, might be found if using a newer version of Vivado. Please use this tutorial only with the version of Vivado that matches this document version. If updates must be made, use the default setting, which has the core container disabled, but skip the actual synthesis of the IP module – this will be done during the next step.

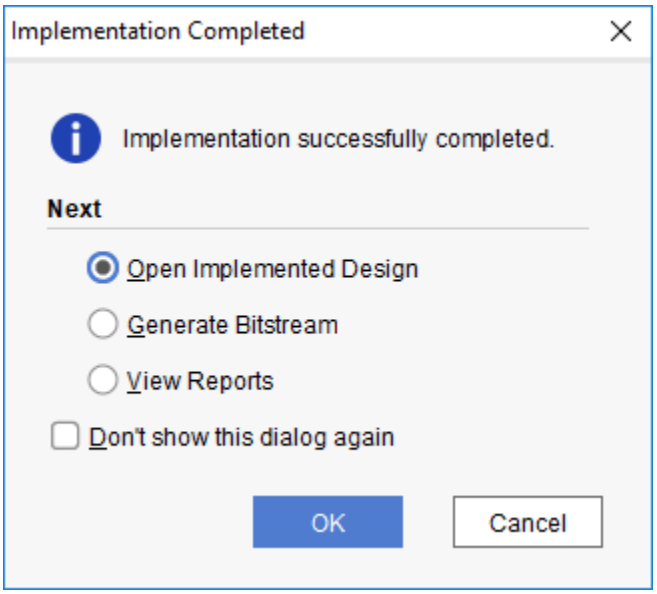
5. In the Flow Navigator, under the IP INTEGRATOR heading, click **Generate Block Design** to prepare the design for processing. Leave the **Out of context per IP Synthesis** option selected, then click **Generate**.

   This step creates all the IP identified in the block design and launches them through synthesis. For this design, this block design covers the vast majority of the static logic representing the design infrastructure. This step does not launch out-of-context synthesis for the RTL submodules, which includes the shift and count Reconfigurable Modules.

6. In the Design Runs tab, right-click on impl_1 and select **Launch Runs**. This will pull the design all the way through synthesis and implementation for the parent run only.

   ***Note*:** Do not use **Run Implementation** from the Flow Navigator, as this will launch both the impl_1 run as well as the child_0_impl_1 run. The latter should not be implemented at this point.

7. When impl_1 completes, select **Open Implemented Design** in the resulting dialog box.



With the routed parent design open in the Vivado IDE, you are ready to create Abstract Shells.

Step 3: Create Abstract Shells

By default, in the DFX flow, multiple design checkpoints will be written after place and route of the parent configuration completes. In addition to the full design routed checkpoint, the Vivado project flow will create a static-only design checkpoint that will be the starting point for all child runs by calling `update_design -black_box` for each Reconfigurable Partition, followed by `lock_design -level routing`. Moreover, module-level checkpoints are written for each RM in the parent configuration by calling `write_checkpoint -cell`. No user intervention is necessary to create these files.

1. Examine the files created for the parent configuration. Within Windows Explorer or in a shell console, navigate to the impl_1 subdirectory:

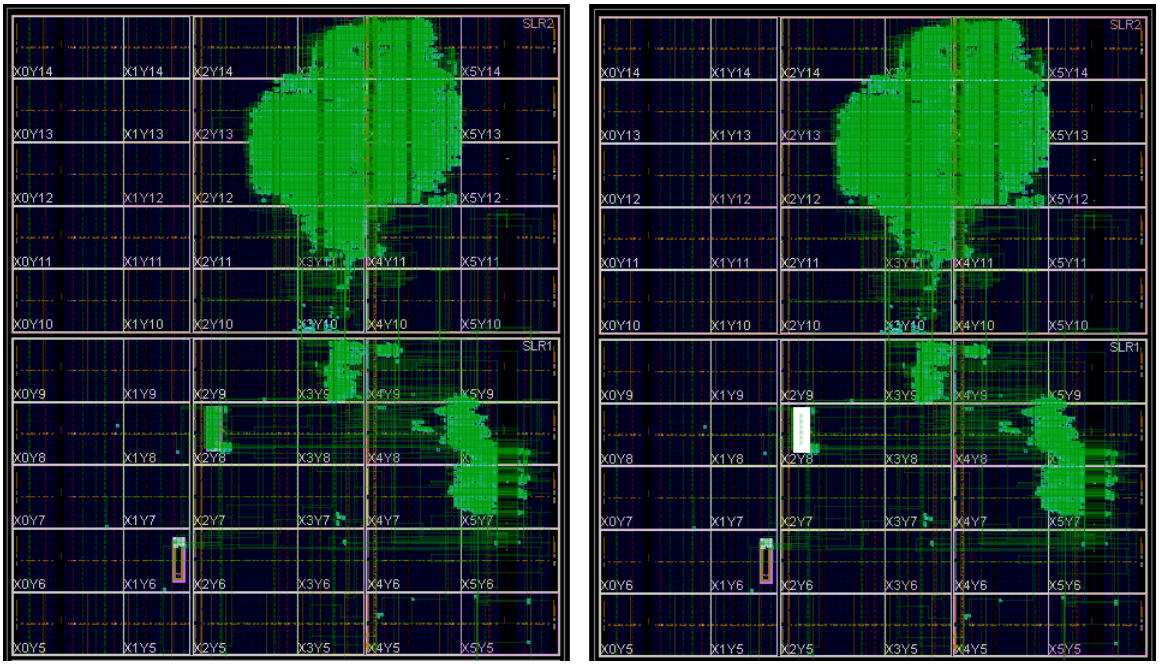
   ```
   \abstract_shell\dfxc_vcu118\project_dfxc_vcu118\project_dfxc_vcu118.runs\
   impl_1
   ```

   Examine the different design checkpoints and their sizes. Note that file sizes listed here may be slightly different depending on Vivado tool version, implementation run options and operating system. Key files include:

   - `top_routed.dcp` (58,284 KB) – full routed design including one RM per RP
   - `top_routed_bb.dcp` (55,819 KB) – static only design with locked placement and routing and black boxes for each RP
   - `u_count_count_up_routed.dcp` (1,267 KB) – routed module-level checkpoint for the count_up RM instance

- `u_shift_shift_right_routed.dcp` (463 KB) – routed module-level checkpoint for the shift_right RM instance

It is no surprise the Reconfigurable Module checkpoints are much smaller than the static design checkpoints given their size and complexity in this design. The following figure shows the full design checkpoint on the left and the static-only checkpoint on the right.



2. Create Abstract Shells for both the u_count and u_shift instances. Make sure your current working directory in the Tcl Console is the <extract_dir> directory, the same place where the project_dfxc_vcu118, sources and abstract_shell folders reside.

```
write_abstract_shell -force -cell u_count ./abstract_shell/
ab_sh_count.dcp
write_abstract_shell -force -cell u_shift ./abstract_shell/
ab_sh_shift.dcp
```

Each call to `write_abstract_shell` first creates a copy of the full design checkpoint in memory, then runs the following steps automatically:

- Carves out the target Reconfigurable Partition (using `update_design -black_box`)
- Locks the remaining design (including any other Reconfigurable Modules)
- Writes the Abstract Shell for the target RP
- Runs `pr_verify` for this checkpoint compared to the original fully routed design

This process does take longer than a simple call to `write_checkpoint`, but in nearly all cases the runtime savings for RM compilation will be worth this initial investment.

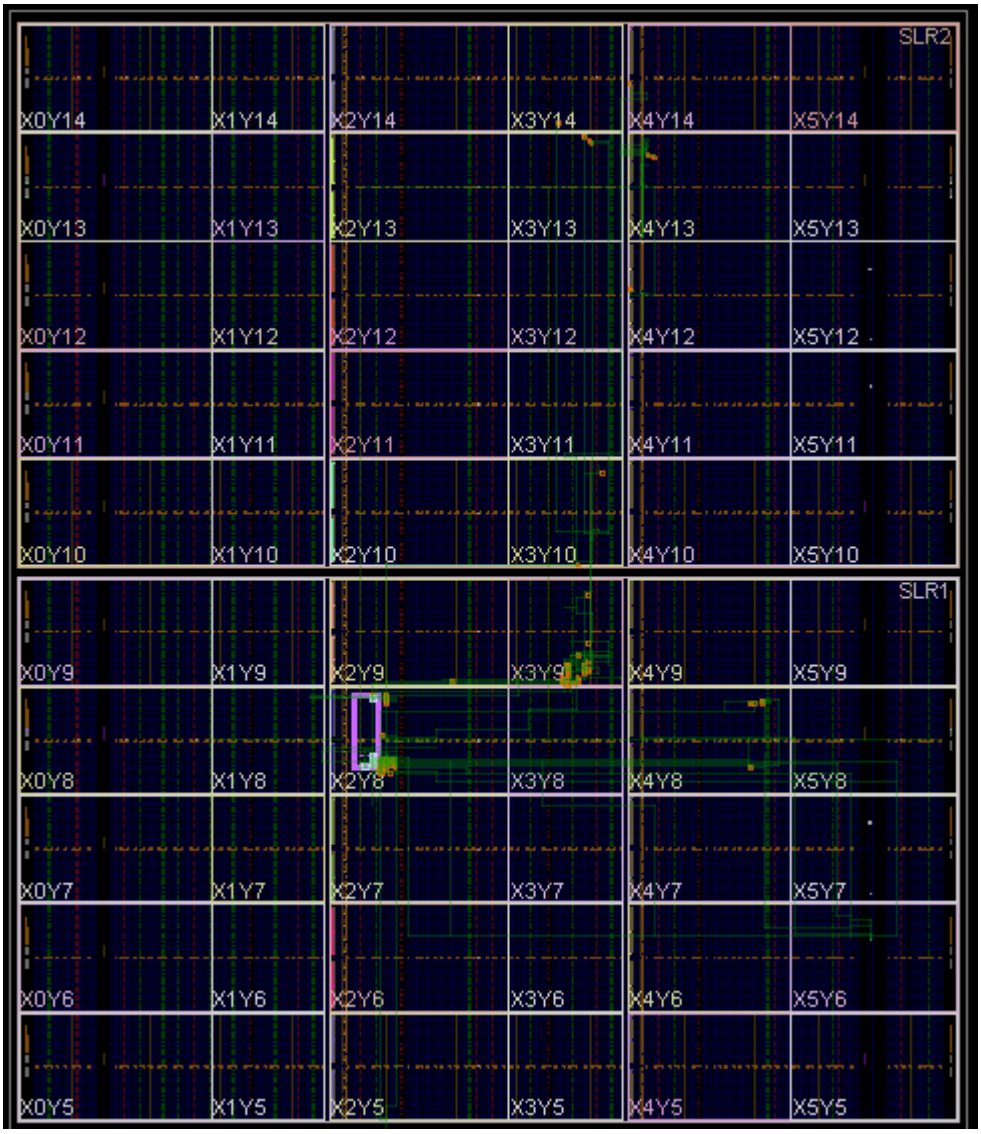
3. Examine the sizes of the Abstract Shells, comparing them to the size of the top_routed_bb.dcp full shell checkpoint.

Again, sizes may vary, but for the initial release of Vivado 2020.2 in Windows, file sizes for the Abstract Shells are:

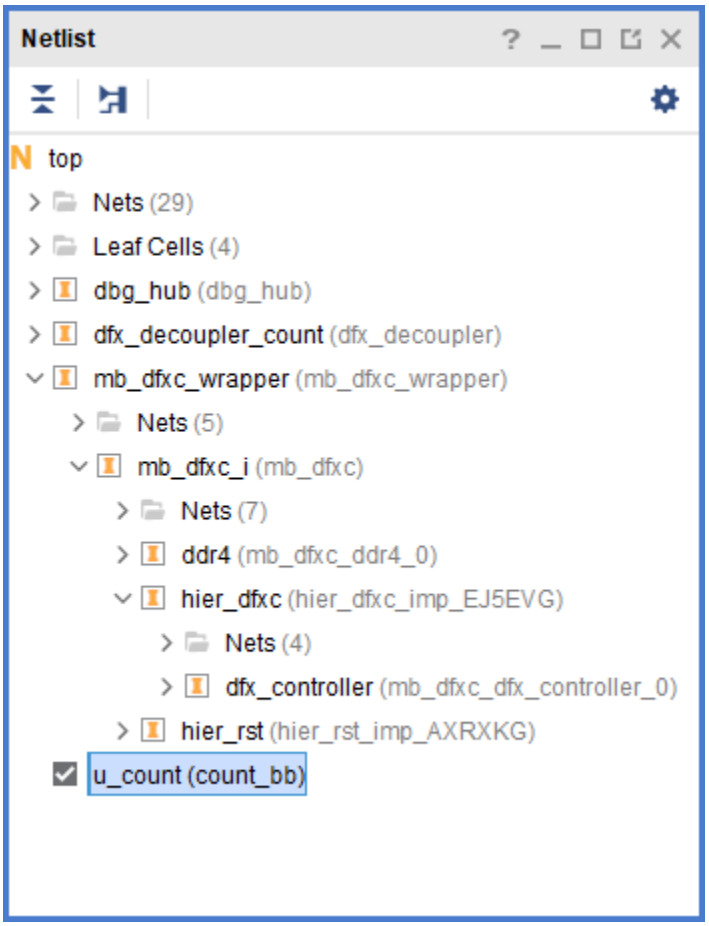
- `ab_sh_count.dcp` (1,785 KB) – Abstract Shell for the Count RP
- `ab_sh_shift.dcp` (1,699 KB) – Abstract Shell for the Shift RP

4. Open each Abstract Shell checkpoint to examine the contents.

```
open_checkpoint ./abstract_shell/ab_sh_count.dcp
```



Note how much of the static design is no longer present. Visually this is quite clear for this simple design – compare the figure in step 1 with the figure in step 4. You will see that only one RP remains in each Abstract Shell (the shell for u_count does not include u_shift and vice versa). But even though the vast majority of the static design has been removed, parts do remain, including elements from the DFX Controller and DFX Decoupler design, as they have connectivity to each target RP.



5. Run a routing report to confirm that the Abstract Shell is intact.

   ```
   report_route_status
   ```

   This step is optional and merely shows that the Abstract Shell is a valid design database with zero routing errors.

6. Close the Abstract Shell checkpoint.

   ```
   close_project
   ```

Step 4: Implement New RM within Abstract Shells

At this point, all remaining Reconfigurable Modules can be implemented within these shells. Each RM can be implemented in parallel in separate Vivado sessions if desired, as each RP can be managed independently. This can be done not only for a specific shell, for example the `u_count` instance, but for all Reconfigurable Partitions in the design. Unlike within the project flow where the focus is on full design configurations, the focus for the Abstract Shell approach is on the Reconfigurable Modules.

1. Start a new Vivado session to work independent of the initial VCU118 example design project. In the Tcl Console, navigate to the tutorial directory.

   IMPORTANT! *Use the same methodology in the Abstract Shell run as was used in the run that created the original implementation. Vivado project mode uses the add_files / link_design approach, and that is continued here. If you used open_checkpoint and read_checkpoint -cell to build the initial design, continue that approach for the Abstract Shell implementation run.*

2. Load an Abstract Shell checkpoint instead of the full static checkpoint.

```
add_files ./abstract_shell/ab_sh_count.dcp
```

3. Then add the post-synthesis netlist for only the count_down module.

```
add_files ./project_dfxc_vcu118/project_dfxc_vcu118.runs/
count_down_synth_1/count.dcp
```

```
set_property SCOPED_TO_CELLS {u_count} [get_files ./project_dfxc_vcu118/
project_dfxc_vcu118.runs/count_down_synth_1/count.dcp]
```

4. When linking the design, only reference the u_count RP.

```
link_design -mode default -reconfig_partitions {u_count} -part
xcvu9p-flga2104-2L-e -top top
```

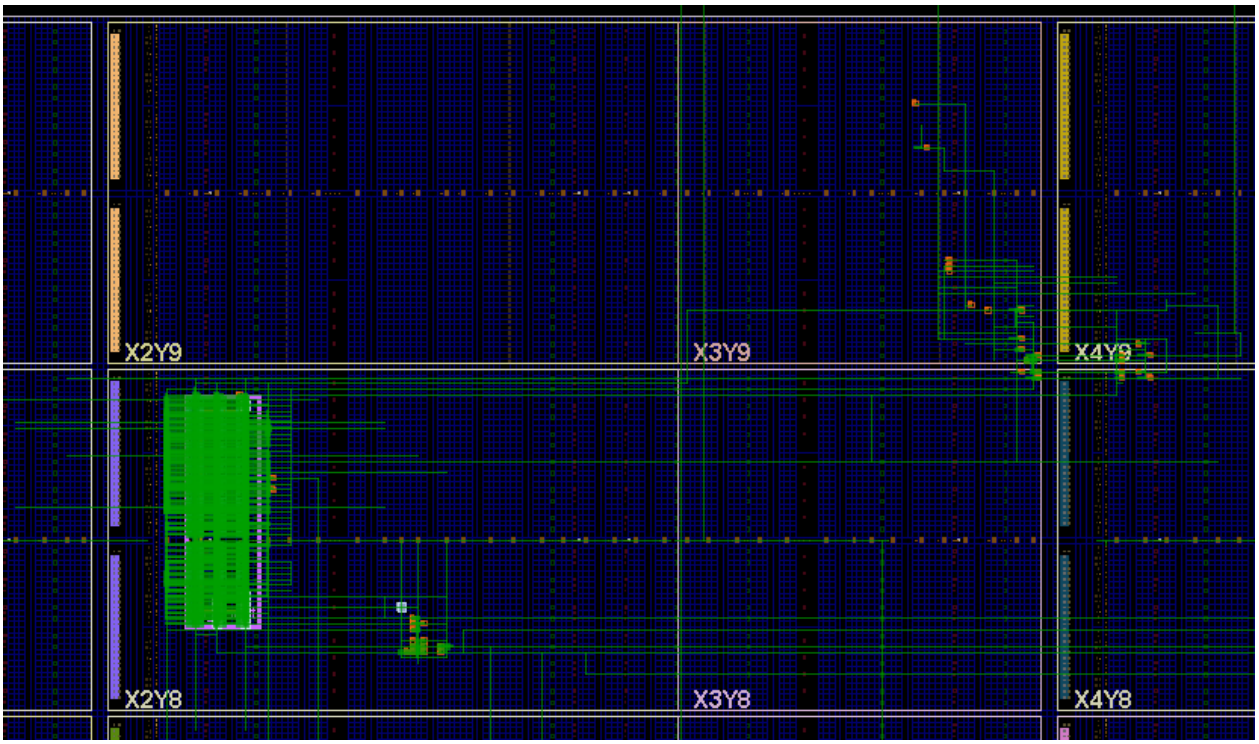
5. Implement the design normally, then when it is time to save the routed design, save both the complete current image (Abstract Shell plus Reconfigurable Module) as well as the RM-only checkpoint.

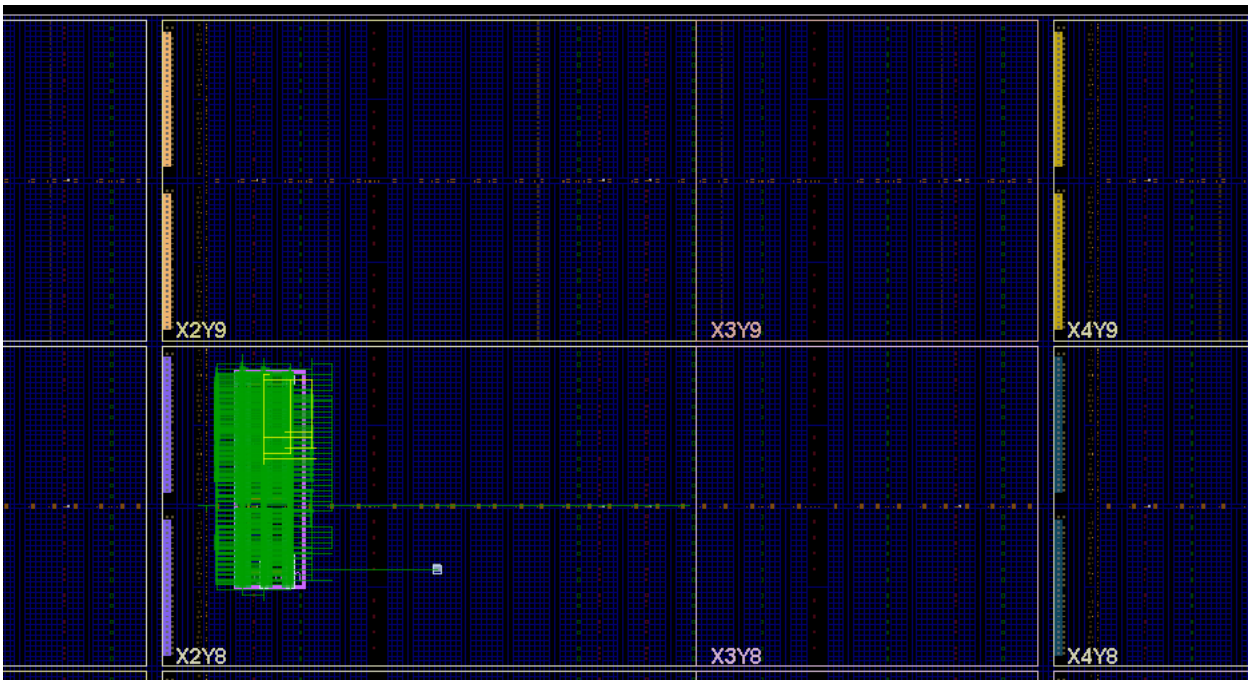
```
opt_design
place_design
route_design
write_checkpoint -force ./abstract_shell/abs_count_down/
abstract_shell_count_down_routed.dcp
write_checkpoint -force -cell u_count ./abstract_shell/abs_count_down/
rm_count_down_route_design.dcp
```

   Repeat these steps for the shift_left module, adjusting the file names and commands accordingly. Tcl scripts for both count_down and shift_left abstract shell implementation can be found in the tutorial directory to automate these steps. Script names are:

- `abs_impl_count_down.tcl`
- `abs_impl_shift_left.tcl`

In the following images compare the count_down RM within the Count RP Abstract Shell with the module-level checkpoint for the count_down RM alone. Only the former should be used for partial bitstream generation, as the latter does not contain static design information in the dynamic region.





At this point you have a collection of routed static and RM checkpoints, where all the RM checkpoints are in sync with the static design.

Note: Before considering partial bitstream generation, PR Verify should always be done. PR Verify compares multiple design images where RMs differ but static is the same to ensure all DFX rules have been followed. If full configuration assembly is done, then PR Verify can be run in the standard way, comparing the entire static design for each checkpoint. However, PR Verify can also be done in the Abstract Shell context, comparing the initial Abstract Shell to the shell with the routed Reconfigurable Module.

6. Using the checkpoints created above, this verification check can be done if no checkpoints are currently open:

```
pr_verify ./abstract_shell/ab_sh_count.dcp ./abstract_shell/
abs_count_down/abstract_shell_count_down_routed.dcp
```

Alternatively, if a routed Abstract Shell checkpoint is still open in Vivado, you can use the `-in_memory` option to compare to the original shell. For example, if the Abstract Shell for u_count with count_down implemented within it is still open use this command to run PR Verify:

```
pr_verify -in_memory -additional ./abstract_shell/ab_sh_count.dcp
```

Note: The comparison here is between the Abstract Shell for `u_shift` with a black box and the Abstract Shell for `u_shift` with either `shift_left` or `shift_right` implemented within it. The goal is to compare different RM implementations with command static checkpoints. PR Verify fails if:

- A full static design checkpoint is compared to an Abstract Shell checkpoint
- An RM checkpoint is loaded without its Abstract Shell

- Abstract Shells for different Reconfigurable Partitions are compared

Step 5: Validate the Design in Hardware

In order to test this tutorial design in hardware, a few additional steps are required. These steps include:

- Building the Vitis application project to run in MicroBlaze
- Creating the full design bitstream with this application present
- Generating all partial bitstreams and the PROM image
- Loading the PROM image in hardware and running hardware tests

Because this design is the same one used for the DFX Controller tutorial in Lab 7, the same process is used to generate the software application and operate the design in hardware. Rather than reiterate these details here, the following steps will reference the appropriate steps in Lab 7. However, given that the Abstract Shell solution was used to generate some of the partial bitstreams, the bitstream generation scripts have been modified.

1. If the main project_dfxc_vcu118 project has been closed, reopen it within Vivado.
2. Turn to Lab 7, Step 2, Instruction 8 and follow this lab through Instruction 23.

   Bitstream generation can be done in two ways when using Abstract Shell. The first is the standard way, where a full design is open in Vivado and both full and partial bitstreams are generated. Alternatively, partial bitstreams only can be generated directly from the Abstract Shell implementation for any RM.

   The following two sections describe the Vivado Tcl commands used to create partial bitstreams using each of these methodologies. The set of commands are embedded in the Tcl script noted at the beginning of each section. Choose one approach and call the script for that approach before moving on to hardware validation.

- Approach 1: Generate Partial Bitstreams from Full Configurations
- Approach 2: Generate Partial Bitstreams from Abstract Shells

Approach 1: Generate Partial Bitstreams from Full Configurations

Using the Abstract Shell approach, you did not create multiple configurations as the standard flow uses, as each RM is implemented on its own, independent of the full static top. However, you can create any possible configuration by linking the original full static checkpoint with one RM checkpoint per RP.

1. Source this script to create all bit files. In the Tcl Console, make sure you are currently in the level above the `project_dfxc_vcu118` directory, where this script exists.

```
source create_all_bitstreams_via_configs_vcu118.tcl
```

This script first generates full and partial bitstreams from the impl_1 run exactly how it was done for Lab 7. Then, it assembles a "child_0" configuration from the count_down and shift_left RMs implemented within the Abstract Shells before generating their partial bitstreams.

The linking portion of the script looks like this:

```
create_project -in_memory -part $part
add_files ./project_dfxc_vcu118/project_dfxc_vcu118.runs/impl_1/
top_routed_bb.dcp
add_files ./abstract_shell/abs_shift_left/
rm_shift_left_route_design.dcp
set_property SCOPED_TO_CELLS {inst_shift} [get_files ./
abstract_shell/abs_shift_left/rm_shift_left_route_design.dcp]
add_files ./abstract_shell/abs_count_down/
count_down_route_design.dcp
set_property SCOPED_TO_CELLS {inst_count} [get_files ./
abstract_shell/abs_count_down/count_down_route_design.dcp
link_design -mode default -reconfig_partitions {u_shift u_count}
-part $part -top top
write_checkpoint -force abstract_shell/
config_shift_left_count_down_import/top_route_design.dcp
```

This configuration is effectively the same as Lab 7 produced for the child_0_impl_1 through the project flow. At this point, you have a full configuration from which you can run `write_bitstream` in the traditional manner using a non-project approach. This by default produces all full and partial bitstreams for this design image. You can use the `-no_partial_bitfile` or `-cell options` to create only full or only partial bit files, respectively. In this lab, you do not use the full design bitstream from the child implementation.

Approach 2: Generate Partial Bitstreams from Abstract Shells

When using Abstract Shells, complete static design information is not required for users to generate partial bitstreams. Each Abstract Shell contains all the information needed not only to implement any RM for that RP, but to create the partial bitstream for that function.

1. Source this script to create all bit files. In the Tcl Console, make sure you are currently in the level above the `project_dfxc_vcu118` directory, where this script exists.

```
source create_all_bitstreams_via_abs_vcu118.tcl
```

This script first generates full and partial bitstreams from the impl_1 run exactly how it was done for Lab 7. Then, it opens the count_down and shift_left RMs implemented within the Abstract Shells to generate their partial bitstreams.

IMPORTANT! *Generate partial bitstreams for Reconfigurable Modules from the design checkpoint that includes both the RM and the Abstract Shell in which it was implemented. The Abstract Shell contains critical information about the static design that must be included in a partial bitstream. See the figures in Step 4 for an illustration of this concept.*

Complete Hardware Validation

With all full and partial bitstreams generated, PROM file generation can be done. The bitstreams are named and located in the same way as was done in Lab 7, so this design testing can be completed in that lab. Return to Lab 7, Step 2, Instruction 25 to complete the hardware testing.

Lab 9 Conclusion

This concludes lab 9. In this lab, you:

- Revisited the UltraScale+ version of a design with the Dynamic Function eXchange (DFX) Controller.
- Implemented the parent configuration using the standard DFX project mode.
- Created Abstract Shells for each Reconfigurable Partition in the design.
- Implemented additional Reconfigurable Modules within these Abstract Shells.
- Generated partial bitstreams using two different methodologies.

Lab 10

DFX BDC Project Flow in IP Integrator for Zynq UltraScale+

Overview

Dynamic Function eXchange (DFX) in AMD FPGAs, SoCs, and adaptive SoCs introduces new design requirements compared to traditional solutions. These requirements include unique approaches to source and run management, as both bottom-up synthesis and multi-pass implementation are needed. Before 2021, only non-project Tcl-based and RTL project-based solutions have been available in the AMD Vivado™ tools. The Vivado tools 2021.1 release introduced an IP-centric project-based environment, which includes new capabilities for block designs and other aspects of IP integrator.

This tutorial summarizes the Vivado tool flow, from project creation to partial bitstream creation for AMD Zynq™ UltraScale+™ MPSoC and RFSoC targets using the Block Design Container feature in IP integrator. This fundamental flow can be used to apply to AMD Virtex™ UltraScale™, AMD Kintex™ UltraScale™, and AMD UltraScale+™ device targets as well. The next lab covers the equivalent solution for AMD Versal™ device targets.

Flow Summary

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

- Creating Block Design Containers (BDC) to identify hierarchy in a project.
- Defining BDCs as Reconfigurable Partitions within the design hierarchy.
- Populating a set of Reconfigurable Modules for each Reconfigurable Partition.
- Creating a set of top-level and module-level synthesis runs.
- Creating a set of related implementation runs.
- Managing dependencies as sources, constraints, or options are modified.

- Checking rules and results.
- Verifying configurations.
- Generating compatible sets of full and partial bitstreams.
- Delivering full and partial images to the Zynq UltraScale+ target device.

Tcl Commands

Like with most everything within the AMD Vivado™ IDE, the features and tasks for Dynamic Function eXchange you see are driven behind the scenes by Tcl commands. One of the key goals for DFX project support is to be able to work seamlessly between GUI and script and command line on the same project. You can examine the specific Tcl commands called by examining the AMD Vivado™ journal file for this project. This can be seen by selecting **File → Open Journal** File. These Tcl commands are not documented in this user guide at this point. Also supported is the ability to export project scripts using the **Write Project Tcl** option. Additional information for each command can be found using the –help option of each command.

DFX Project Tutorial within IP Integrator

Tutorial Requirements

The design supplied here specifically targets the ZCU102 development system. The entire design can be processed completely or partially via scripts. The key scripts found in the project archive are as follows.

- `run_all.tcl`: This script runs the entire flow from block design creation to bitstream generation and hardware export. This includes all the steps managed by the DFX Wizard. It calls multiple scripts, in the following order:

1. `create_top_bd.tcl` – This script creates the static block design and all IP for a completely flat design. One could run this script and then launch the implementation to generate a standard (non-DFX) implementation for the ZCU102
2. `create_rp1_bdc.tcl` – This script creates two levels of hierarchy and converts one to be a block design container.
3. `enable_dfx_bdc.tcl` – This script turns the standard BDC into a DFX BDC.
4. `create_rp1rm2.tcl` – This script creates a new RM for the existing DFX BDC.
5. `run_impl.tcl` – This script is the remainder of the flow. This script creates a top-level wrapper, adds design constraints, and walks through the DFX Wizard. It then generates all outputs for the design, from IP results to synthesis and implementation runs, then creates bitstreams and XSA files for hardware handoff.

This tutorial can be run completely via a script by sourcing the run_all.tcl script in an AMD Vivado™ Tcl shell. Each sub-section can also be run individually using the remaining Tcl scripts listed above. The tutorial as described will manually run through most steps to show what is happening throughout the flow. Sub-section scripts will be noted along the way. This tutorial does not show iterative interaction with the IP integrator design.

Vivado Hardware Design Flow

In this section, you will create and compile a complete hardware design through AMD Vivado™, using IP integrator and Block Design Containers.

Step 1: Create a Flat Design in Vivado IP Integrator

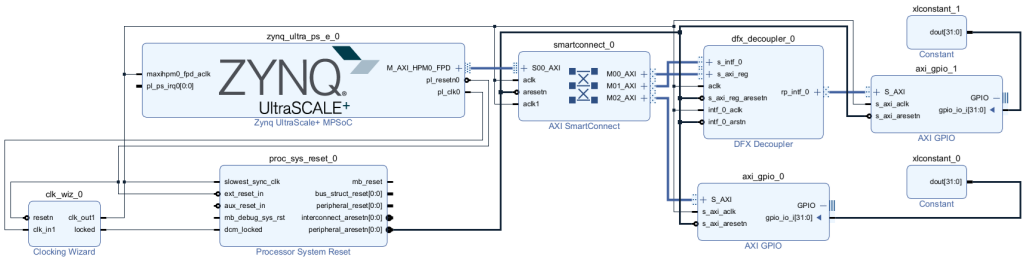
1. Open the AMD Vivado™ IDE. In the Tcl Console, navigate to the folder where the tutorial archive is unzipped. The source files and scripts for this tutorial reside in the `\ipi_bdc_dfx_zu` folder.
2. Source the first Tcl script to create a flat version of the design that targets the ZCU102.

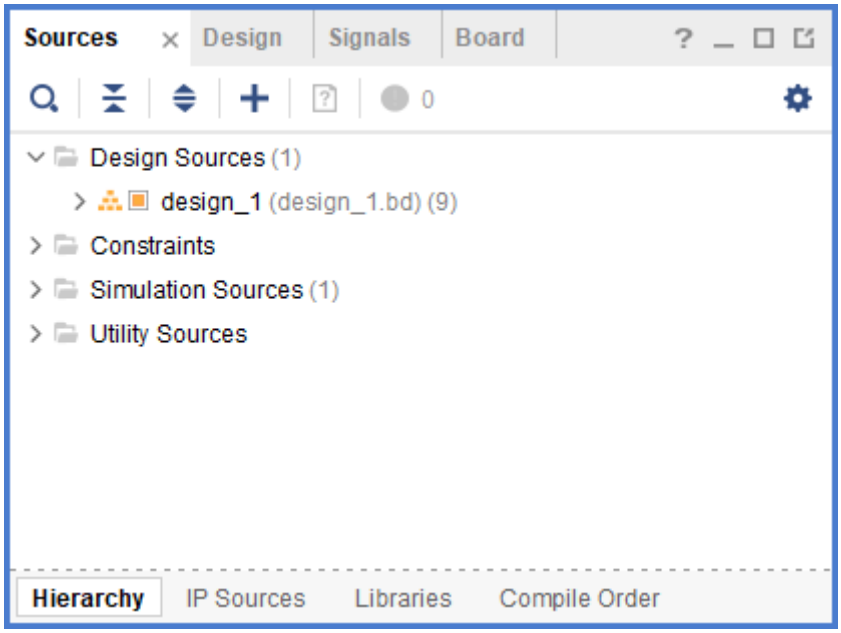
```
source create_top_bd.tcl
```

This script performs a few tasks:

- Creates a new project for a ZCU102 target
- Adds and customizes a collection of IP
- Connects the IP within the block design
- Validates and saves the block design

The create_top_bd.tcl script was generated from an existing block design by calling `write_bd_tcl -no_ip_version`. The only modification made to this script was to customize the project name from the default.





Step 2: Create Levels of Hierarchy in the Block Design

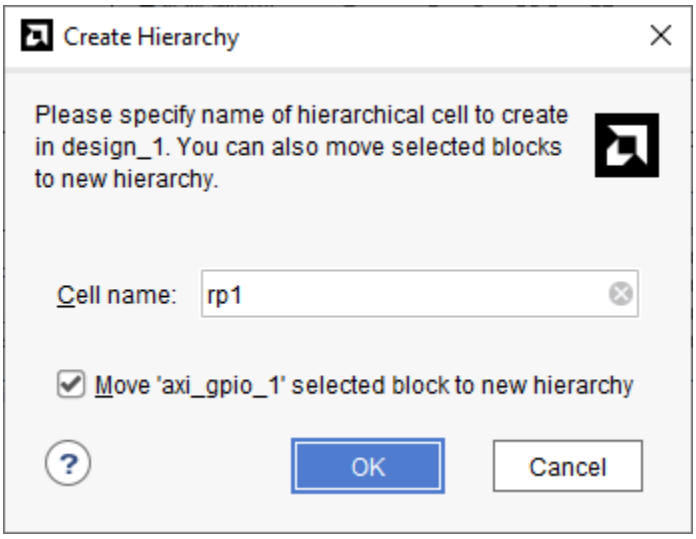
In this section, you will split the design into two hierarchical instances. The creation of a level of hierarchy for the static part of the design is not required; this is done simply to organize the design and focus attention on the dynamic region but could also be used for easier floorplanning for implementation if desired. The level of hierarchy for what will be the Reconfigurable Partition, however, is required, as this will be converted to a block design container.

Follow the instructions below or source `create_rp1_bdc.tcl` to automate the steps.

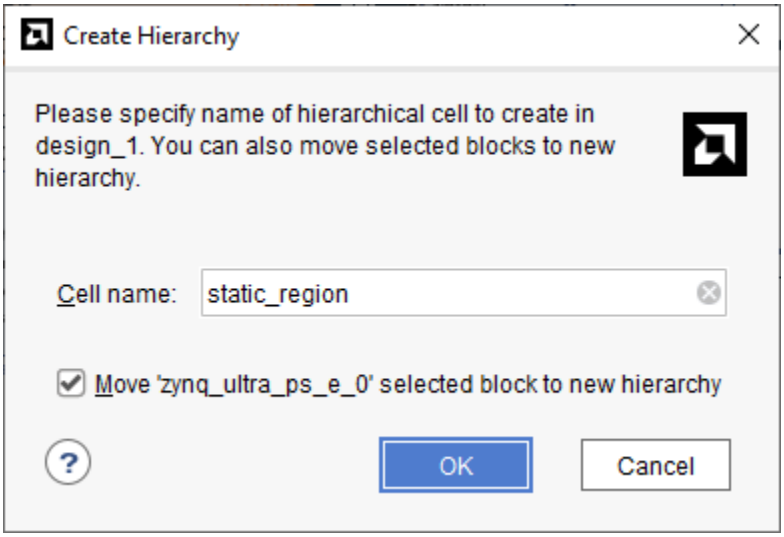
1. Right-click on the **axi_gpio_1** instance and select **Create Hierarchy**.

   ***Note*:** This is the GPIO IP connected to the DFX Decoupler IP.

2. In the resulting dialog box, name the hierarchy **rp1** and click **OK**.



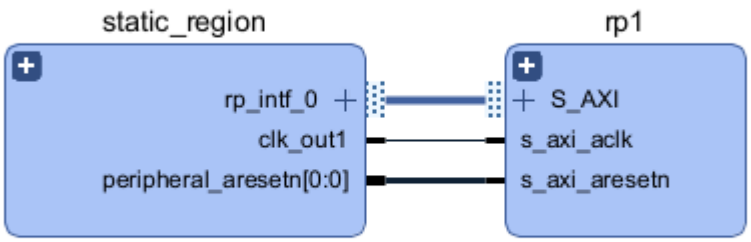
3. Select the **xlconstant_1** instance and drag and drop it into the **rp1 hierarchy** instance.
4. Right-click on the AMD Zynq™ UltraScale+™ MPSoC instance and select **Create Hierarchy**.
5. In the resulting dialog box, name the hierarchy **static_region** and click **OK**.



***Note*:** You will receive a message about losing .elf file association, but thus far, no .elf files have been associated with this processor. It is safe to ignore this warning.

6. One by one, select all remaining instances (other than rp1) and drag and drop them into the static_region level of hierarchy.

   The resulting block design can be cleaned up by running Regenerate Layout, and each level of the hierarchy can be collapsed. The resulting block design should look like this:

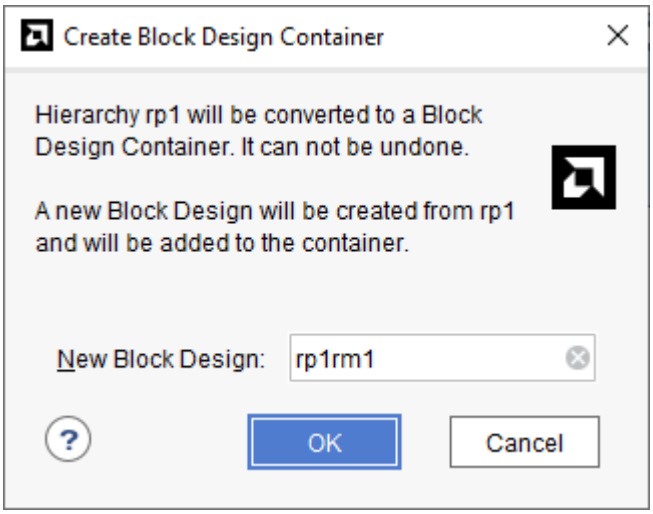


7. Right-click on the canvas to select Validate Design, then save the block design.

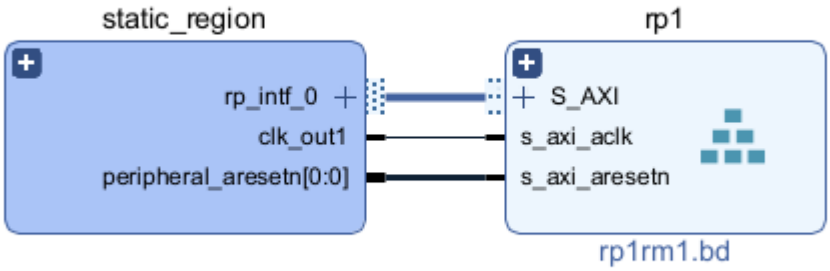
Step 3: Create a Block Design Container

Now that levels of hierarchy are established, the rp1 instance can be converted to a block design container, which will represent the Reconfigurable Partition.

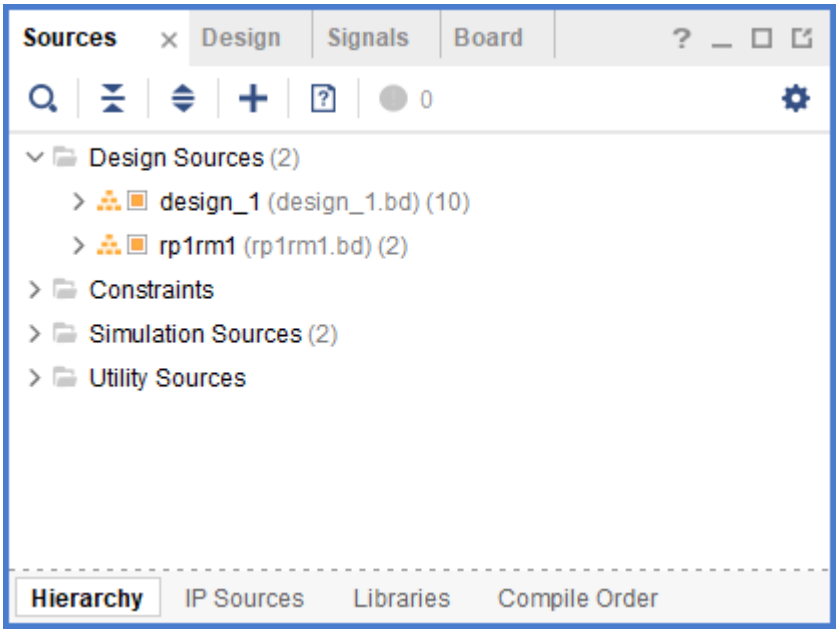
1. Right-click on the collapsed rp1 instance and select **Create Block Design Container**.
2. Name the container **rp1rm1** and click **OK**.



This converts the hierarchical instance into a block design container. The level of the hierarchy is labeled rp1rm1.bd and the block contains an icon that looks like a pyramid of six rectangles.

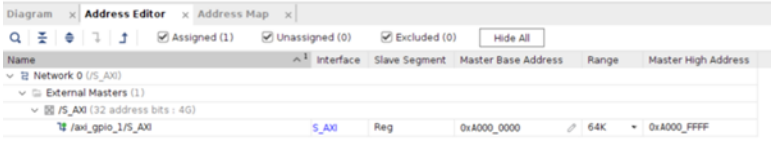


In the Sources window, a new block design is added to the project.



This action created a new block design for the rp1 submodule. If you expand the rp1 instance in the design_1 block design, you cannot edit the design at that level. This is a read-only copy, so to edit the design you must open the source `rp1rm1.bd` block design from the Sources view. This is not necessary now.

3. Modify the address information for the rp1rm1 instance by selecting the Address Editor window for the top-level block design. Completely expand the information for Network 0 to modify the range for /axi_gpio_1/S_AXI by changing it to **64K**.



4. Return to the diagram, right-click and select **Validate Design**. After validation completes, click **Save** to save the block design.

   The design as it stands now is still a standard IP integrator project but with two block designs instead of only one. The block design container feature in IP integrator allows you to add multiple design sources for the rp1 hierarchical instance, enabling changes through the use of multiple design revisions, or allowing for team design by sharing submodule block designs with team members.

Step 4: Enable Dynamic Function eXchange

In this section, you will enable the DFX capabilities within IP integrator and add new Reconfigurable Modules in the rp1 block design container.

Follow the instructions or source enable_dfx_bdc.tcl to automate the steps.

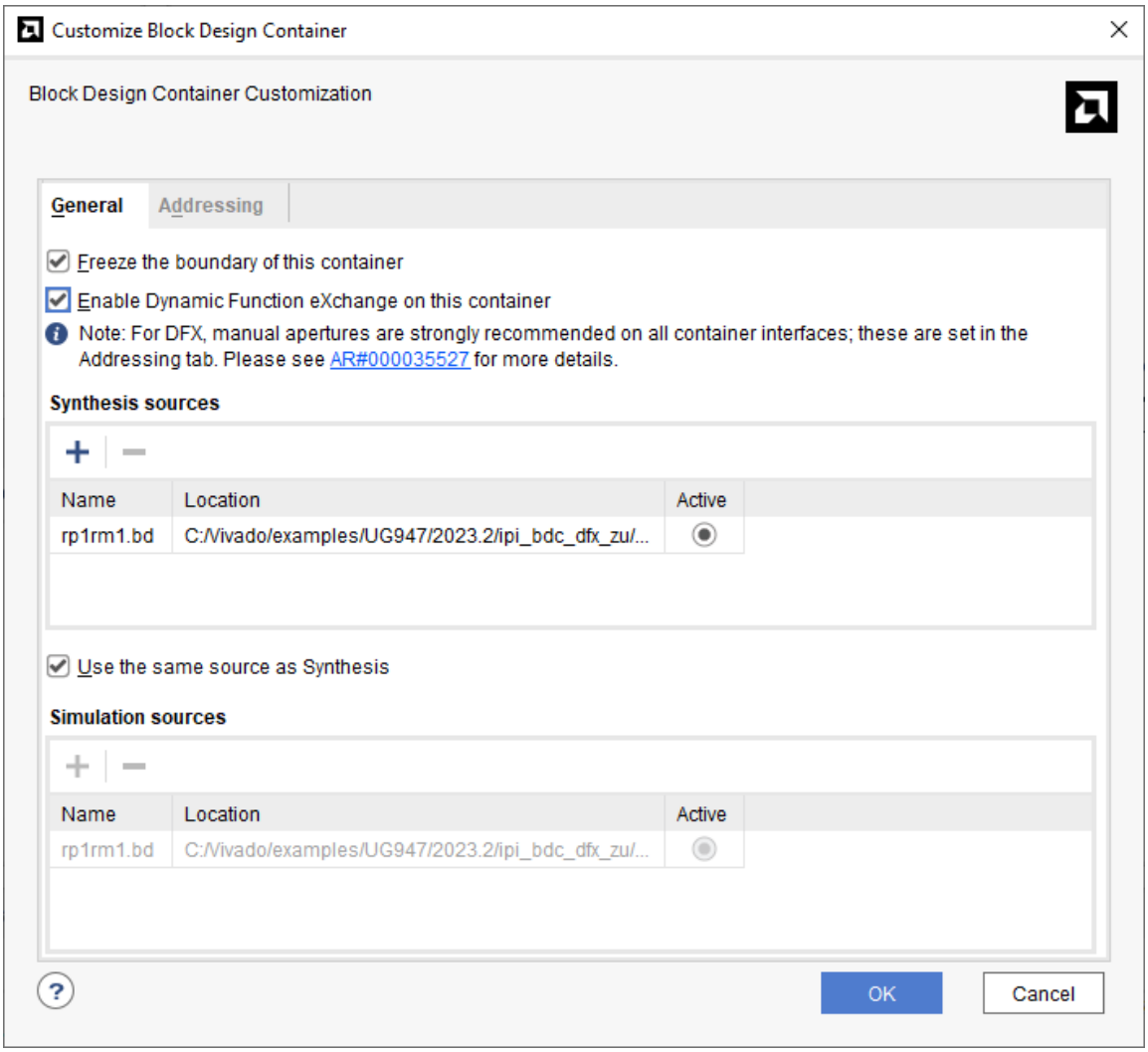
***Note*:** This action is irreversible. Once a project is converted to a DFX project, it cannot be changed back. The design runs infrastructure and all the DFX-centric settings are expected from this point forward, and DRCs are enabled to keep users on the correct path. It is recommended that designs be archived before this conversion to save a non-DFX version.

1. Select **Tools → Enable Dynamic Function eXchange** to expose DFX features within the AMD Vivado™ IDE. Select the **Convert** option in the dialog box that opens.

   Once this step has been run, you will see new menu items appear, most notably the **Dynamic Function eXchange Wizard** in the Flow Navigator and under the Tools menu.

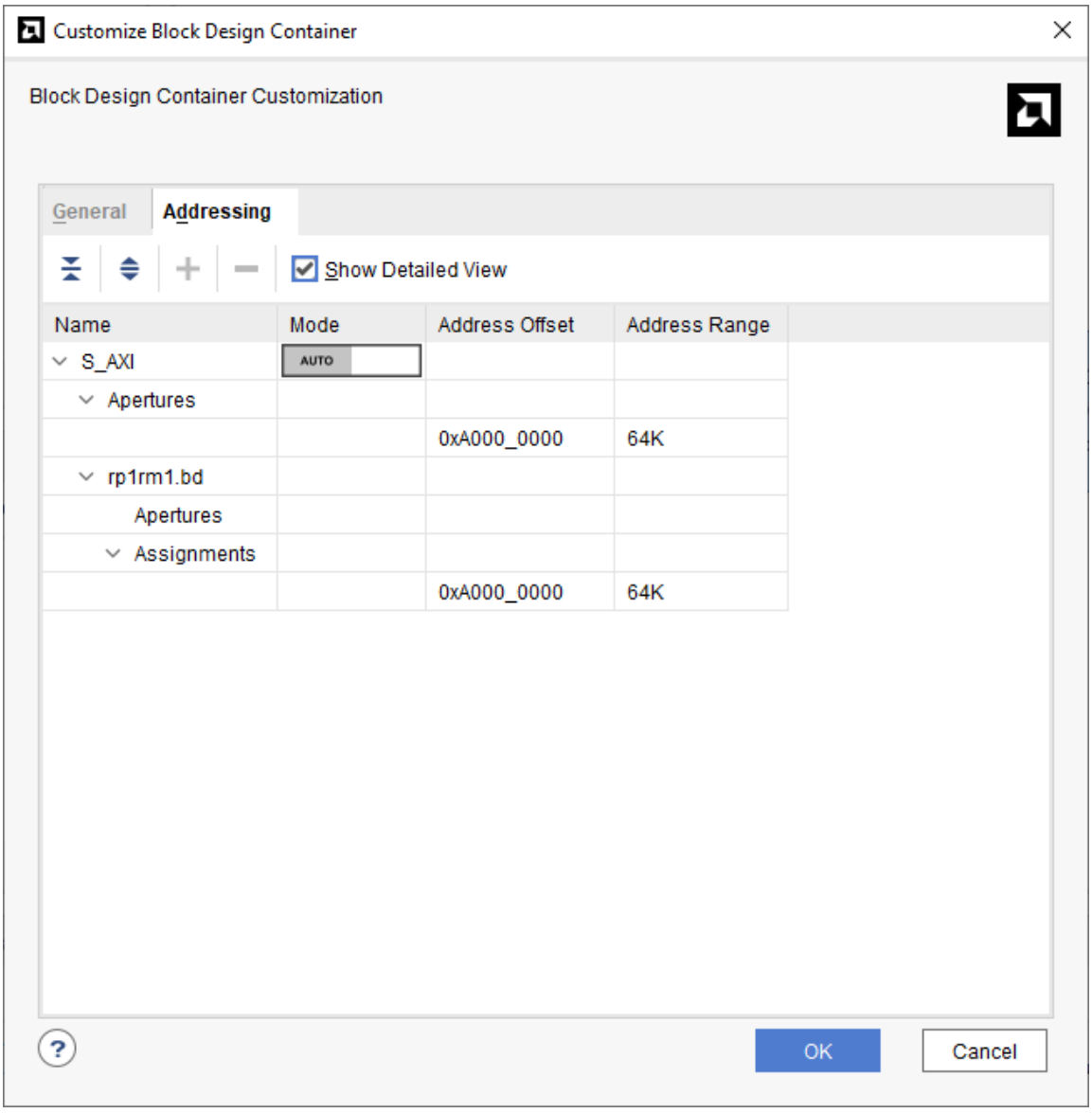
   ***Note*:** If the project is not explicitly converted by the user, it will be automatically done when the block design is generated later in the flow, based on the DFX setting on the block design container. Even if the conversion is automatic, it is still irreversible.

2. In the design_1 diagram, double click on the **rp1** instance to edit the block design container.
3. Under the General tab, check both the **Enable Dynamic Function eXchange on this container** and the **Freeze the boundary of this container** options.



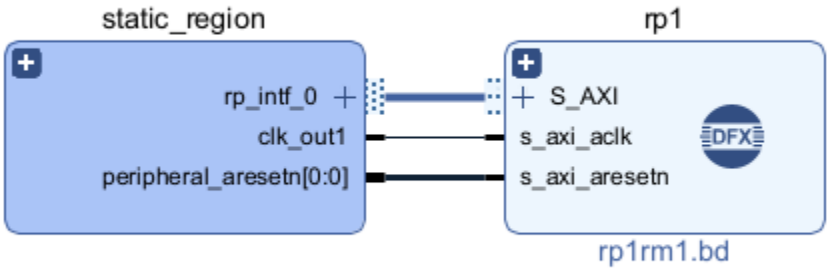
Checking the Enable Dynamic Function eXchange on this container option defines the rp1 instance to be a Reconfigurable Partition (RP). Freezing the boundary of this container prevents parameter propagation across the boundary interface.

4. Click the **Addressing** tab to see the aperture for this block design container. The **Address Offset** is 0xA000_000 and the **Address Range** is 64K, matching the information supplied in rp1rm1. Check the **Show Detailed View** to see that the aperture for rp1rm1 matches the general aperture for rp1 overall. No changes are necessary at this point; this tab will be revisited later.



5. Click **OK** to save the changes and return to the design_1 diagram.

   You will see that the icon on the rp1 block design container has changed to show a “DFX” label.



6. Click **Validate Design** then **Save** to save the design.

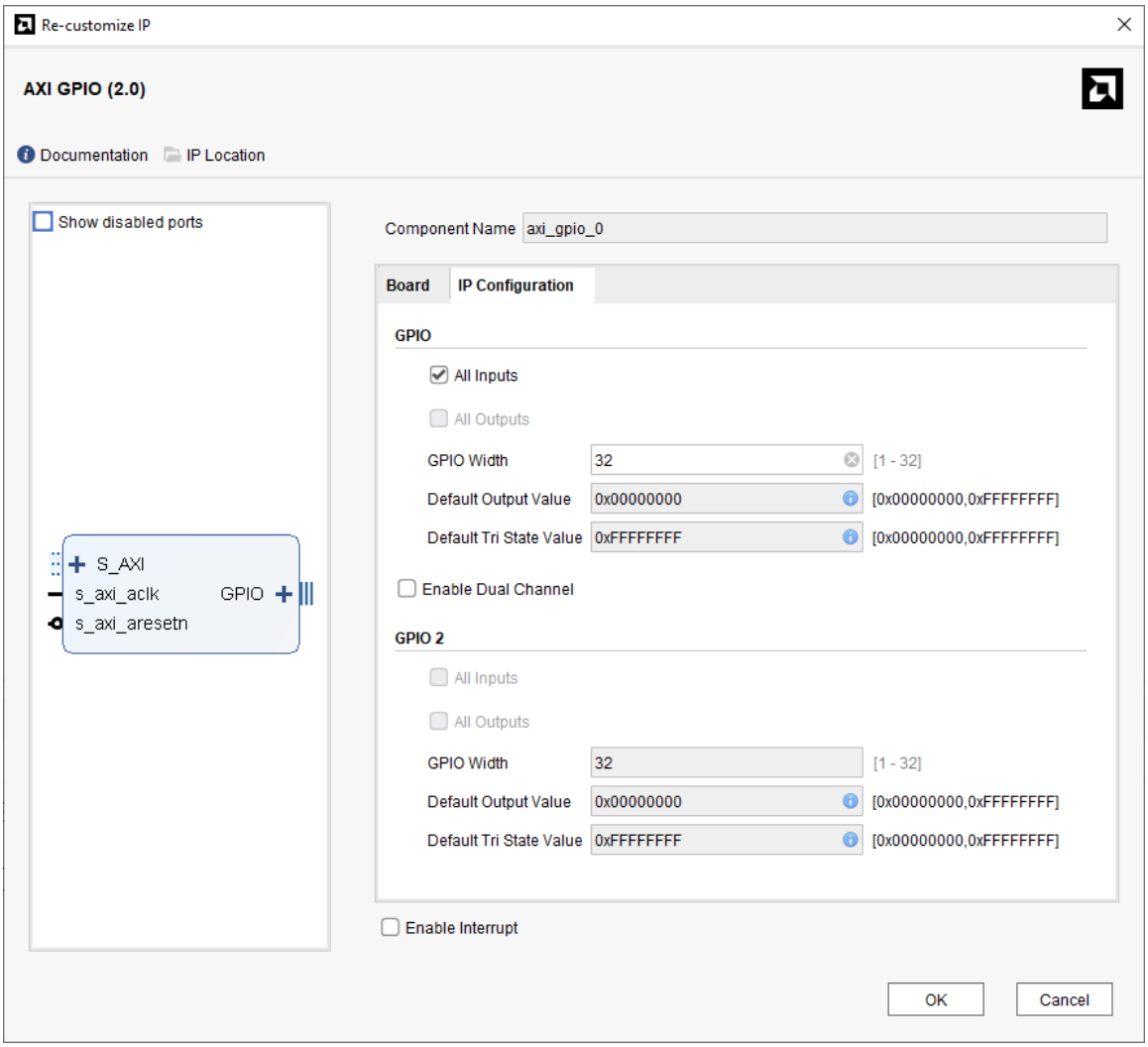
Step 5: Add a New Reconfigurable Module

Dynamic Function eXchange would not be very compelling without multiple Reconfigurable Modules (RM) to swap between, so the next step is to create a new RM for the RP that now exists. Follow the instructions below or source create_rp1rm2.tcl to automate the steps.

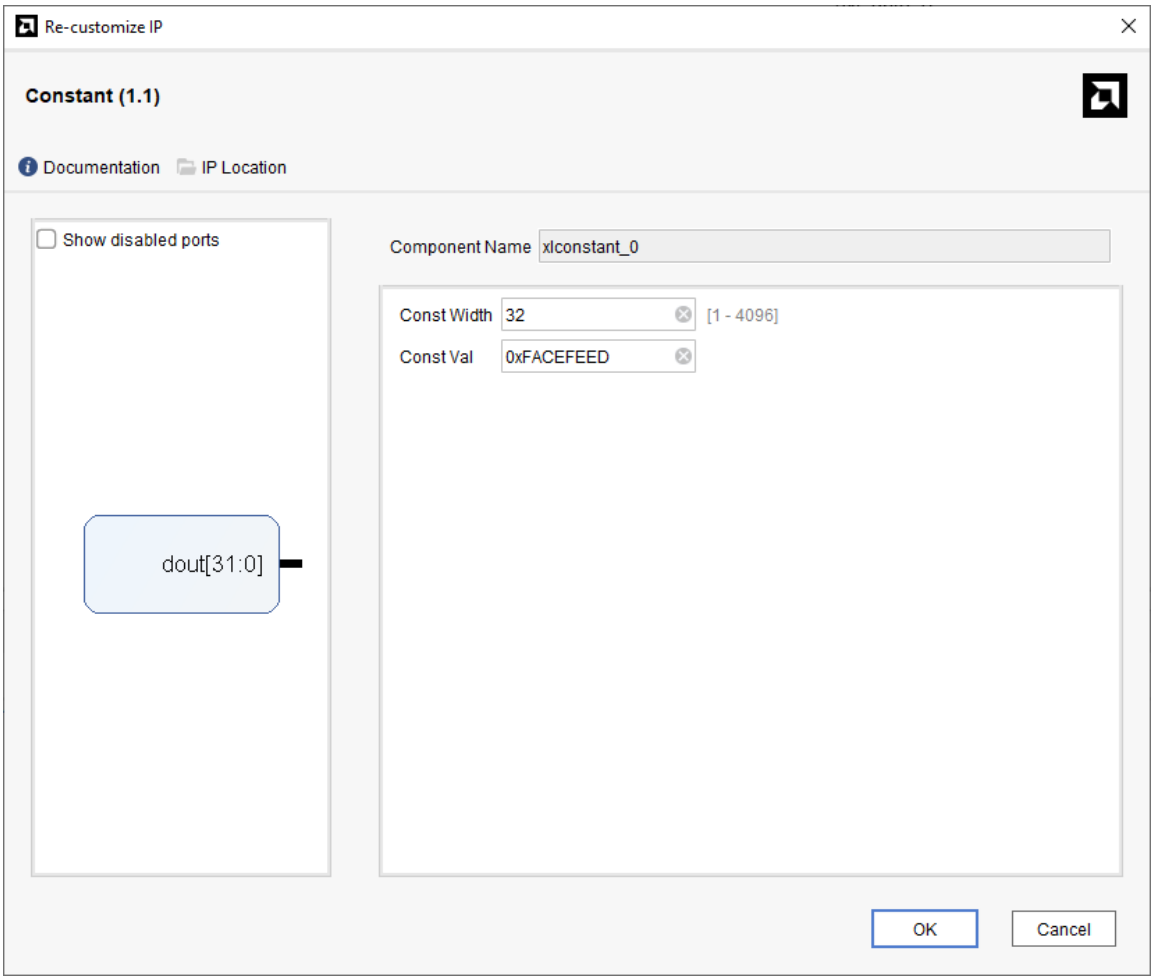
1. Right-click on the rp1 instance and select **Create Reconfigurable Module**. In the dialog box that opens, give the RM a name of **rp1rm2** and click **OK**.

   A new block design is created and opened. The diagram consists of three input pins, which are the same port list as the first RM for the rp1 partition. The port list for each RM for a given RP must be identical, even if not all of the ports are used by each RM. Note that in the log (and script) the `create_bd_design` command uses the `-boundary_from_container` option, copying the explicit port list from the block design container.

2. Add a new IP to the canvas by clicking the + icon and using the search field to find the AXI GPIO IP. Add it to the canvas, then double-click to customize. Check the **All Inputs** box for GPIO, ensure the **GPIO Width** is set to 32, then **OK** to return to the canvas.

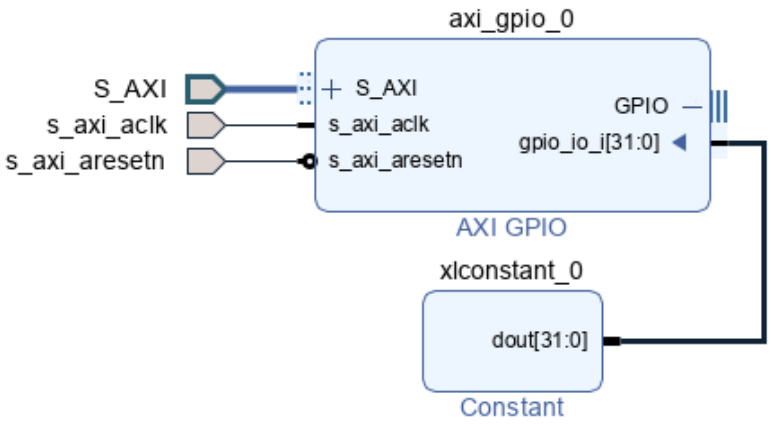


3. Click the + again and use the search field to add a **Constant** IP to the canvas. Double-click to customize. Change the **Const Width** to 32 and **Const Val** to 0xFACEFEED. Click **OK** to accept the edits.

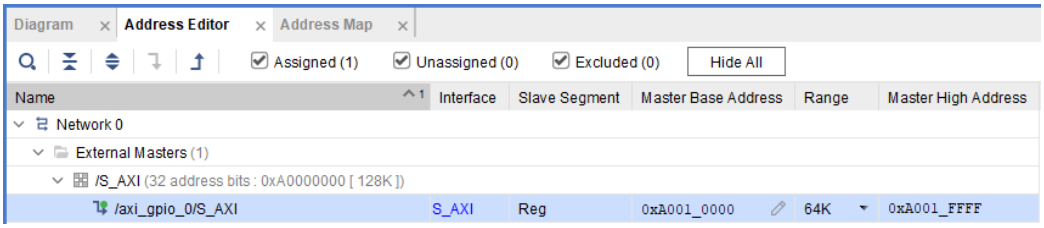


4. Connect the pins to create the diagram as shown in following figure.

   Note: You will need to expand the GPIO port to expose the 32-bit input bus to match the type of the Constant dout bus. Regenerate the layout to make it look nice.



5. Change to the Address Editor tab and note that no addresses have been assigned. Right-click on the row for /axi_gpio_0/S_AXI and select **Assign**. This sets a 64K range starting at address 0x4000_0000.
6. Modify the **Master Base Address** so it starts at 0xA001_0000 then keep the **Range** at 64K.



7. Validate and save the rp1rm2 block design.

   In this simple design, there are only two differences between rp1rm1 and rp1rm2:

   a. The S_AXI base addresses are different.

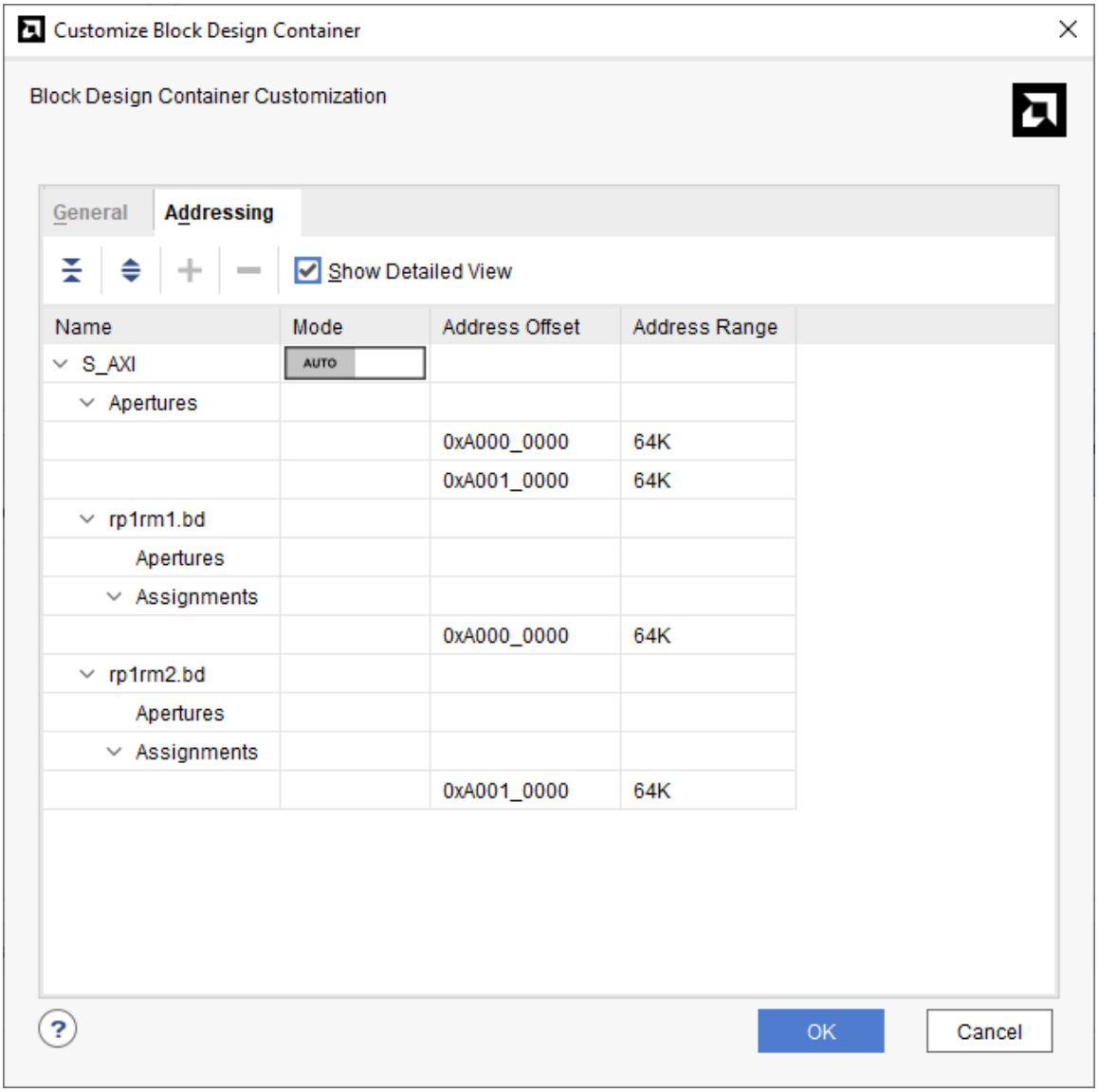
   b. The constant values that can be read via GPIO are different.

   This first difference will be used to show that device tree overlays must be created and managed for designs that might have different requirements between Reconfigurable Modules. The second difference will be used to confirm that dynamic reconfiguration in hardware has been done successfully.

Step 6: Confirm Apertures for All Reconfigurable Modules

Ensure that each RM has the appropriate aperture for its AXI slave interface, aligned to the instance in the top level. This is automatically done but can be manually set if desired.

1. In the top-level block design, double click on the rp1 block design container. Switch to the **Addressing** tab and click the **Show Detailed View** checkbox.



You can see that the overall aperture for rp1 for the S_AXI port starts at address 0xA000_0000 and has an overall range of 128K. This is automatically calculated by collecting address information from each design source in the block design container and summarizing each module's requirements.

If the aperture must be expanded to include new Reconfigurable Modules that have not been created yet, toggle the **Mode** from **Auto** to **Manual** and edit the master Offset or Range.

***Note*:** Adjustments to these values for individual block designs must be done at the source.bd.

2. Click **OK** to return to the top-level block design.
3. **Validate** and **save** design_1.bd.

Step 7: Create a Wrapper and Generate the Targets for the Top BD

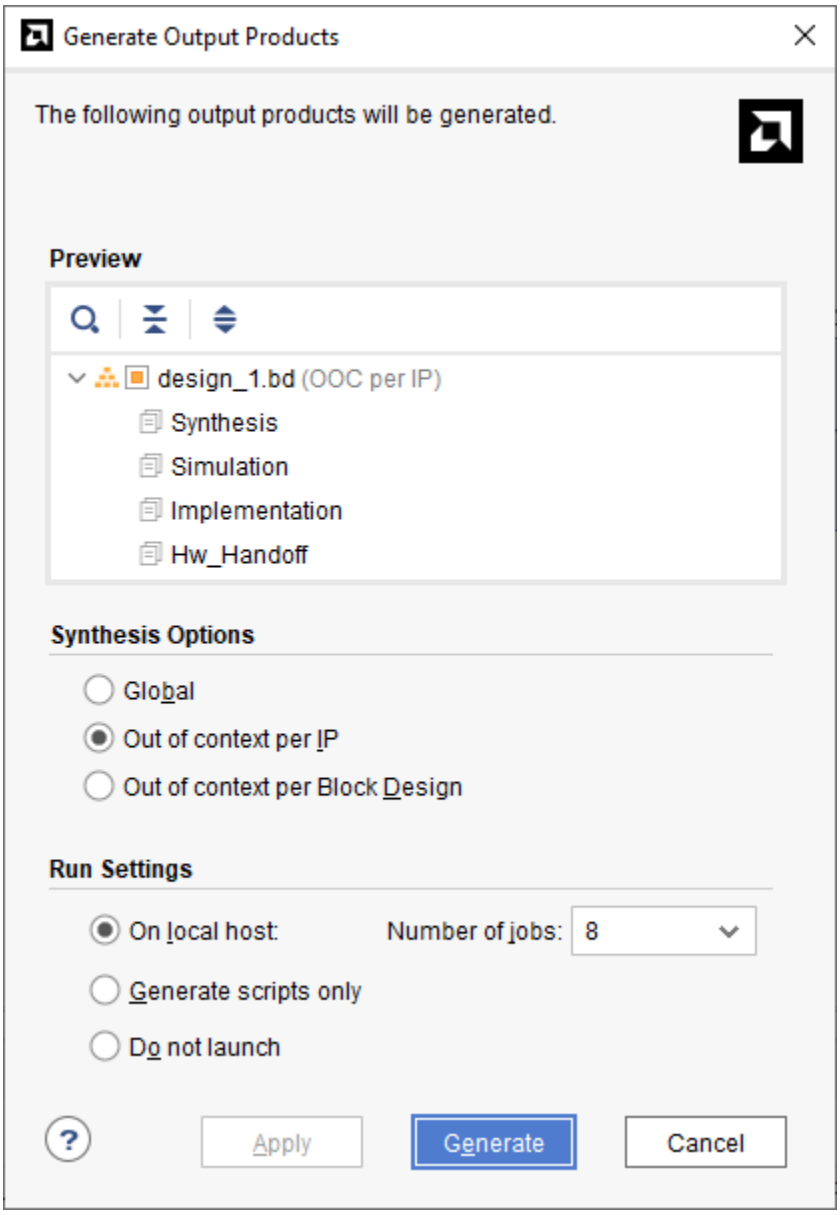
The final step before processing through synthesis and implementation is to create an HDL wrapper for the top-level block design, then generate targets for synthesis.

Follow the instructions below or source run_impl.tcl to automate the steps from here through the end of Section 1.

1. In the Sources window, right-click on design_1.bd and select **Create HDL Wrapper**. Keep the Let Vivado Manage option selected and click **OK**.

   In the Sources window, design_1_wrapper.v has been created and added to the project. This HDL file instantiates the design_1 block design.

2. In the Flow Navigator, click the **Generate Block Design** command under the IP INTEGRATOR header. In the resulting dialog box, keep the Out of context per IP option selected, then click **Generate**.



This action creates synthesizable output products for each IP in design_1, building out-of-context synthesis runs for each IP. Under the Design Runs window, you will see the list of synthesis runs for all the IP contained in design_1 (within the static_region hierarchy, so everything not included in the rp1 block container) have been created and are now running. The IP within the block container, for sources rp1rm1 and rp1rm2, have been created but are not running – this action will be requested later.

***Note*:** If the design has not been converted to a DFX design, this step will perform that conversion automatically. The definition of a block design container as a DFX partition enforces the need to be in DFX project mode, but nothing to this point has required it.

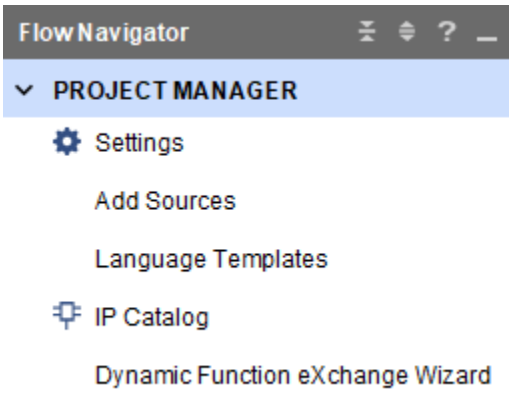
Step 8: Use the DFX Wizard to Define Configurations

The Dynamic Function eXchange Wizard is used to define relationships between the different parts of a DFX design. Using block design containers, you have created a level of the hierarchy of a design that can have more than one source in a DFX design. The block design container represents a Reconfigurable Partition (RP) and each source is a Reconfigurable Module (RM).

Within the DFX Wizard, you define configurations and configuration runs. A configuration is a full design image, with one RM per RP. A configuration run is a pass of the place and route tools to create a routed checkpoint for that configuration. The DFX Wizard also establishes parent-child relationships between configuration runs, helping automate required parts of the flow including static design locking and pr_verify, and sets up dependencies between runs, so AMD Vivado™ knows what steps must be rerun when sources are modified.

For more information on the DFX project flow, see this link in the *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909).

1. Open the DFX Wizard by clicking **Dynamic Function eXchange Wizard** in the Flow Navigator or by selecting that option under the Tools menu.



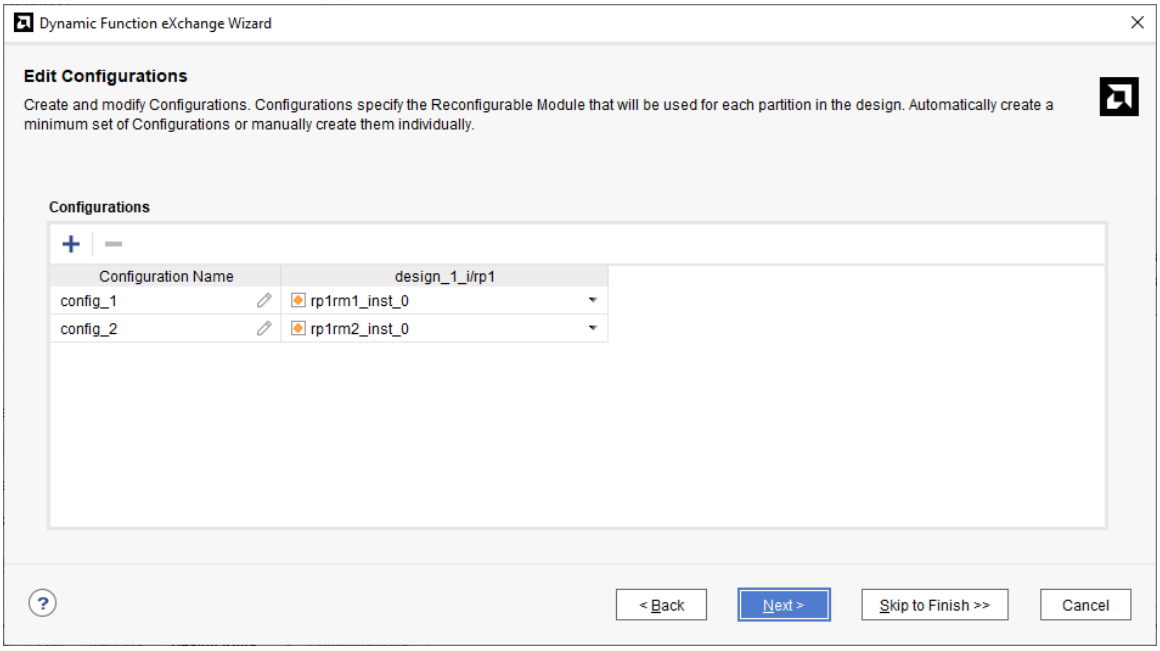
2. Click **Next**. In the Edit Reconfigurable Modules step, you will see the two RMs, rp1rm1_inst_0 and rp1rm2_inst_0, that you created within the rp1 block design container.

   IMPORTANT! *Unlike within the RTL Project flow for Dynamic Function eXchange, the DFX Wizard is NOT the entry point for new Reconfigurable Modules. New RMs should be added from the canvas in the same way RM2 was added.*

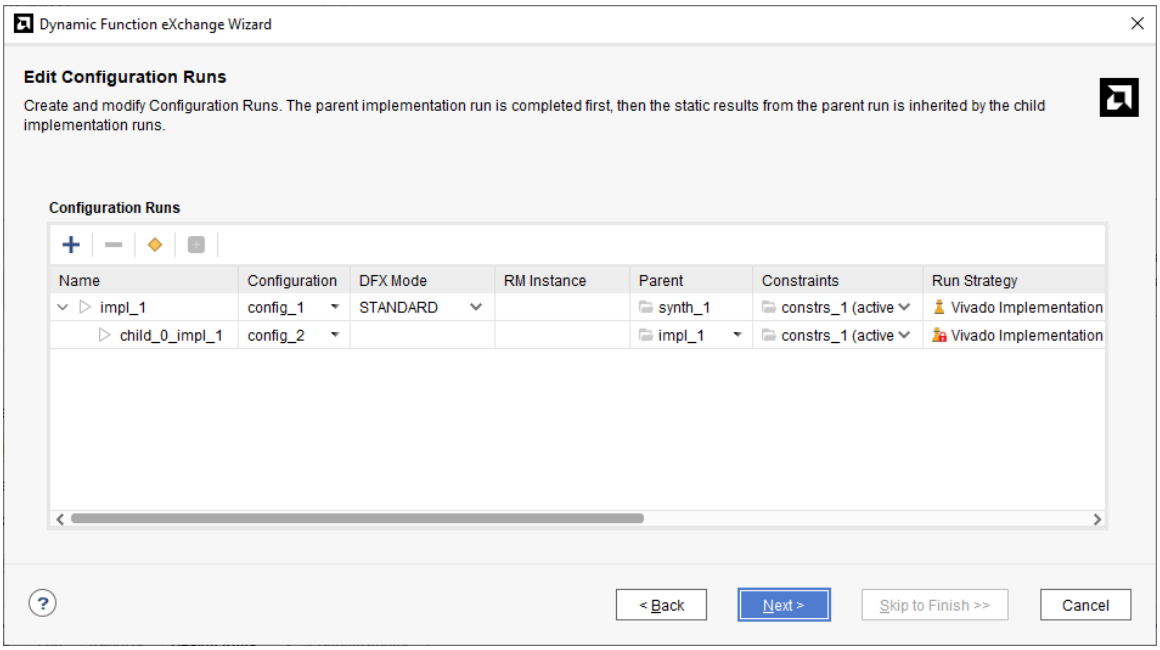
3. Click **Next**. In the Edit Configurations step, click the **automatically create configurations** link to generate two configurations.

   While you can also click the + button to generate these configurations, for designs with a single RP, automatic creation is the easiest way to create the list of configurations covering all RMs.

Note: The Configuration Name field is editable.



4. Click **Next**. In the Edit Configuration Runs step, click the **Standard DFX** link to create one run per configuration. For information on Abstract Shell configuration runs, see Lab 12: Abstract Shell Project Mode.



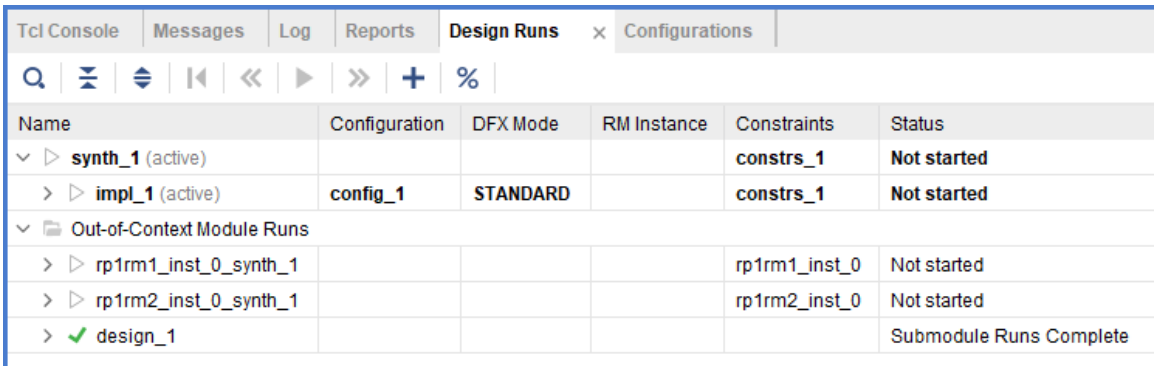
Note: The auto-generated names are not editable; the parentrun is impl_1 and the child run is child_0_impl_1. Like the Configurations themselves, you can manually create the Configuration Runs, and when doing so, you can name the runs anything you'd like.

A critical aspect of the DFX project flow is the parent-child relationship, and that is shown here under the Parent category. A parent implementation starts with a synthesis run, and all child runs must reference the parent implementation to establish the static design consistency between them. In this simple example, the parent of child_0_impl_1 is impl_1, and the indentation of the child run illustrates this relationship.

5. Click **Next** then **Finish** to complete this section.

   The DFX Wizard can be revisited any time to create new or modified configurations and configuration runs within the project.

   In the Design Runs window, a new implementation runs for child_0_impl_1 has been created. Like the view in the DFX Wizard, this new run is indented to show its dependency on the impl_1 run above it.



Step 9: Add Design Constraints for the Reconfigurable Partition

All DFX designs require a floorplan. Each Reconfigurable Partition requires a Pblock containing enough programmable resources to implement any Reconfigurable Module that can be inserted in that partition. In this tutorial, these Pblock constraints have been created for you.

1. In the Sources window, click the + to open the Add Sources dialog box. Select **Add or create constraints** then **Next**. Click **Add Files** and navigate to the tutorial directory to find `pblocks.xdc` in the constraints directory, then click **Finish** to add this constraint file to the project.

   If desired, you can open the post-synthesis design to view the Pblock created for this design. For more information on floor-planning requirements and methodology recommendations, see the *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909).

Step 10: Implement the Configurations and Generate Bitstreams

With all design sources now added to the project, and all settings complete for a DFX design, it is time to implement the design.

1. In the Design Runs window, right-click on child_0_impl_1 and select **Launch Runs**. Click **OK** to start the process.

   This action will launch all runs necessary to implement both parent and child configurations, in the proper order.

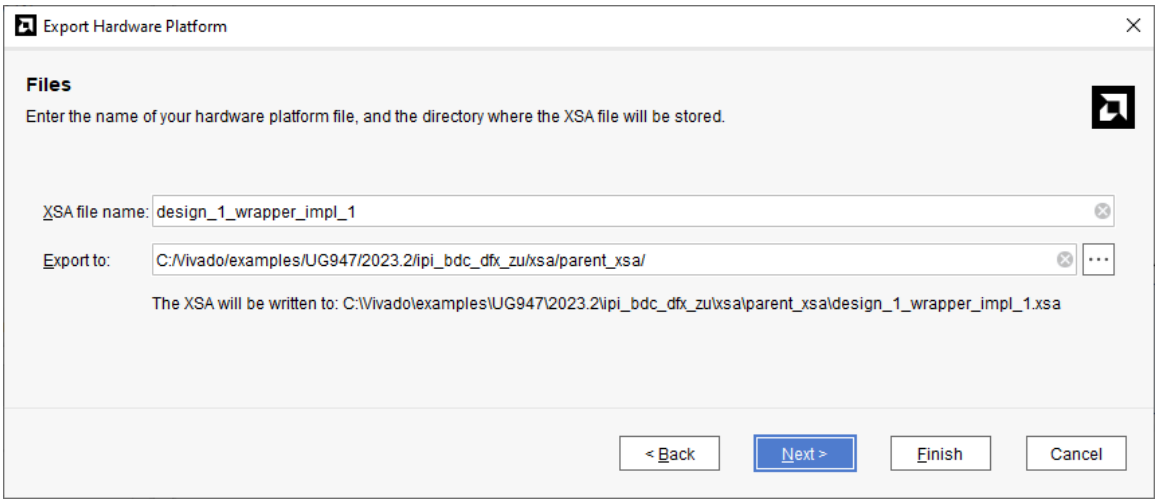
   - Out-of-context (OOC) synthesis will be run for the two RMs. These are launched in parallel as they do not depend on each other.
   - Synthesis of the top-level design launches with the OOC runs completely. This completes very quickly as the top level is nothing more than IO insertion plus two black boxes.
   - The parent-run is implemented first. This is a standard AMD Vivado™ implementation run that applies DFX constraints. At the end of the run, multiple design checkpoints are written:
     1. A standard placed and routed checkpoint for the full design.
     2. A module-level checkpoint for the RM rp1rm1 was also placed and routed.
     3. A static-only design checkpoint, with all placement and routing locked, and a black box for rp1
   - The child run is run last, and it starts with the locked static-only checkpoint from the parent-run.
2. Select **Cancel** on the resulting dialog box when implementation completes. Synthesized and implemented design checkpoints can be viewed at this point.
3. Shift-click in the Design Runs window to select both impl_1 and child_0_impl_1, then right-click to select **Generate Bitstream**. Click **OK** in the resulting dialog to continue.

This will create full device bitstreams for both configurations, and partial bitstreams for rm1 and rm2 Reconfigurable Modules.

Step 11: Export the Hardware Platform for Each Configuration

The final step in the hardware flow is to export the platform for PetaLinux. This fixed Xilinx Support Archive (XSA) platform contains the full device bitstream, the hardware handoff, and other files needed to build a PetaLinux system in . In this release of AMD Vivado™, these .xsa files do not include partial bitstreams. Expect this enhancement in a future version of Vivado.

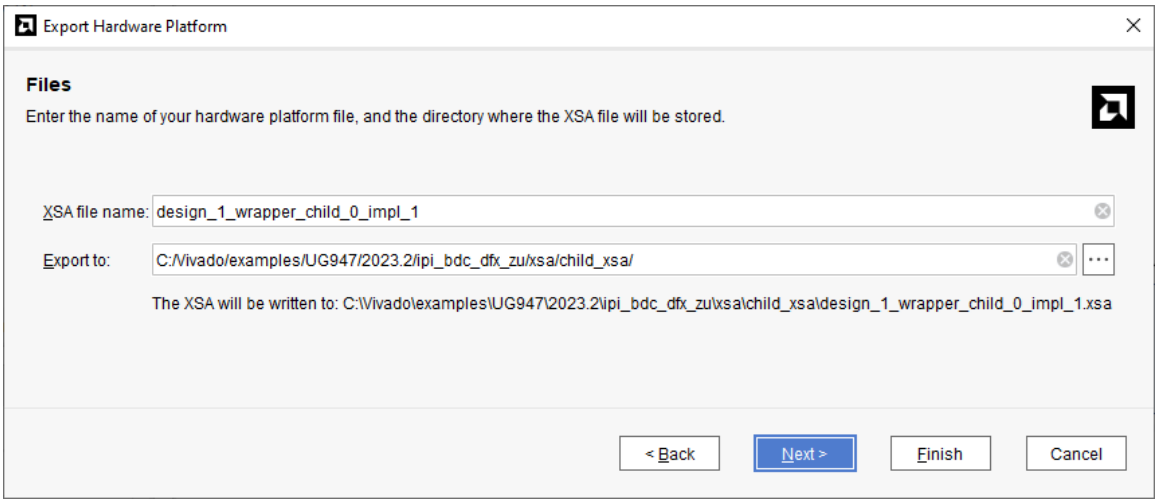
1. Select File > Export > Export Hardware.
2. Select the **Include bitstream** option and click **Next**.
3. Add "_impl_1" to the **XSA file name**, then change the **Export to** the directory to the xsa/parent_xsa folder beside the current project directory. Click **Next**, then **Finish** to write the XSA.



4. Open the child_0_impl_1 design run by right-clicking on that run and selecting Open Run.

   ***Note*:** Export Hardware defaults to the Active run, which by default is the parent impl_1 run. Exporting the hardware handoff XSA for any other configuration run requires that the desired implemented design be open in the IDE first.

5. Repeat steps 1 through 3 but change the XSA file name to design_1_wrapper_child_0_impl_1, and change the Export to field to the xsa/child_xsa folder.



At this point, the Vivado IP integrator for DFX tutorial is complete.

Lab 10 Conclusion

This concludes lab 10. In this lab, you:

- Created AMD Zynq™ AMD UltraScale+™ project using Block Design Containers in IP integrator.
- Converted this project to become a DFX project.
- Added a second Reconfigurable Module for the Reconfigurable Partition.
- Processed the design through synthesis, implementation, and bitstream generation.

Flow Summary

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

- Creating Block Design Containers (BDC) to identify hierarchy in a project.
- Defining BDCs as Reconfigurable Partitions within the design hierarchy.
- Populating a set of Reconfigurable Modules for each Reconfigurable Partition.
- Creating a set of top-level and module-level synthesis runs.
- Creating a set of related implementation runs.

- Managing dependencies as sources, constraints, or options are modified.
- Checking rules and results.
- Verifying configurations.
- Generating compatible sets of full and partial bitstreams.
- Delivering full and partial images to the Zynq UltraScale+ target device.

Lab 11

DFX BDC Project Flow in IP Integrator for Versal

Overview

Dynamic Function eXchange (DFX) in AMD FPGAs, SoCs, and adaptive SoCs introduces new design requirements compared to traditional solutions. These requirements include unique approaches to source and run management, as both bottom-up synthesis and multi-pass implementation are needed. Before 2021, only non-project Tcl-based and RTL project-based solutions have been available in the AMD Vivado™ tools. The Vivado tools 2021.1 release introduced an IP-centric project-based environment, which includes new capabilities for block designs and other aspects of IP integrator.

This tutorial summarizes the Vivado tool flow, from project creation to partial image creation for AMD Versal™ device targets using the Block Design Container feature in IP integrator. This fundamental flow can be used to apply to Virtex and AMD Kintex™ UltraScale™ and AMD UltraScale+™ device targets as well. The previous lab covers the equivalent solution for AMD Zynq™ UltraScale+™ device targets.

Flow Summary

The Dynamic Function eXchange IP integrator Project Flow inserts the key requirements of DFX into the existing Vivado project solution, accessible within the Vivado IDE as well as via Tcl commands. Coupled with the Vitis tools, a full hardware and software environment can be created and managed. Key requirements include:

- Creating Block Design Containers (BDC) to identify hierarchy in a project.
- Defining BDCs as Reconfigurable Partitions within the design hierarchy.
- Populating a set of Reconfigurable Modules for each Reconfigurable Partition.
- Creating a set of top-level and module-level synthesis runs.
- Creating a set of related implementation runs.

- Managing dependencies as sources, constraints, or options are modified.
- Checking rules and results.
- Verifying configurations.
- Generating compatible sets of full and partial programming images.
- Delivering full and partial images to the Versal adaptive SoC target device.

Tcl Commands

Like with most everything within the AMD Vivado™ IDE, the features and tasks for Dynamic Function eXchange you see are driven behind the scenes by Tcl commands. One of the key goals for DFX project support is to be able to work seamlessly between GUI and script and command line on the same project. You can examine the specific Tcl commands called by examining the AMD Vivado™ journal file for this project. This can be seen by selecting **File → Open Journal File**. These Tcl commands are not documented in this user guide at this point. Also supported is the ability to export project scripts using the **Write Project Tcl** option. Additional information for each command can be found using the `-help` option of each command.

DFX Project Tutorial within IP Integrator

Tutorial Requirements

The design supplied here specifically targets the VCK190 (and its Versal AI Core VC1902 device) development system. The entire design can be processed completely or partially via scripts. The VMK180 (and its Versal Prime VM1802 device) could also be targeted by changing the part and board references in the project creation script, but this design has not been tested in that hardware. The key scripts found in the project archive are as follows:

- `run_all.tcl`: This script runs the entire flow from block design creation to bitstream generation and hardware export. This includes all the steps managed by the DFX Wizard. It calls multiple scripts, in the following order:

1. `create_top_bd.tcl`: This script creates the static block design and all IP for a completely flat design. One could run this script and then launch the implementation to generate a standard (non-DFX) implementation for the VCK190
2. `create_rp1_bdc.tcl`: This script creates two levels of hierarchy and converts one to be a block design container.
3. `enable_dfx_bdc.tcl`: This script turns the standard BDC into a DFX BDC.
4. `create_rp1rm2.tcl`: This script creates a new RM for the existing DFX BDC.

5. `run_impl.tcl`: This script is the remainder of the flow. This script creates a top-level wrapper, adds design constraints, and walks through the DFX Wizard. It then generates all outputs for the design, from IP results to synthesis and implementation runs, then creates bitstreams and XSA files for hardware handoff.

This tutorial can be run completely via a script by sourcing the `run_all.tcl` script in a Vivado Tcl shell. Each sub-section can also be run individually using the remaining Tcl scripts listed above. The tutorial as described manually runs through most steps to show what is happening throughout the flow. Sub-section scripts are noted along the way. This tutorial does not show iterative interaction with the IP integrator design.

Vivado Hardware Design Flow

In this section, you create and compile a complete hardware design through AMD Vivado™, using IP integrator and Block Design Containers.

Step 1: Create a Flat Design in Vivado IP Integrator

1. Open the AMD Vivado™ IDE. In the Tcl Console, navigate to the folder where the tutorial archive has been unzipped. The source files and scripts for this tutorial reside in the `\ipi_bdc_dfx_versal` folder.
2. Source the first Tcl script to create a flat version of the design that targets the VCK190.

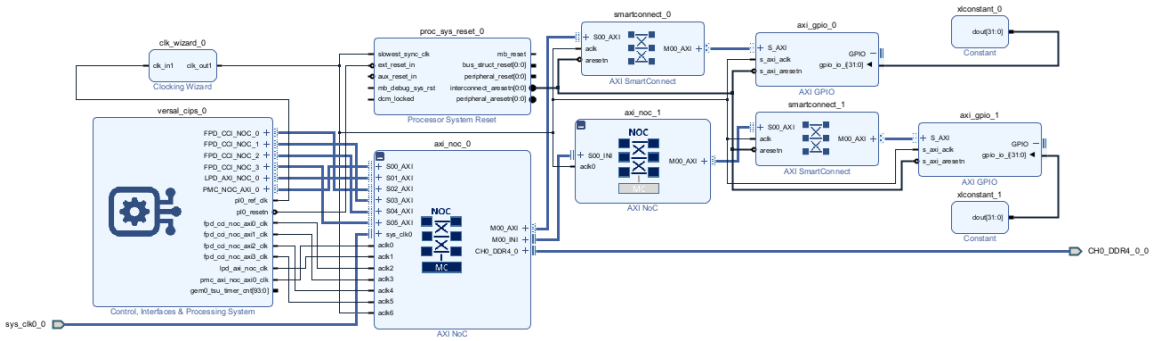
```
source create_top_bd.tcl
```

This script performs a few tasks:

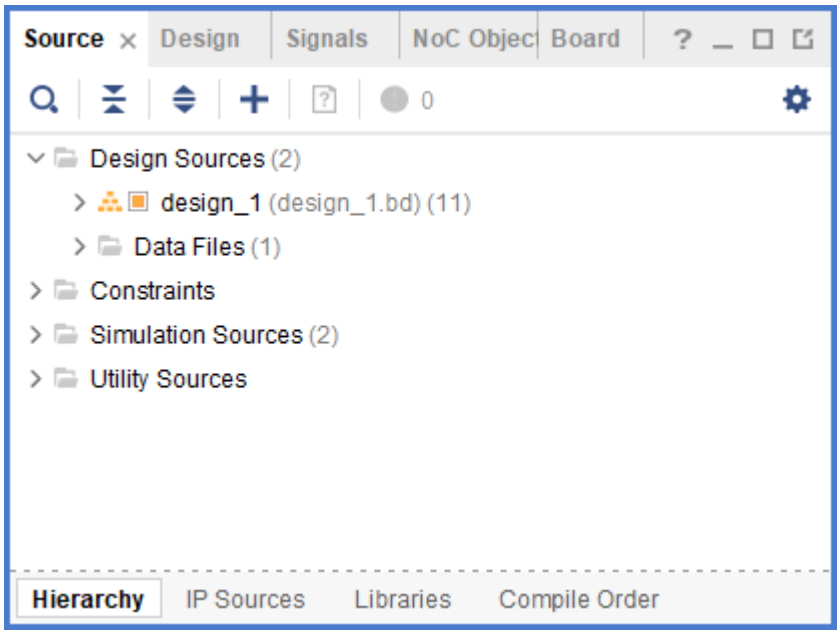
- Creates a new project for a VCK190 target
- Adds and customizes a collection of IP
- Connects the IP within the block design
- Validates and saves the block design

The create_top_bd.tcl script was generated from an existing block design by calling `write_bd_tcl -no_ip_version`. The only modification made to this script was to customize the project name from the default.

The following figure shows the initial block design.



The following figure shows the initial design hierarchy.



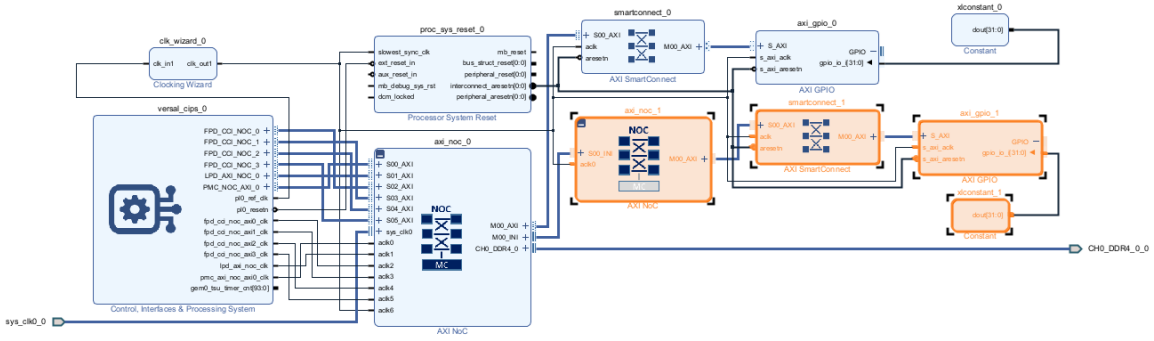
Step 2: Create Levels of Hierarchy in the Block Design

In this section, split the design into two hierarchical instances. The creation of a level of hierarchy for the static part of the design is not required. This is done simply to organize the design and focus attention on the dynamic region, but could also be used for easier floor planning for implementation if desired. The level of hierarchy for the Reconfigurable Partition is required, as it is converted to a block design container.

Follow the instructions below or source `create_rp1_bdc.tcl` to automate the steps.

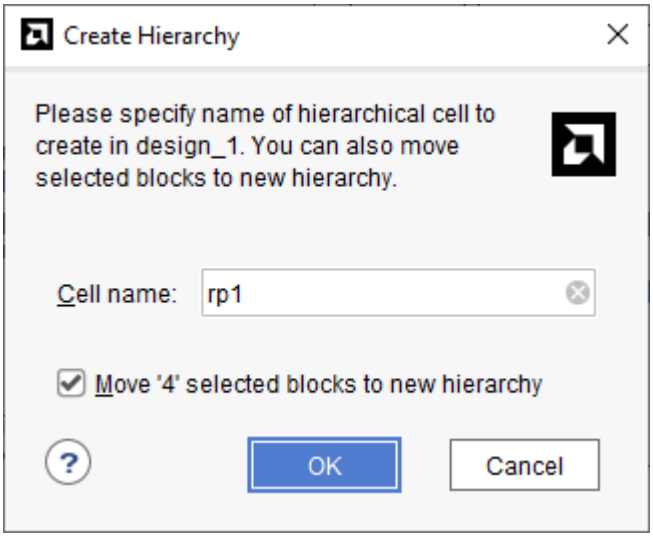
For the reconfigurable part of the design, these steps show how to create a level of hierarchy for all elements at once.

1. Ctrl-click to select these four instances in the block design: axi_noc_1, smartconnect_1, axi_gpio_1, and xlconstant_1. Then right-click on one of these four instances and select **Create Hierarchy**.

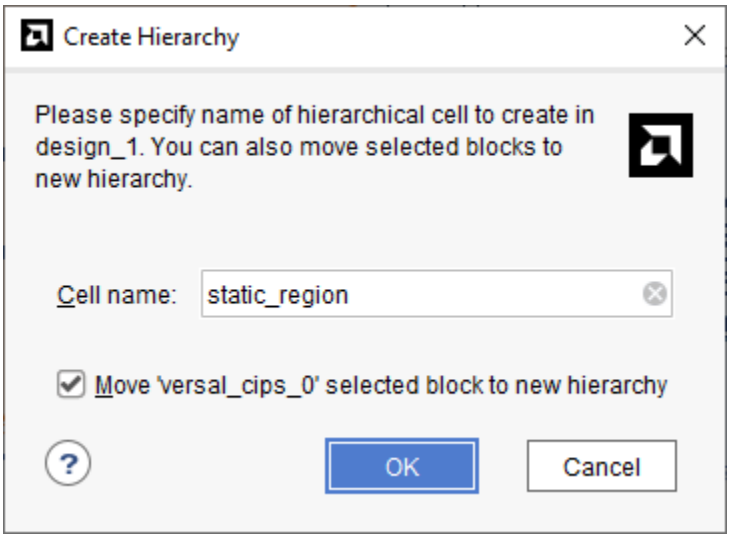


2. In the resulting dialog box, name the hierarchy **rp1** and click **OK**.

   Levels of hierarchy can also be built incrementally, adding IP one at a time. The hierarchical boundary is flexible until it is converted to a block design container.

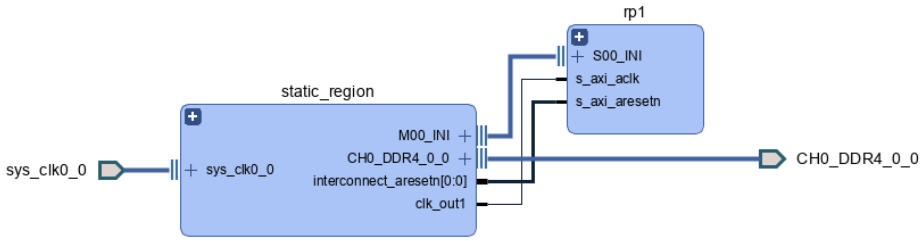


3. Right-click on the Versal CIPS instance and select **Create Hierarchy**.
4. In the resulting dialog box, name the hierarchy **static_region** and click **OK**.



5. One by one (or by using ctrl-click), select all remaining instances (other than rp1) and drag and drop them into the static_region level of hierarchy.

   The resulting block design can be cleaned up by running Regenerate Layout, and each level of the hierarchy can be collapsed. The resulting block design should look like this:

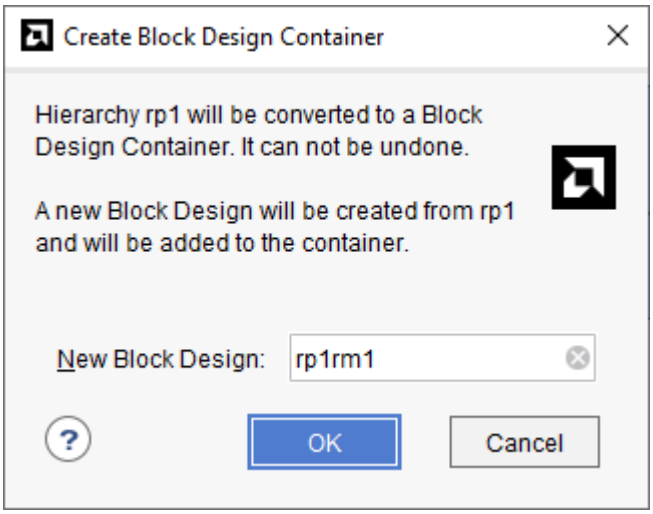


6. Right-click on the canvas to select Validate Design, then save the block design.

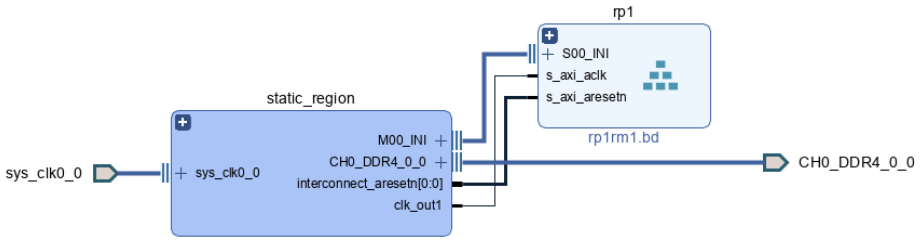
Step 3: Create a Block Design Container

Now that levels of hierarchy are established, the rp1 instance can be converted to a block design container, which represents the Reconfigurable Partition.

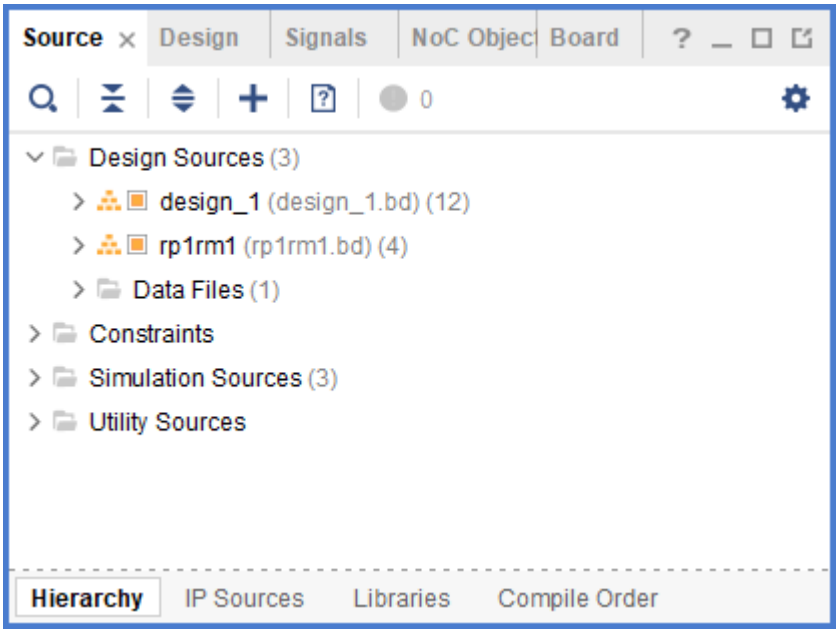
1. Right-click on the collapsed rp1 instance and select **Create Block Design Container**.
2. Name the container **rp1rm1** and click **OK**.



This converts the hierarchical instance into a block design container. The level of the hierarchy is labeled rp1rm1.bd and the block contains an icon that looks like a pyramid of six rectangles.

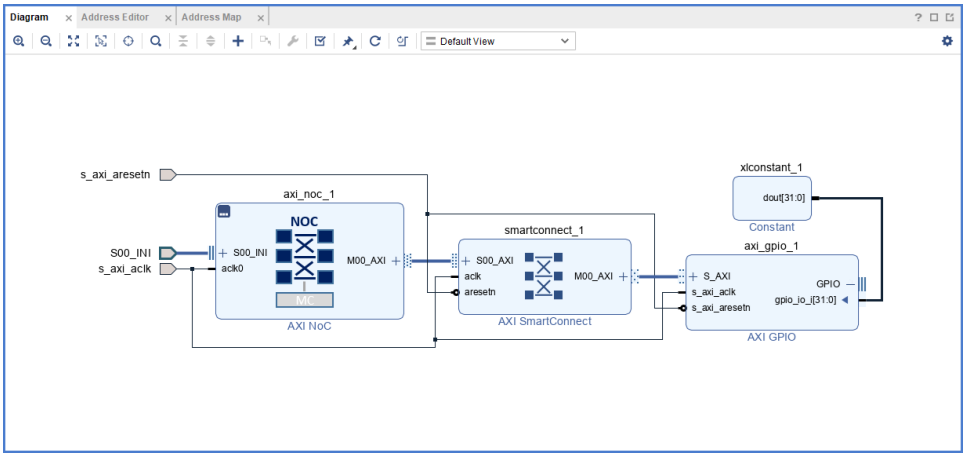


In the Sources window, you see a new block design has been added to the project.

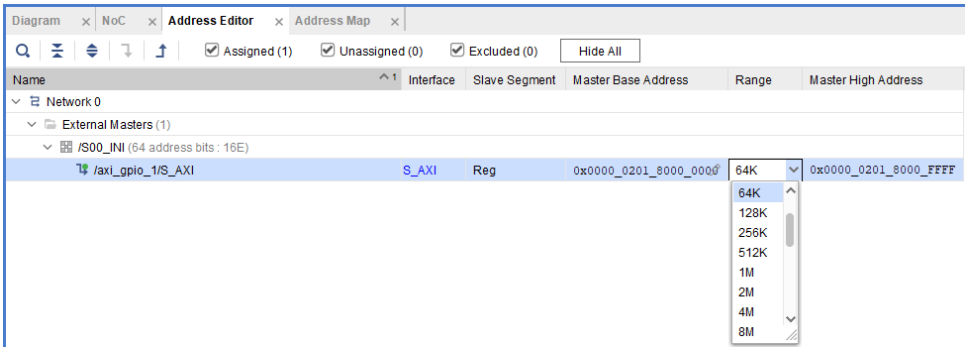


This action has created a new block design for the rp1 submodule. If you expand the rp1 instance in the design_1 block design, you see that you cannot edit the design at that level. This is a read-only copy, so to edit the design you must open the source rp1rm1.bd block design from the Sources window as shown in the previous figure.

3. Double click on the rp1rm1 block design in the Sources window to open the block design.



4. Select the **Address Editor** window and ensure the range for /axi_gpio_1/S_AXI is **64K**.



5. If any changes are made, return to the diagram, right-click and select **Validate Design**. After validation completes, click **Save** to save the block design.

   The design as it stands now is still a standard IP integrator project but with two block designs instead of one. The block design container feature in IP integrator allows you to add multiple design sources for the rp1 hierarchical instance, enabling changes through the use of multiple design revisions, or allowing for team design by sharing submodule block designs with team members.

Step 4: Enable Dynamic Function eXchange

In this section, enable the DFX capabilities within IP integrator and add new Reconfigurable Modules in the rp1 block design container.

Follow the instructions or source enable_dfx_bdc.tcl to automate the steps.

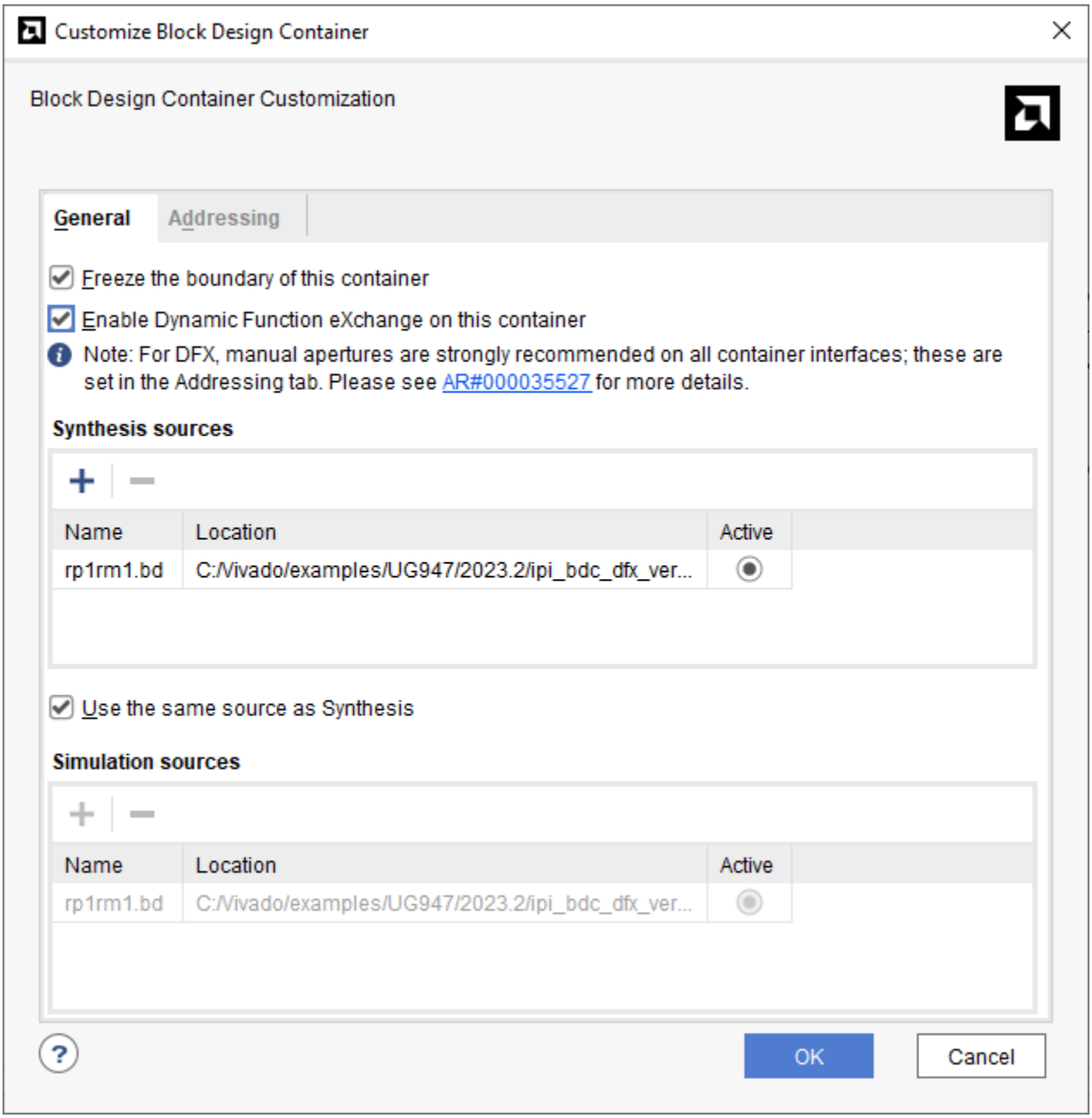
***Note*:** This action is irreversible. Once a project is converted to a DFX project, it cannot be changed back. The design runs infrastructure and all the DFX-centric settings are expected from this point forward, and DRCs are enabled to keep users on the correct path. It is recommended that designs be archived before this conversion to save a non-DFX version.

1. Select **Tools → Enable Dynamic Function eXchange** to expose DFX features within the AMD Vivado™ IDE. Select the **Convert** option in the dialog box that opens.

   Once this step has been run, a new menu items appear, most notably the **Dynamic Function eXchange Wizard** in the Flow Navigator and under the Tools menu.

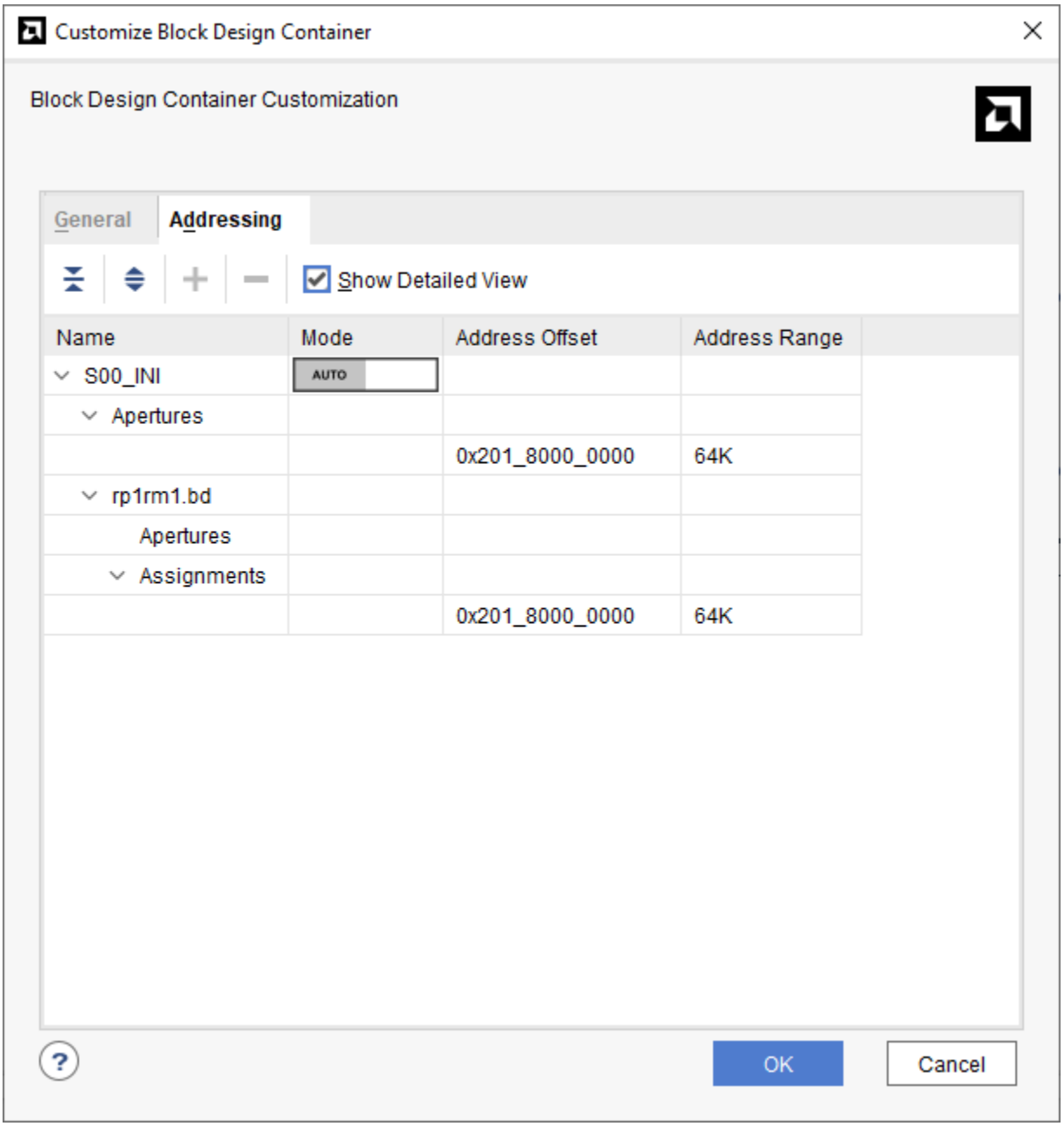
   ***Note*:** If the project is not explicitly converted by the user, it automatically does when the block design is generated later in the flow, based on the DFX setting on the block design container. Even if the conversion is automatic, it is still irreversible.

2. In the design_1 diagram, double click on the **rp1** instance to edit the block design container.
3. Under the General tab, check both the **Enable Dynamic Function eXchange on this container** and the **Freeze the boundary of this container** options.



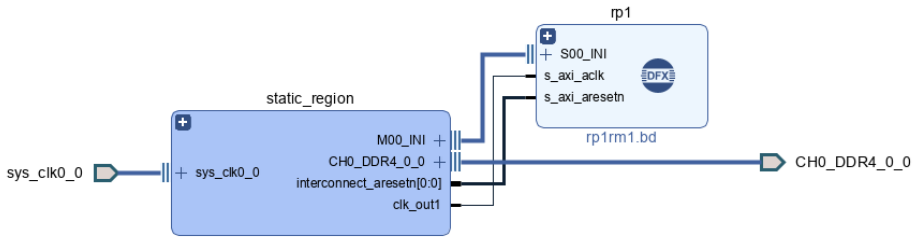
Checking the Enable Dynamic Function eXchange on this container option defines the rp1 instance to be a Reconfigurable Partition (RP). Freezing the boundary of this container prevents parameter propagation across the boundary interface.

4. Click the Addressing tab to see the aperture for this block design container. The **Address Offset** is 0x201_8000_0000 and the **Address Range** is 64K, matching the information supplied in rp1rm1. Check the **Show Detailed View** to see that the aperture for rp1rm1 matches the general aperture for rp1 overall. No changes are necessary at this point. More details on the Addressing tab are covered in Step 6: Confirm Apertures for All Reconfigurable Modules.



5. Click **OK** to save the changes and return to the design_1 diagram.

   You will see that the icon on the rp1 block design container has changed to show a “DFX” label.



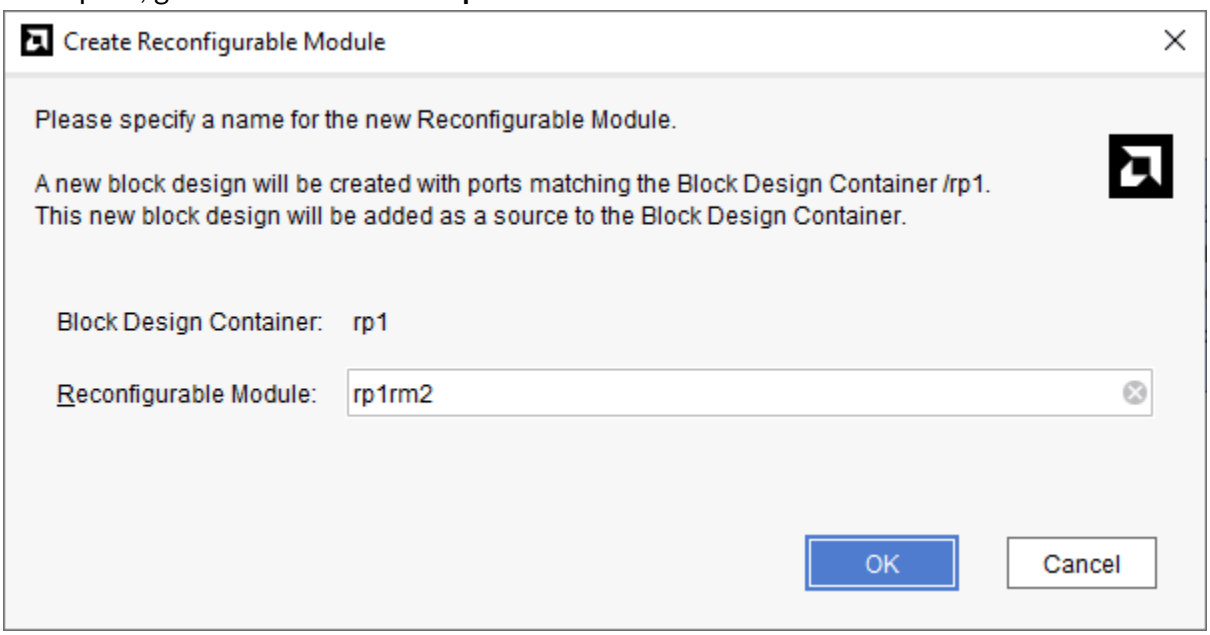
6. Click **Validate Design** then **Save** to save the design.

Step 5: Add a New Reconfigurable Module

Dynamic Function eXchange would not be very compelling without multiple Reconfigurable Modules (RM) to swap between, so the next step is to create a new RM for the RP that now exists.

Follow the instructions below or source create_rp1rm2.tcl to automate the steps.

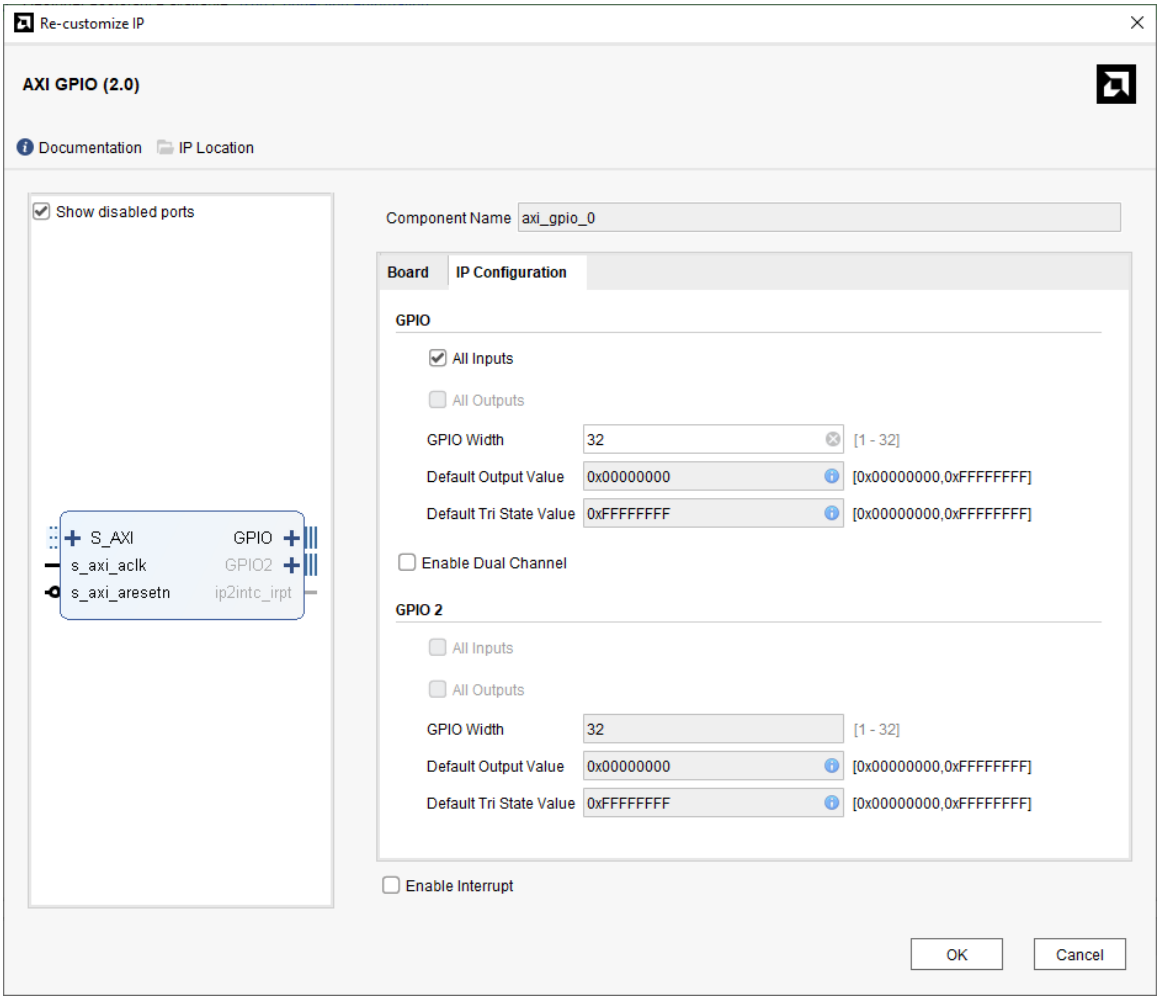
1. Right-click on the rp1 instance and select **Create Reconfigurable Module**. In the dialog box that opens, give the RM a name of **rp1rm2** and click **OK**.



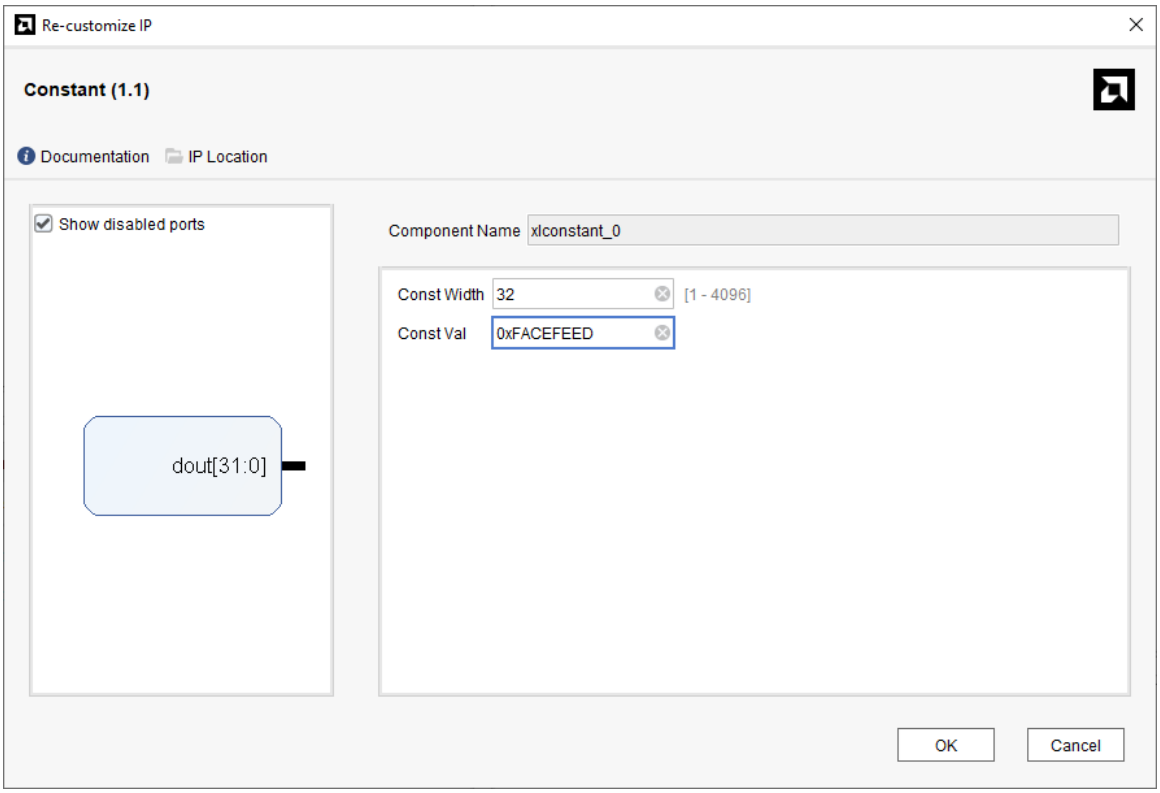
A new block design is created and opened. The diagram consists of three input pins, which are the same port list as the first RM for the rp1 partition. The port list for each RM for a given RP must be identical, even if not all of the ports are used by each RM.

Note: In the log and script the `create_bd_design` command uses the `-boundary_from_container` option, copying the explicit port list from the block design container.

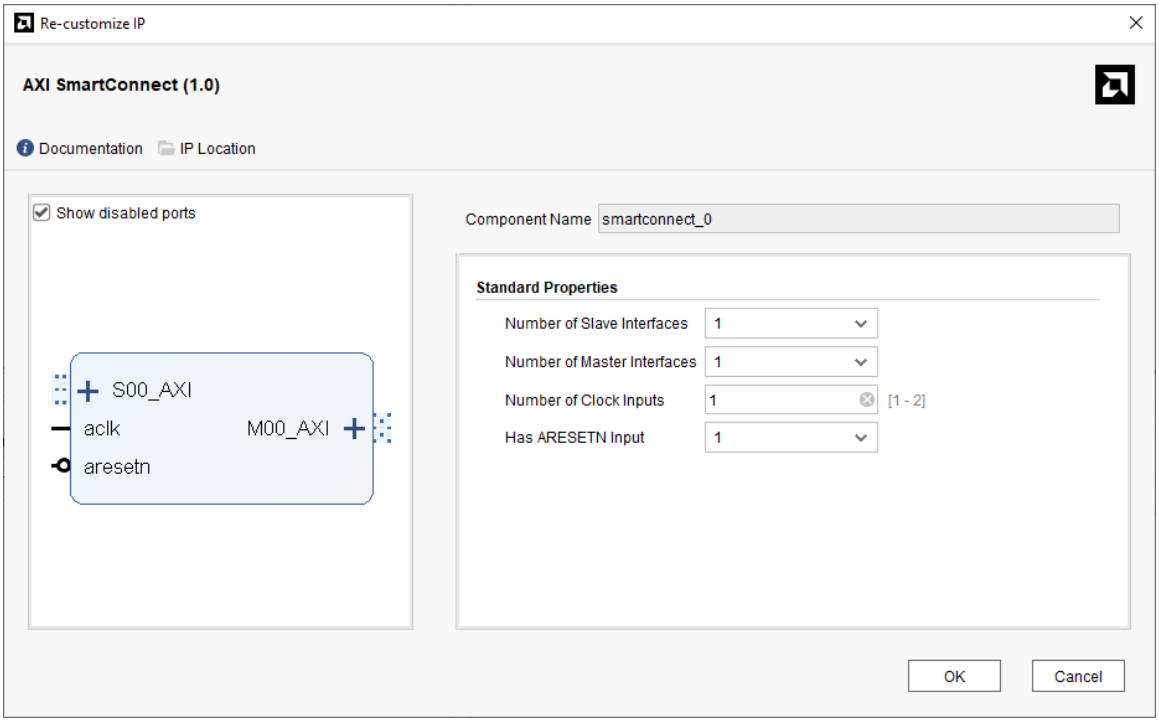
2. Add a new IP to the canvas by clicking the + icon and using the search field to find the AXI GPIO IP. Add it to the canvas, then double-click to customize. Check the **All Inputs** box for GPIO, ensure the **GPIO Width** is set to 32, then **OK** to return to the canvas.



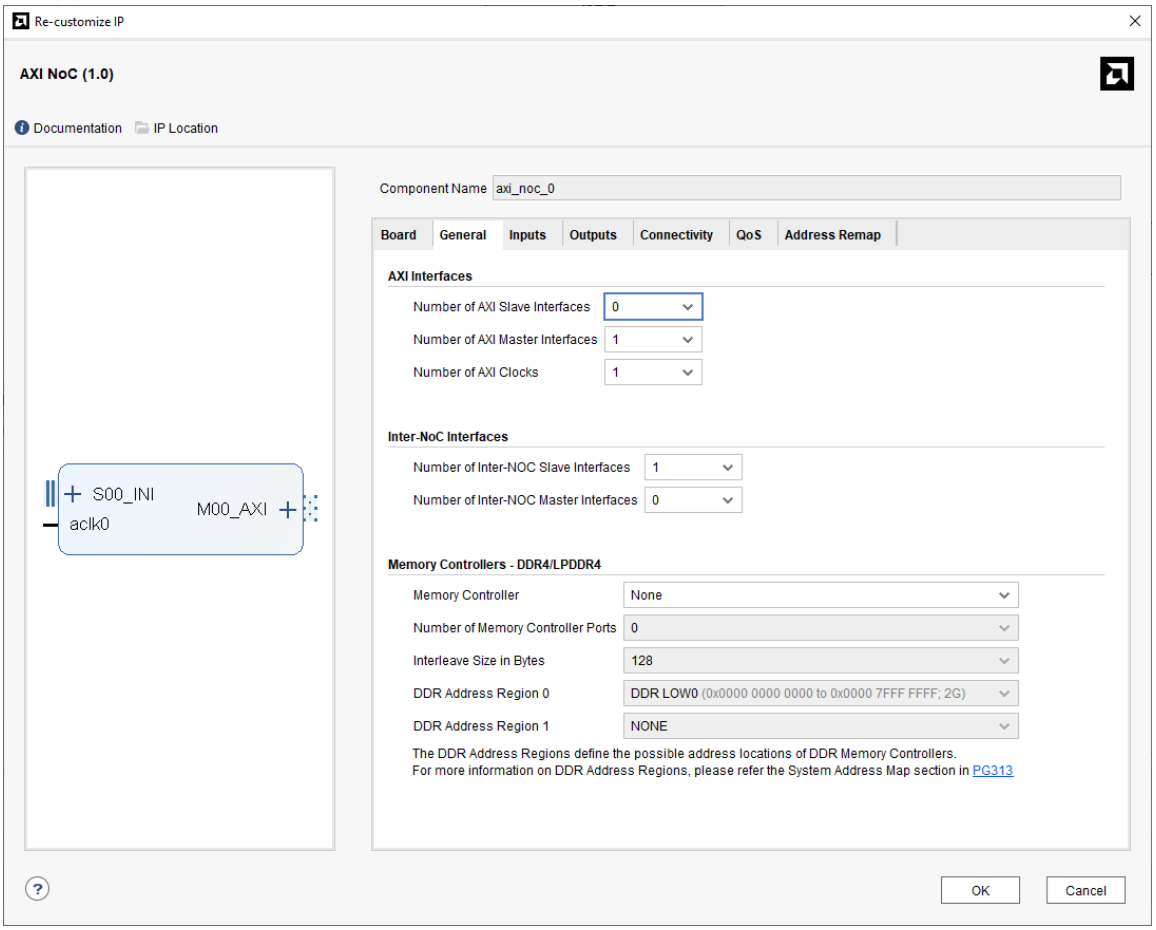
3. Click the + again and use the search field to add a Constant IP to the canvas. Double-click to customize. Change the **Const Width** to 32 and **Const Val** to 0xFACEFEED. Click **OK** to accept the edits.



4. Click the + again and use the search field to add an AXI SmartConnect IP to the canvas. Double-click to customize. Change the **Number of Slave Interfaces** to 1 so that all values are now 1. Click **OK** to accept the edits.



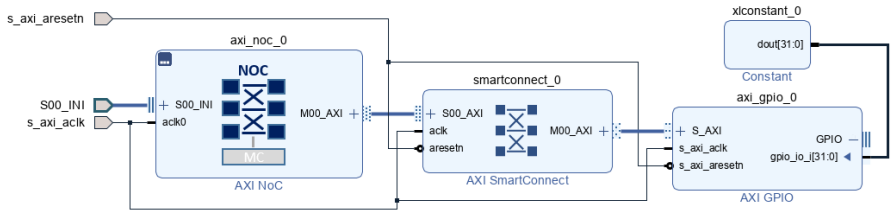
5. Click the + one last time and use the search field to add an AXI NoC IP to the canvas. Double-click to customize. Under the General tab, set the **Number of AXI Slave Interfaces** to 0 and the **Number of Inter-NOC Slave Interfaces** to 1. On the Connectivity tab, check the single box to connect M00_AXI to S00_INI. Click **OK** to accept the edits.



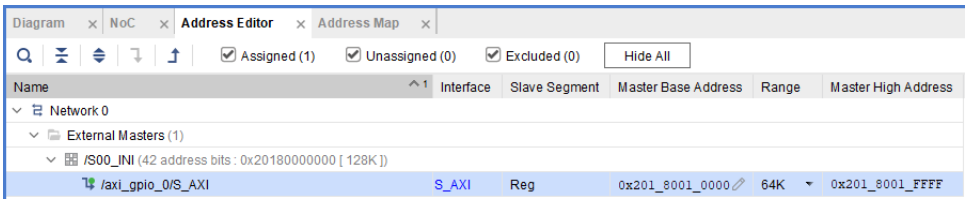
The Inter-NoC Interface (INI) provides a means of connecting two NoC instances. An INI link represents a logical connection within the physical NoC that is resolved at the time the NoC compiler is called. The boundary between static and reconfigurable sections of the NoC must be done using INI ports – note there is a corresponding AXI NoC IP with an Inter-NOC Master Interface in the static design.

6. Connect the pins to create the diagram as shown in following figure.

   ***Note*:** You need to expand the GPIO port to expose the 32-bit input bus to match the type of the Constant dout bus. Regenerate the layout to make it look nice.



7. Switch to the Address Editor window and see that no addresses are assigned. Right-click the row for `/axi_gpio_0/S_AXI` and select **Assign**. This sets a 64K range starting at address 0x0000_0202_0000_0000.
8. Modify the **Master Base Address** so it starts at 0x0201_8001_0000 then keep the **Range** at 64K.



9. Back in the block design, click **Validate Design** and then **Save** to save the rp1rm2 block design.

   In this simple design, there are only two differences between rp1rm1 and rp1rm2:

   a. The S_AXI base addresses are different.

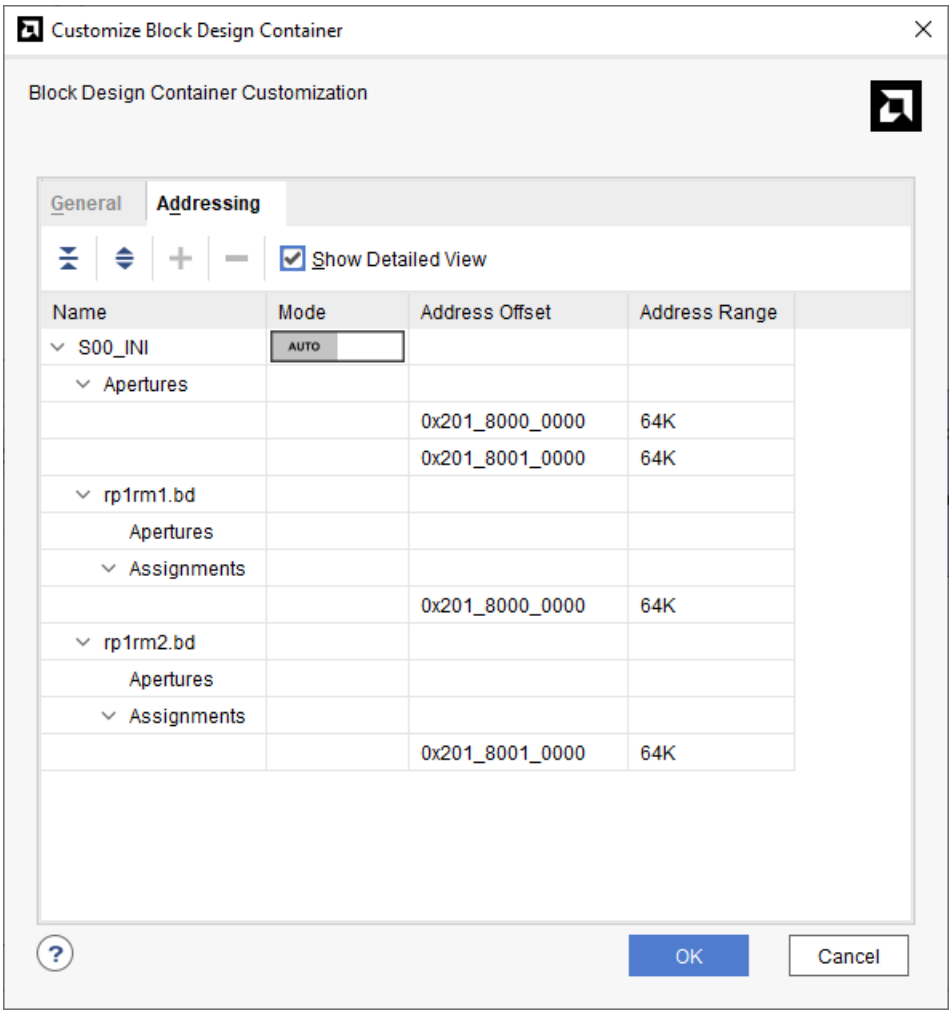
   b. The constant values that can be read via GPIO are different.

   This first difference is used to show that device tree overlays must be created and managed for designs that might have different requirements between Reconfigurable Modules. The second difference is used to confirm that dynamic reconfiguration in hardware has been done successfully.

Step 6: Confirm Apertures for All Reconfigurable Modules

Ensure that each RM has the appropriate aperture for its AXI slave interface, aligned to the instance in the top level. This is automatically done but can be manually set if desired.

1. In the top-level block design, double click on the rp1 block design container. Switch to the **Addressing** tab and click the **Show Detailed View** checkbox.



You can see that the overall aperture for rp1 for the S00_INI port starts at address 0x201_8000_0000 and has an overall range of 128K. This is automatically calculated by collecting address information from each design source in the block design container and summarizing each module's requirements.

If the aperture must be expanded to include new Reconfigurable Modules that have not been created yet, toggle the **Mode** from **Auto** to **Manual** and edit the master Offset or Range.

***Note*:** Adjustments to these values for individual block designs must be done at the source.bd.

2. Click **OK** to return to the top-level block design.
3. **Validate** and **save** design_1.bd.

Step 7: Create a Wrapper and Generate the Targets for the Top BD

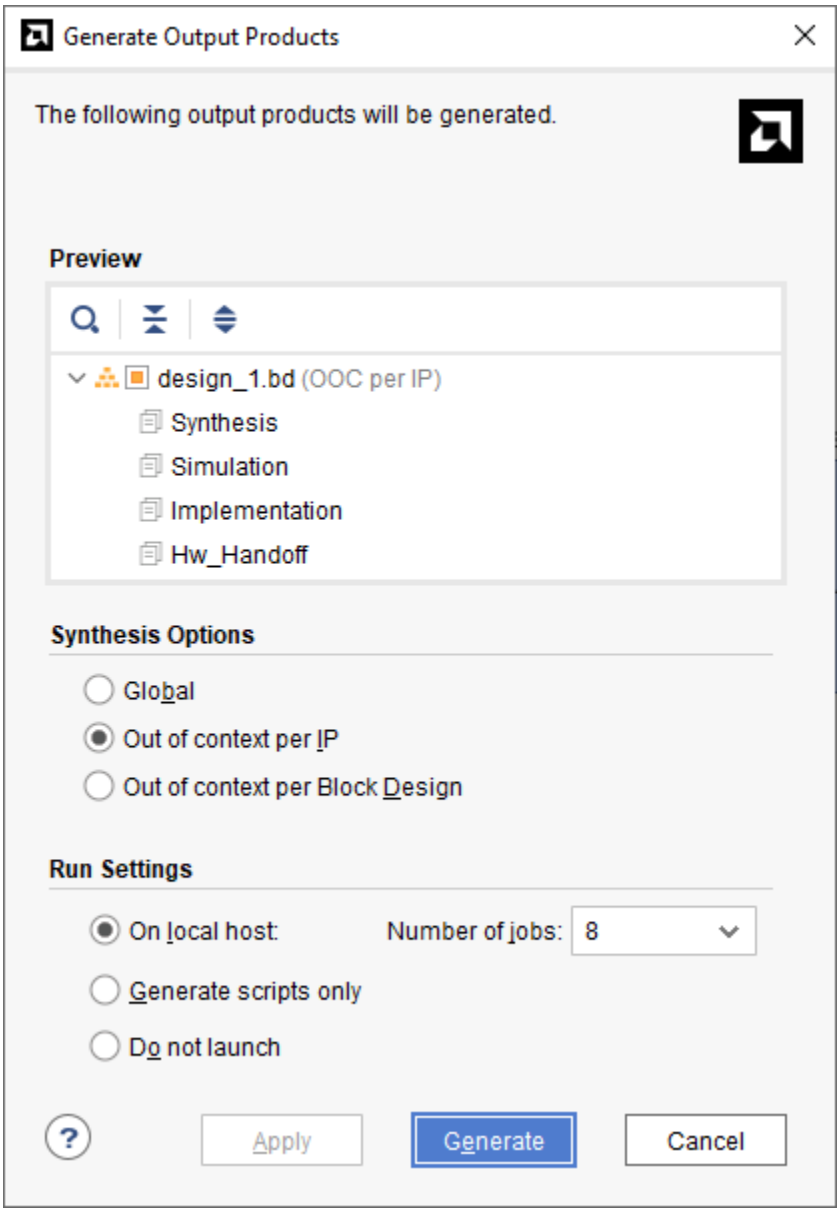
The final step before processing through synthesis and implementation is to create an HDL wrapper for the top-level block design, then generate targets for synthesis.

Follow the instructions below or source run_impl.tcl to automate the steps from here through the end of Section 1.

1. In the Sources window, right-click on design_1.bd and select **Create HDL Wrapper**. Keep the **Let Vivado Manage** option selected and click **OK**.

   In the sources, design_1_wrapper.v has been created and added to the project. This HDL file instantiates the design_1 block design.

2. In the Flow Navigator, click the **Generate Block Design** command under the IP INTEGRATOR header. In the resulting dialog box, keep the Out of context per IP option selected, then click **Generate**.



This action creates synthesizable output products for each IP in design_1, building out-of-context synthesis runs for each IP. Under the Design Runs tab, you see the list of synthesis runs for all the IP contained in design_1 (within the static_region hierarchy, so everything not included in the rp1 block container) are created and are now running. The IP within the block container for sources rp1rm1 and rp1rm2 are created but not running (this action will be requested later).

***Note*:** If the design has not been converted to a DFX design, this step performs that conversion automatically. The definition of a block design container as a DFX partition enforces the need to be in DFX project mode, but nothing to this point has required it.

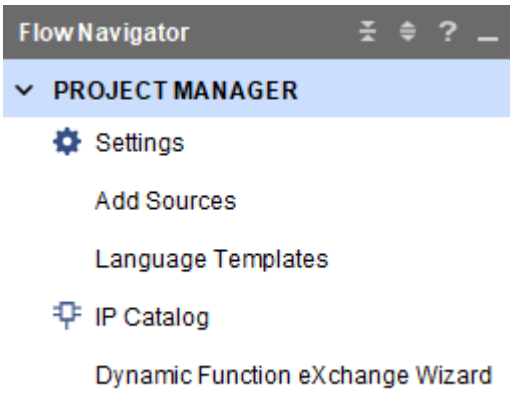
Step 8: Use the DFX Wizard to Define Configurations

The Dynamic Function eXchange Wizard is used to define relationships between the different parts of a DFX design. Using block design containers, you have created a level of the hierarchy of a design that can have more than one source in a DFX design. The block design container represents a Reconfigurable Partition (RP), and each source is a Reconfigurable Module (RM).

Within the DFX Wizard, define configurations and configuration runs. A configuration is a full design image, with one RM per RP. A configuration run is a pass of the place and route tools to create a routed checkpoint for that configuration. The DFX Wizard also establishes parent-child relationships between configuration runs, helping automate required parts of the flow including static design locking and pr_verify, and sets up dependencies between runs, so the AMD Vivado™ tools know what steps must be rerun when sources are modified.

For more information on the DFX project flow, see *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909).

1. Open the DFX Wizard by clicking **Dynamic Function eXchange Wizard** in the Flow Navigator or by selecting that option under the Tools menu.



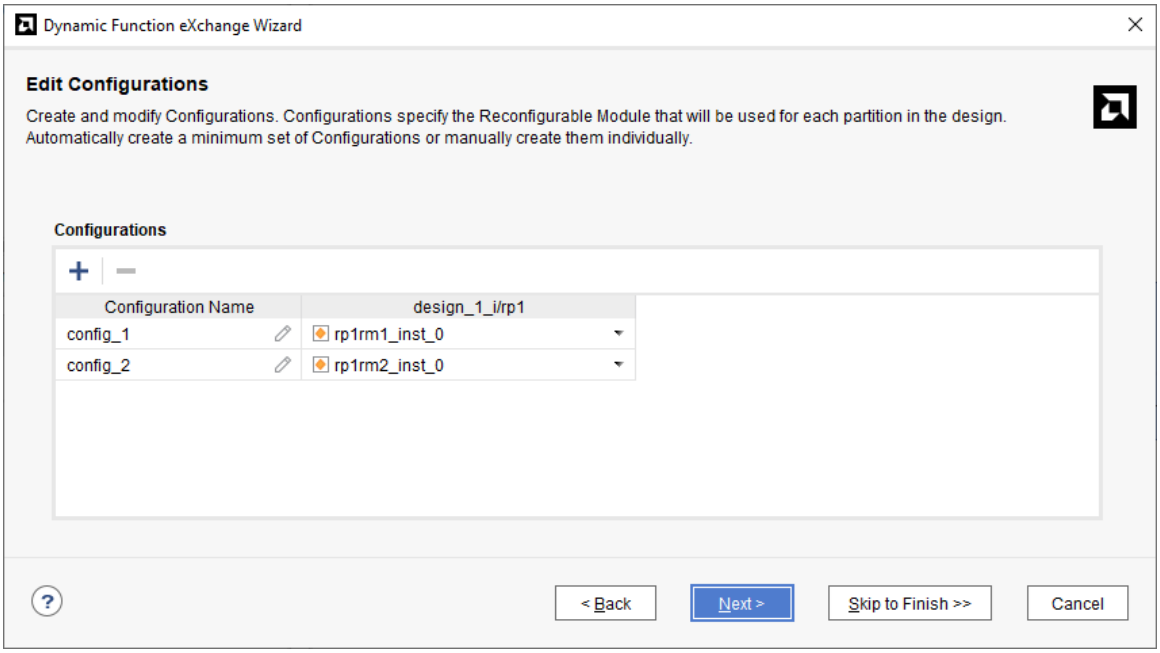
2. Click **Next**. In the Edit Reconfigurable Modules step, you see the two RMs: rp1rm1_inst_0 and rp1rm2_inst_0, that you created within the rp1 block design container.

> **IMPORTANT!** *Unlike within the RTL Project flow for Dynamic Function eXchange, the DFX Wizard is NOT the entry point for new Reconfigurable Modules. New RMs should be added from the canvas in the same way RM2 was added.*

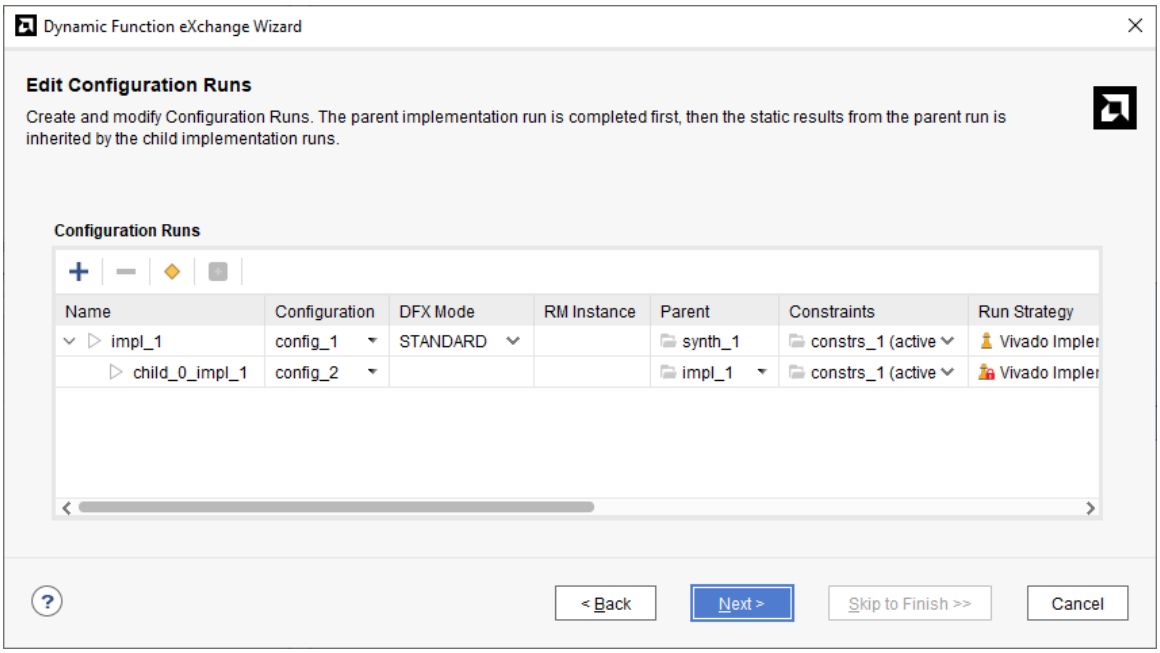
3. Click **Next**. In the Edit Configurations step, click the **automatically create configurations** link to generate two configurations.

   While you can also click the + button to generate these configurations, for designs with a single RP, automatic creation is the easiest way to create the list of configurations covering all RMs.

Note: The Configuration Name field is editable.



4. Click **Next**. In the Edit Configuration Runs step, click the **Standard DFX** link to create one run per configuration. For information on Abstract Shell configuration runs, see Lab 12: Abstract Shell Project Mode.



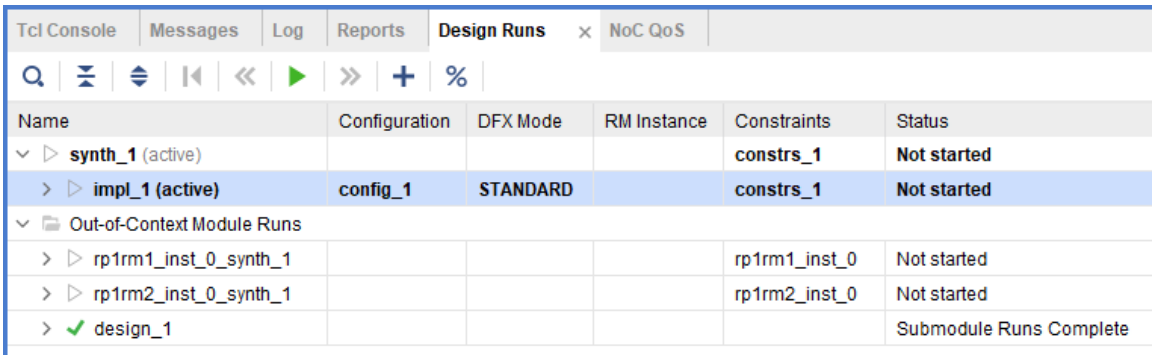
Note: The auto-generated names are not editable; the parent run is impl_1 and the child run is child_0_impl_1. Like the Configurations themselves, you can manually create the Configuration Runs, and when doing so, you can name the runs anything you'd like.

A critical aspect of the DFX project flow is the parent-child relationship, and that is shown here under the Parent category. A parent implementation starts with a synthesis run, and all child runs must reference the parent implementation to establish the static design consistency between them. In this simple example, the parent of child_0_impl_1 is impl_1, and the indentation of the child run illustrates this relationship.

5. Click **Next** then **Finish** to complete this section.

   The DFX Wizard can be revisited any time to create new or modified configurations and configuration runs within the project.

   In the Design Runs window, a new implementation runs for child_0_impl_1 has been created. Like the view in the DFX Wizard, this new run is indented to show its dependency on the impl_1 run above it.



Step 9: Add Design Constraints for the Reconfigurable Partition

All DFX designs require a floorplan. Each Reconfigurable Partition requires a Pblock containing enough programmable resources to implement any Reconfigurable Module that can be inserted in that partition. In this tutorial these Pblock constraints are created for you.

1. In the Sources window, click the + to open the Add Sources dialog box. Select **Add or create constraints** then **Next**. Click **Add Files** and navigate to the tutorial directory to find pblocks.xdc in the constraints directory, then click **Finish** to add this constraint file to the project.

   If desired, you can open the post-synthesis design to view the Pblock created for this design. For more information on floor planning requirements and methodology recommendations, see the *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909).

Step 10: Implement the Configurations and Generate Bitstreams

With all design sources now added to the project, and all settings complete for a DFX design, it is time to implement the design.

1. In the Design Runs window, right-click on child_0_impl_1 and select **Launch Runs**. Click **OK** to start the process.

   This action launches all runs necessary to implement both parent and child configurations, in the proper order.

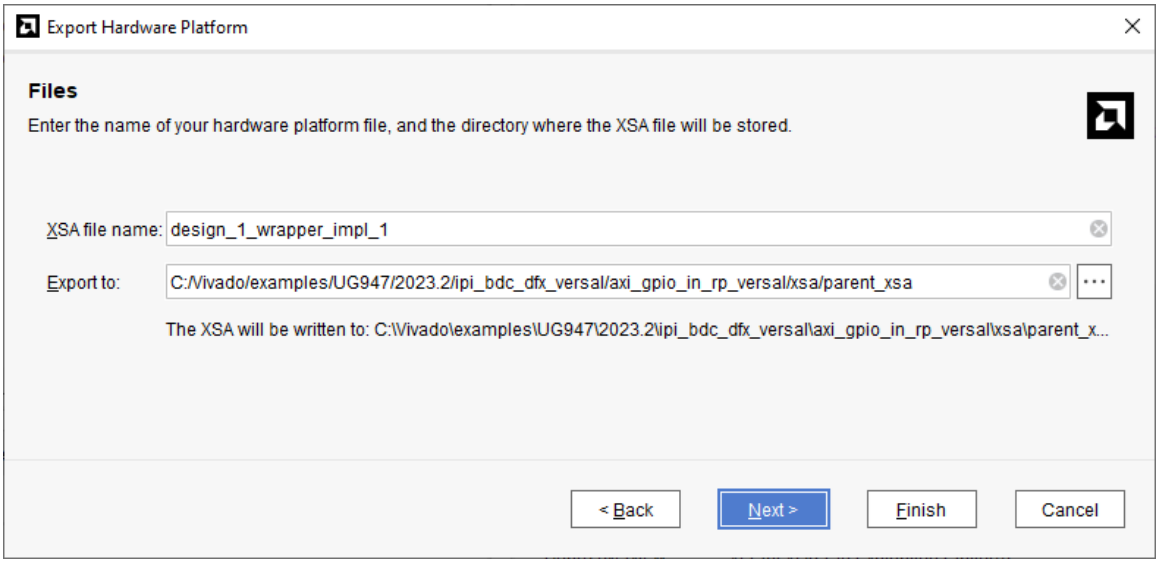
   - Out-of-context (OOC) synthesis runs for the two RMs. These are launched in parallel as they do not depend on each other.
   - Synthesis of the top-level design launches with the OOC runs completely. This completes very quickly as the top level is nothing more than IO insertion plus two black boxes.
   - The parent-run is implemented first. This is a standard AMD Vivado™ implementation run that applies DFX constraints. At the end of the run, multiple design checkpoints are written:
     1. A standard placed and routed checkpoint for the full design.
     2. A module-level checkpoint for the RM rp1rm1 was also placed and routed.
     3. A static-only design checkpoint, with all placement and routing locked, and a black box for rp1.
   - The child run is run last, and it starts with the locked static-only checkpoint from the parent-run.
2. Select **Cancel** on the resulting dialog box when implementation completes. Synthesized and implemented design checkpoints can be viewed at this point.
3. Shift-click in the Design Runs window to select both impl_1 and child_0_impl_1, then right-click to select **Generate Device Image**. Click **OK** in the resulting dialog to continue.

This creates full device bitstreams for both configurations, and partial bitstreams for rm1 and rm2 Reconfigurable Modules.

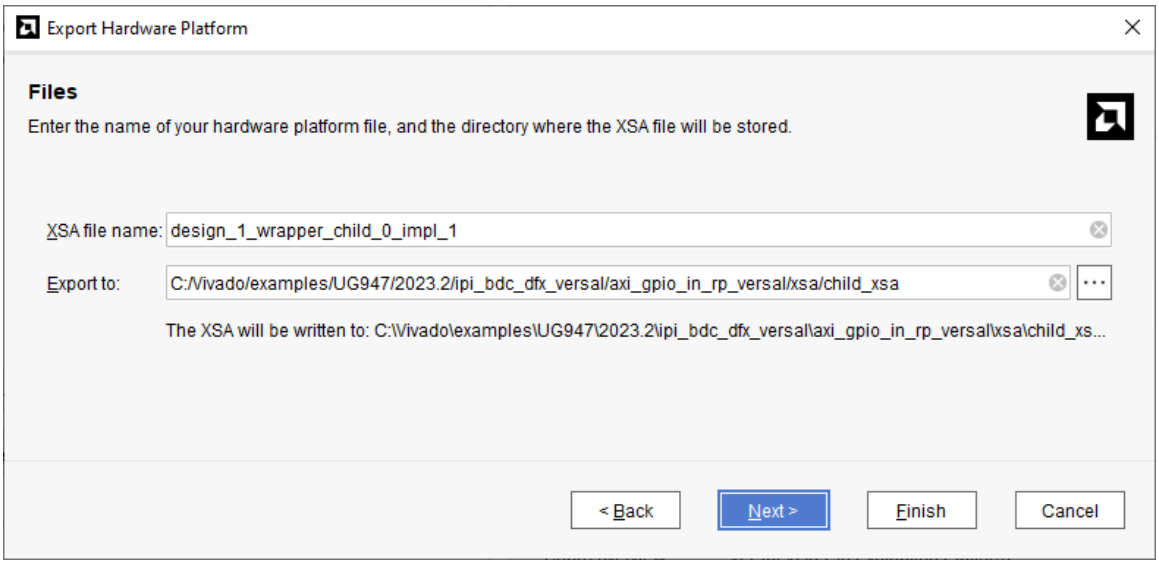
Step 11: Export the Hardware Platform for Each Configuration

The final step in the hardware flow is to export the platform for PetaLinux. This fixed Xilinx Support Archive (XSA) platform contains the full device bitstream as well as hardware handoff and other files needed to build a PetaLinux system in . In this release of AMD Vivado™, these .xsa files do not include partial bitstreams. Expect this enhancement in a future version of Vivado.

1. Open the impl_1 design run by right-clicking on that run and selecting **Open Run**.
2. Select **File > Export > Export Hardware**.
3. Select the **Include device image** option and click **Next**.
4. Add "_impl_1" to the **XSA file name**, then change the **Export to** the directory to the `xsa/parent_xsa` folder beside the current project directory. Click **Next**, then **Finish** to write the XSA.



5. Open the child_0_impl_1 design run by right-clicking on that run and selecting Open Run.
6. Repeat steps 1-4 but change the XSA file name to design_1_wrapper_child_0_impl_1, and change the Export to field to the xsa/child_xsa folder.



At this point, the Vivado IP integrator for DFX tutorial is complete.

Lab 11 Conclusion

This concludes lab 11. In this lab, you:

- Created AMD Versal™ project using Block Design Containers in IP integrator.
- Converted this project to become a DFX project.
- Added a second Reconfigurable Module for the Reconfigurable Partition.
- Processed the design through synthesis, implementation, and bitstream generation.

Lab 12

Abstract Shell Project Mode

Overview

This design demonstrates the methodology to debug DFX designs in AMD Versal™ devices using JTAG or HSDP. It covers the following debug scenarios:

- Debug Hub and ILA in the Static Region
- Debug Hub and ILA in a reconfigurable module (rp1rm1)
- Debug Hub and VIO in a reconfigurable module (rp1rm2)
- Debug Hub and two ILA in a reconfigurable module (rp1rm3) to demonstrate the automatic stitching of cores (ILA in this case) to the debug hub
- Static-RM interface for the Debug Hub in RM using Inter NoC Interconnect (NoC INI)
- Enabling HSDP in CIPS (specifically for the VCK190)
- Demonstrate using hardware manager to observe the waveforms generated by ILA

This design also showcases the Abstract Shell feature in IP integrator project mode. Design runs for both standard DFX and abstract shells are created to compare the two approaches. It covers the following features:

- Using the DFX Wizard to create both types of Design Runs.
- Compiling parent runs followed by parallel compilation of child runs.
- Examining the resulting runs to compare the two approaches.

Both solutions will generate a full collection of design checkpoints and partial PDI needed, but the latter will produce results more quickly.

Step 1: Extract the Tutorial Design Files

1. Download the reference design files.
2. Extract the zip file contents to any write-accessible location.

3. In the extracted files hierarchy, navigate to `\abstract_shell_project`.

Step 2: Compile the Design in the Vivado Tools

The entire lab can be run through implementation by sourcing the `run_all.tcl` script. This script calls underlying scripts that can also be run individually. The interactive lab in this document steps through details contained in some of these scripts, after block designs have been created.

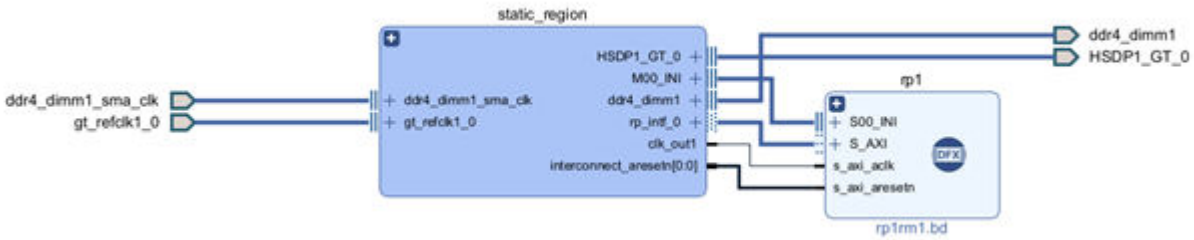
Bottom-Up IP Integrator Design Creation Approach

1. Open AMD Vivado™, and in the Tcl Console navigate to the folder where the design scripts and constraint file are located.
2. Source this script to generate the block designs for this lab.

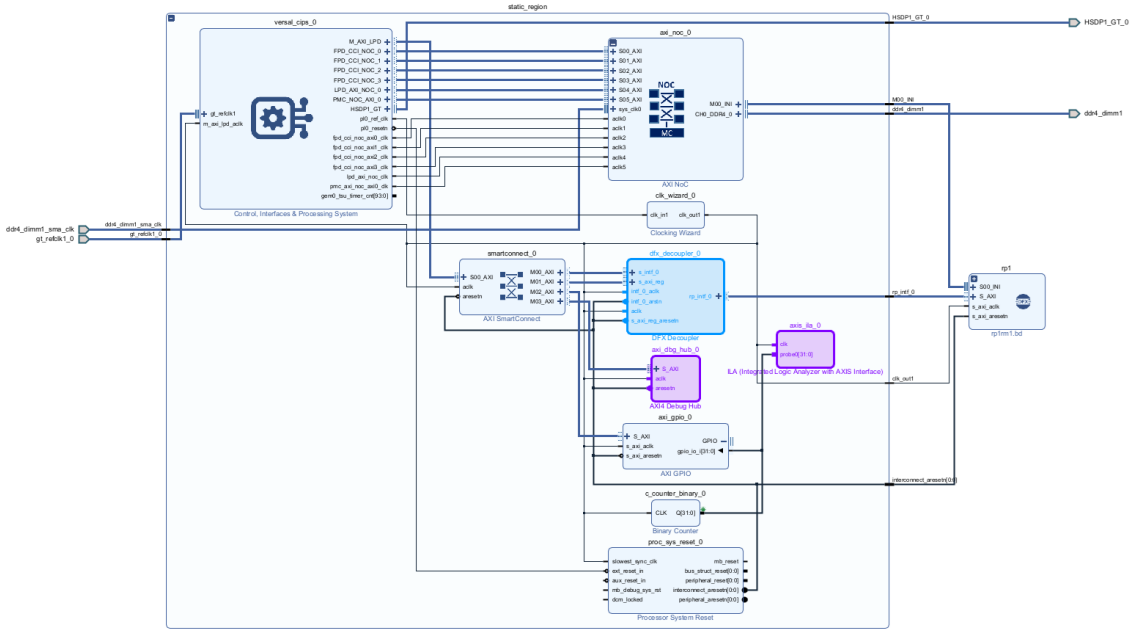
```
source create_ipi.tcl
```

The `create_ipi.tcl` script calls four underlying scripts, each of which creates an individual block design. The first three are reconfigurable modules (RM), and the fourth is a top block design that uses a block design container to reference the individual reconfigurable modules. Given that the RMs are created before the top-level, this is considered a bottom-up approach.

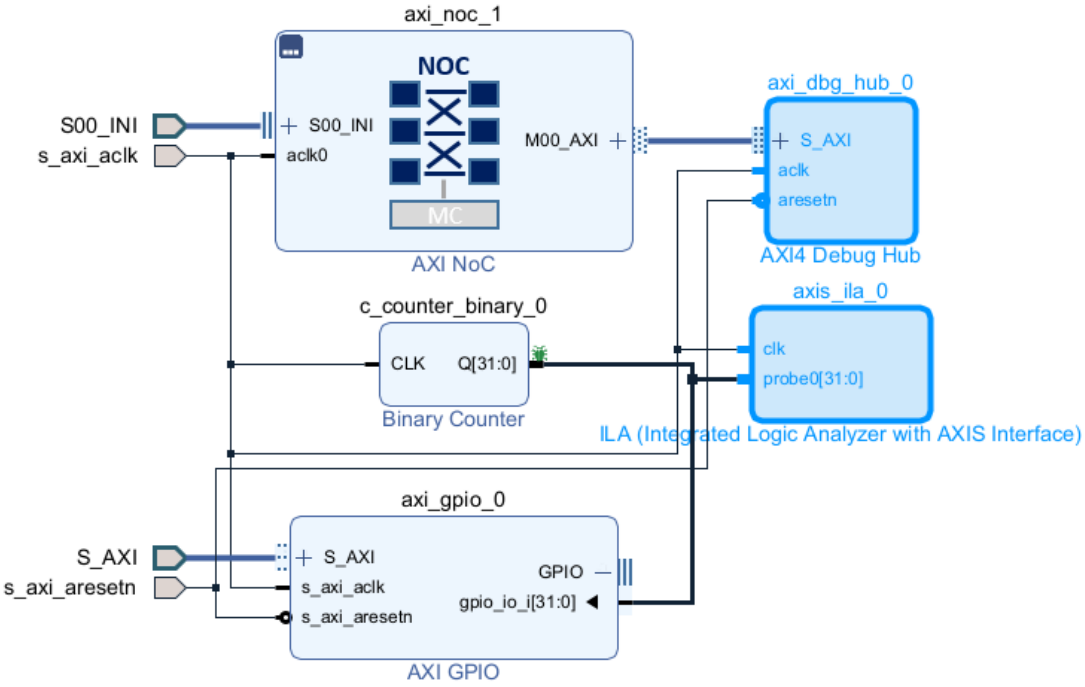
Once the design creation is complete, observe that the design is divided into one static region and one reconfigurable partition.



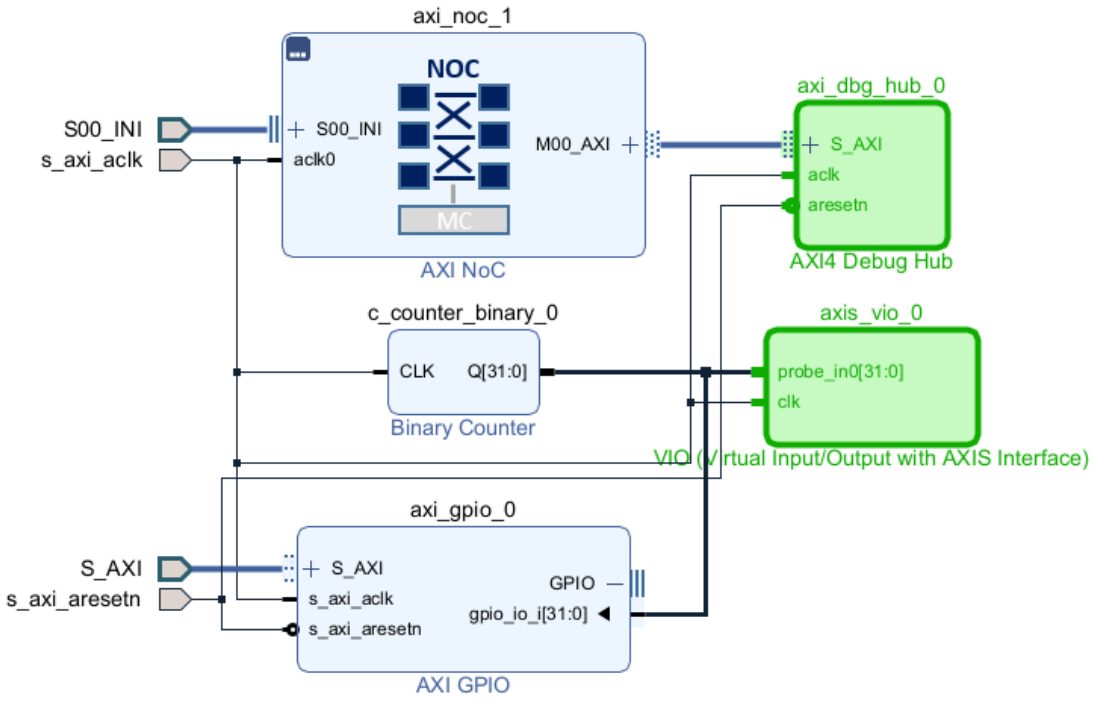
The static_region is a level of hierarchy that sets up the management of the design. It includes CIPS, NoC (with DDR), clocking, and reset. It also includes a static debug hub, an ILA core, and a DFX decoupler instance.



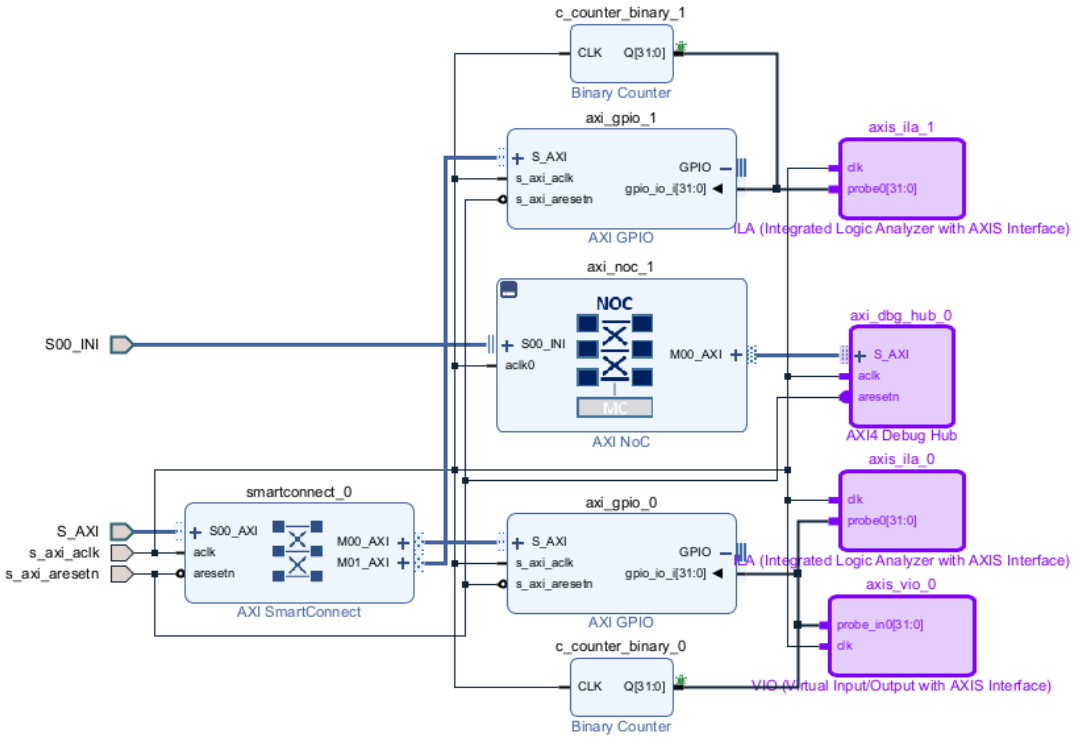
Reconfigurable Module rp1rm1 is the default RM and can be seen when expanding the Block Design Container rp1. This first module has an up counter connected to ILA. This ILA will be stitched to the Debug Hub in the rp1rm1 during opt_design of the parent run.



Reconfigurable Module rp1rm2 has a down counter connected to VIO. This debug core is stitched to the Debug Hub in rp1rm2 during `opt_design` of the first child run.



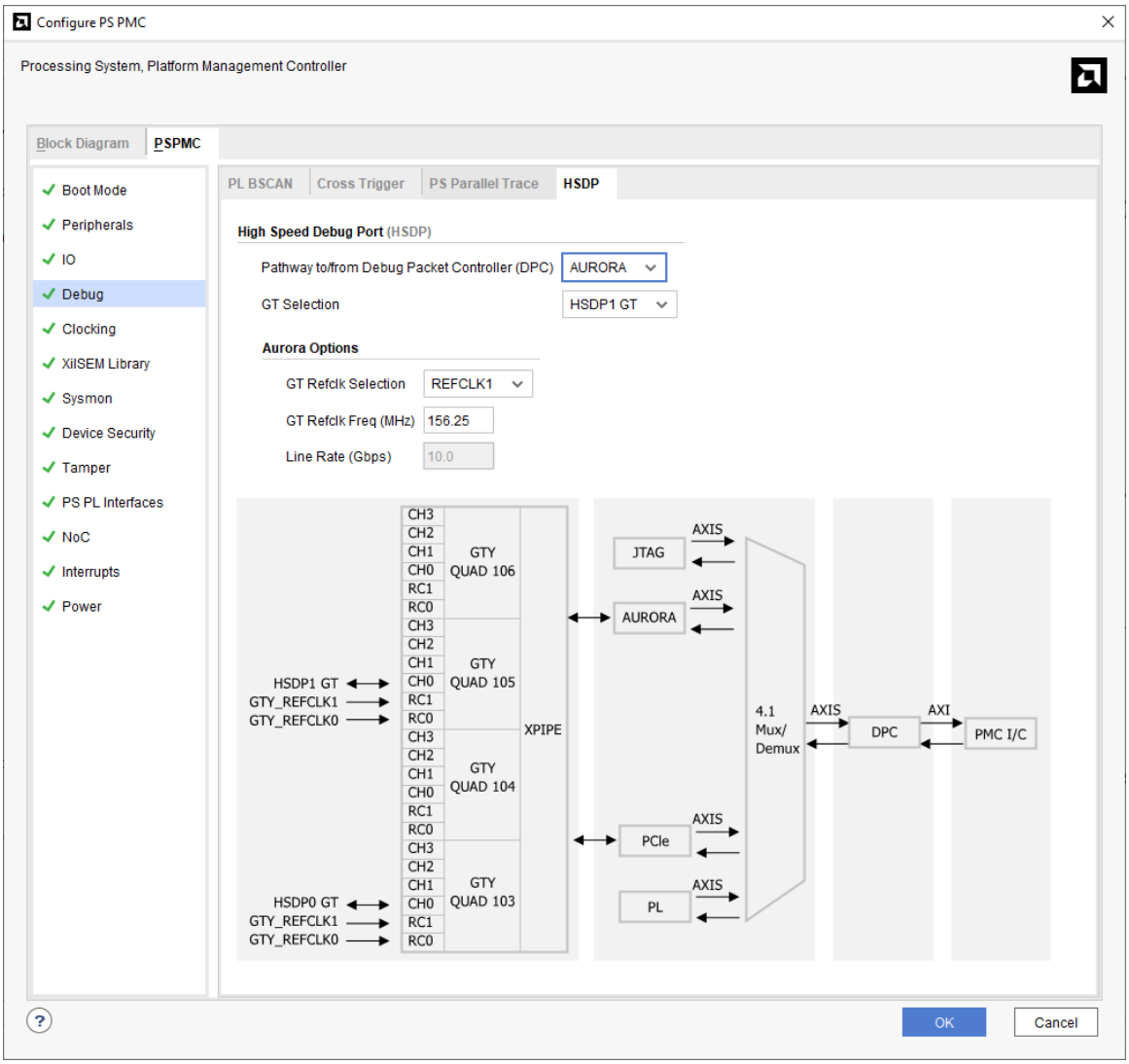
Reconfigurable Module rp1rm3 has a down counter and up counter connected to two AXI GPIOs. Both counters are connected to separate ILAs. Both of the ILAs, along with a VIO core, is connected to the Debug Hub in rp1rm3 during `opt_design` of the second child run.



Note: This tutorial shows the ChipScope instantiation flow, where ILA and VIO cores are explicitly added to the design on the block design canvas. The Vivado tools also supports the ChipScope insertion flow, where signals are tagged for debug and core details are added after synthesis. For an example of the insertion flow, see the DFX Debug Tutorial available from the GitHub repository.

3. Enable HSDP1 for CIPS.

   Follow the steps from the HSDP tutorial to enable HSDP for high speed debug. Confirm that the Aurora-based HSDP has been enabled by opening the **CIPS IP → PS PMC → Debug selection → HSDP** tab.



   To complete the set of design sources, first add the constraint file that defines the Pblock for the reconfigurable region. Alternatively, you can create this Pblock yourself after synthesis.

4. Use the Add Sources button to add the Pblocks constraint file, or call this command on the Tcl Console:

```
add_files -fileset constrs_1 -norecurse ./pblocks.xdc
```

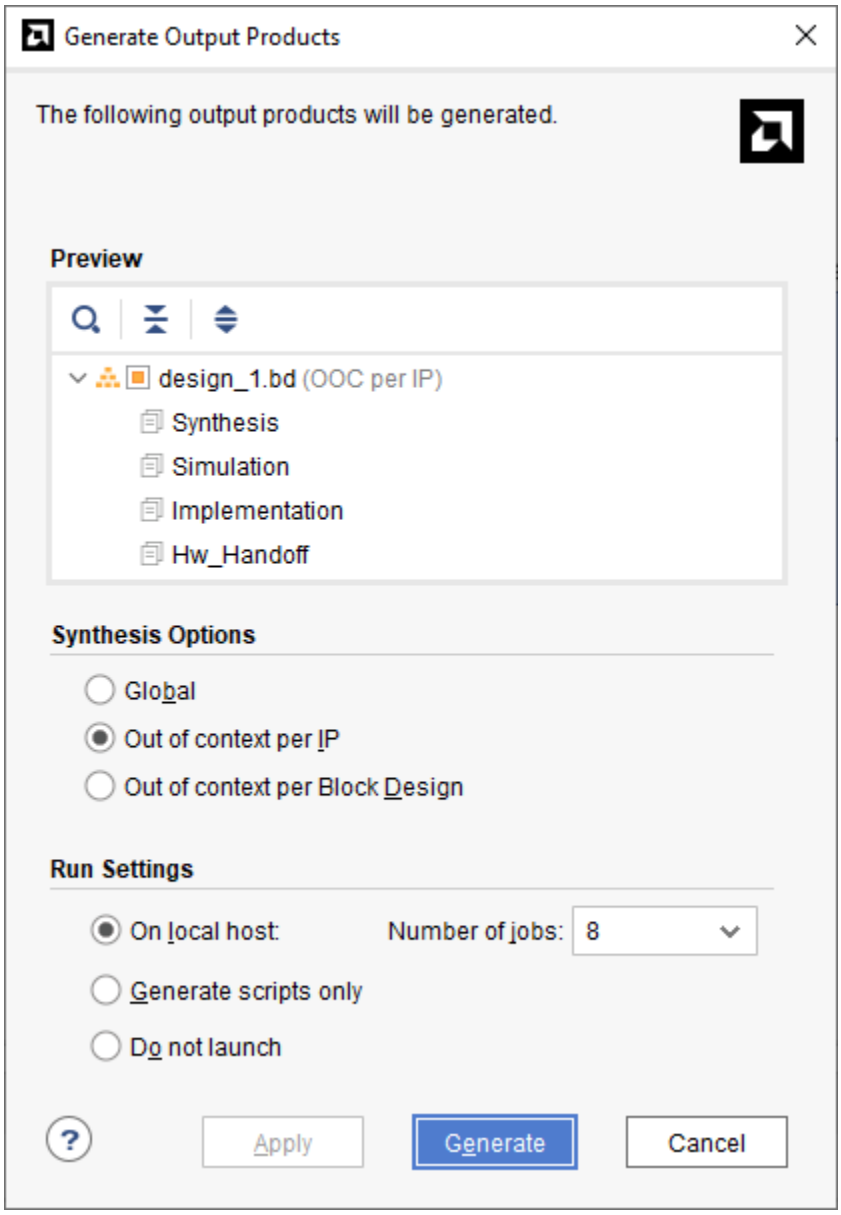
Next, create the HDL wrapper for the top-level block design.

5. Right-click `design_1.bd` and select **Create HDL Wrapper → OK** to have this file auto-generated and added to the project. The corresponding Tcl for this action is:

```
make_wrapper -files [get_files ./dfx_debug_abs/dfx_debug_abs.srcs/
sources_1/bd/design_1/design_1.bd] -top
```

Step 3: Set Up DFX Design Runs

1. Generate the block design:
   a. With `design_1.bd` open and active, click **Generate Block Design** in the Flow Navigator.
   b. Select **Out of context per IP**, then click **Generate**.



This creates IP and RTL products and runs synthesis for static design components and IP inside RMs. RM top-level generation and synthesis occur later in separate block designs.

2. Open the DFX wizard:
   a. When synthesis completes, the DFX Wizard appears in the Flow Navigator.
   b. Click **Dynamic Function eXchange Wizard** to begin creating complete design configurations from static and dynamic portions.

      With one RP, this design shows the simplest use case.
3. Edit configurations:
   a. Click **Next** to get to the Edit Configurations page.

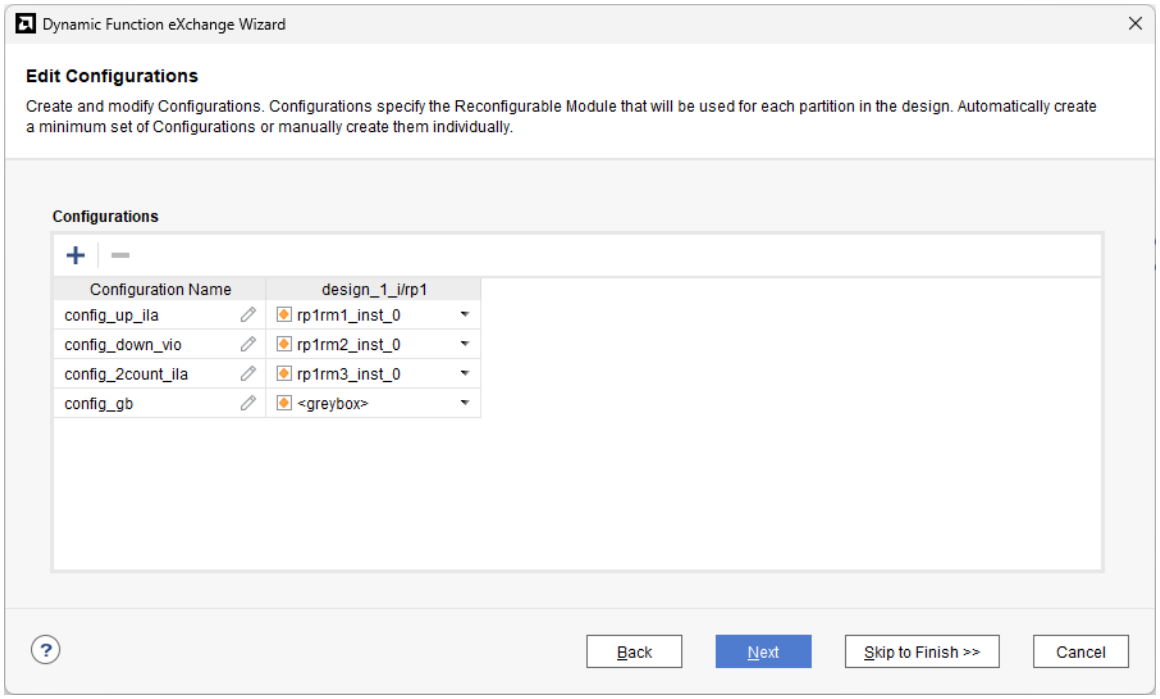
b. Click **automatically create configurations** to generate three configurations for the three RMs.

c. Use the pencil icons to change the names of each of these configurations. Name them as follows:

- `config_up_ila`
- `config_down_vio`
- `config_2count_ila`

d. Click **+** to add a fourth configuration named `config_gb`, then click **OK**.

e. Assign `<greybox>` to the RP.

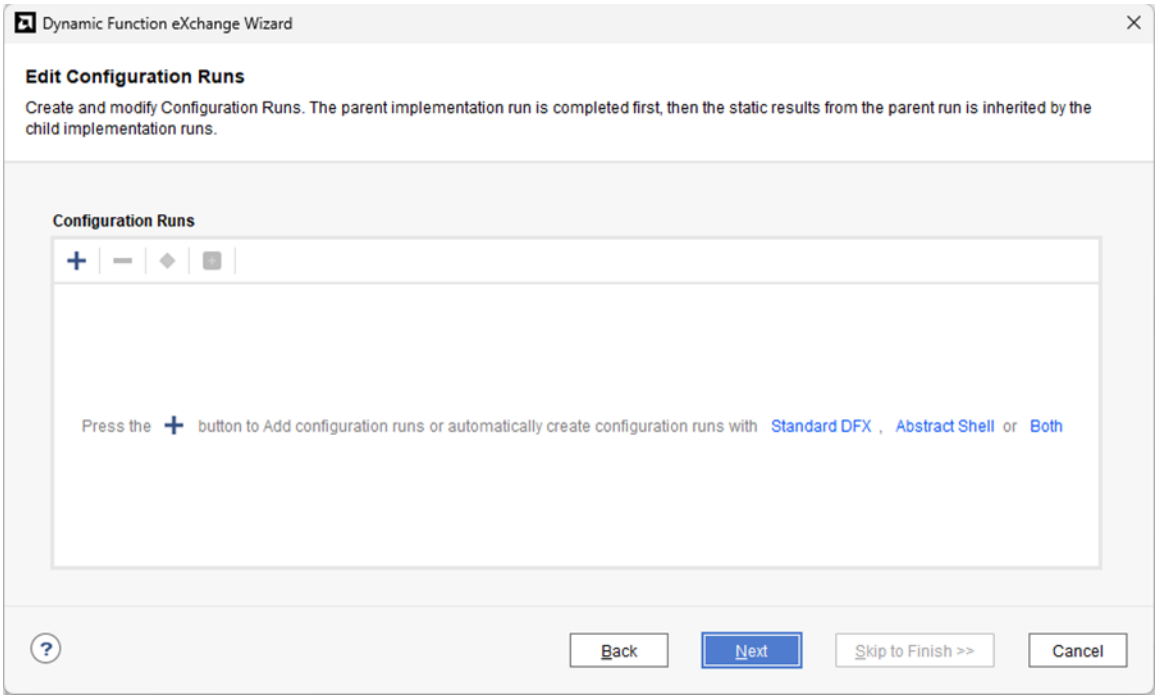


f. Click **OK**, then `Next`.

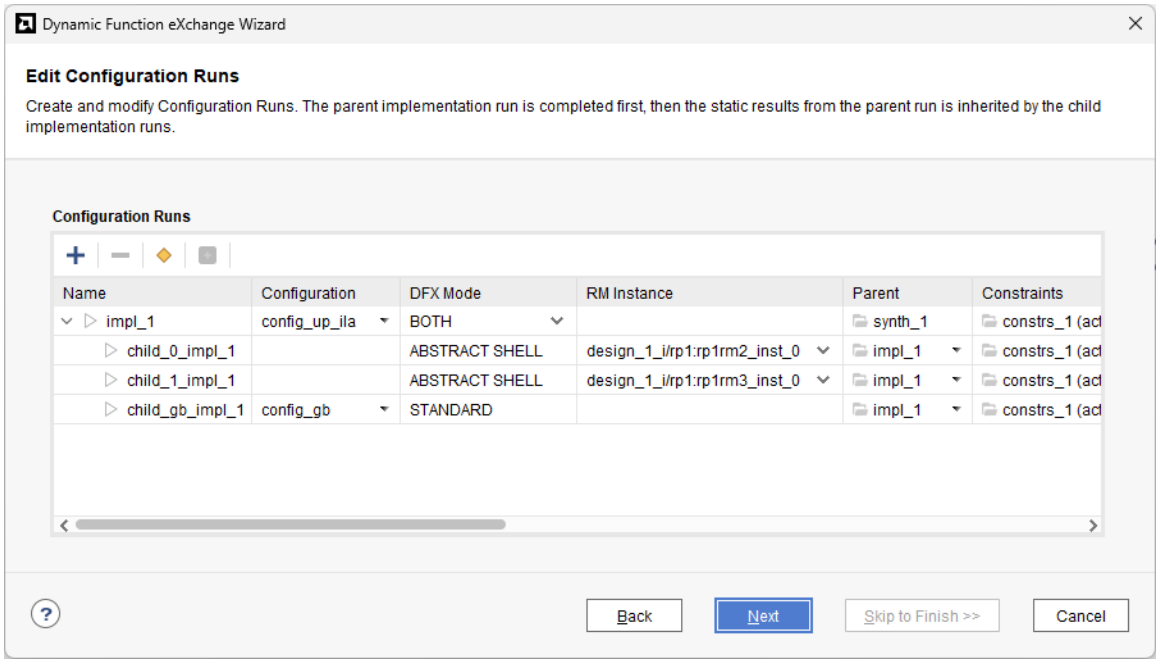
4. Edit configuration runs:

a. On the Edit Configuration Runs page, manually create the configuration runs to explore the options for the Standard DFX or Abstract Shell runs flows.

- Standard DFX creates one run per configuration from the prior page. The first configuration becomes the parent run; the rest are child runs of that parent.
- Abstract Shell creates one parent run, and one child run per RM. In this design, you also create three runs. For designs with multiple RPs, you create more runs.
- Select Both to create one parent run then one child run per RM, matching the Abstract Shell option. This choice also creates full and abstract shells, enabling extra child runs using full configurations.



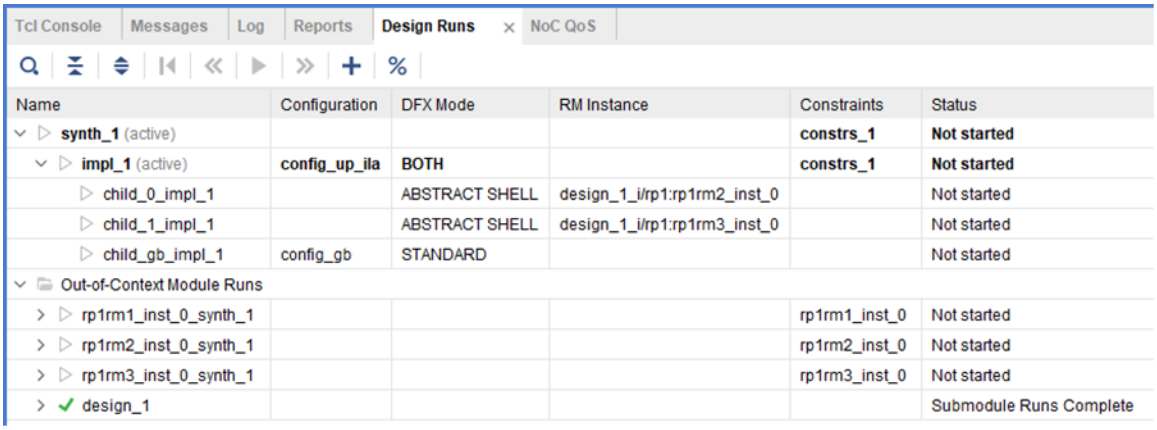
5. Create minimum configuration runs:
   a. Click **Both** to generate the minimum set of configuration runs for the project.
   b. In this design, you create one parent and two child runs.
   c. This action creates implementation results for the static design and all the RMs defined by block designs.
   d. Because no child run exists for the greybox configuration, create one manually:
      - Run: `child_gb_impl_1`
      - Parent: `impl_1`
      - Child Run DFX Mode: `STANDARD`
      - Configuration: `config_gb`
6. Finalize configuration runs and build:
   a. Click **OK**.
   b. The DFX Wizard displays your defined configuration runs:



With the Configuration Runs defined as shown, the following results are generated:

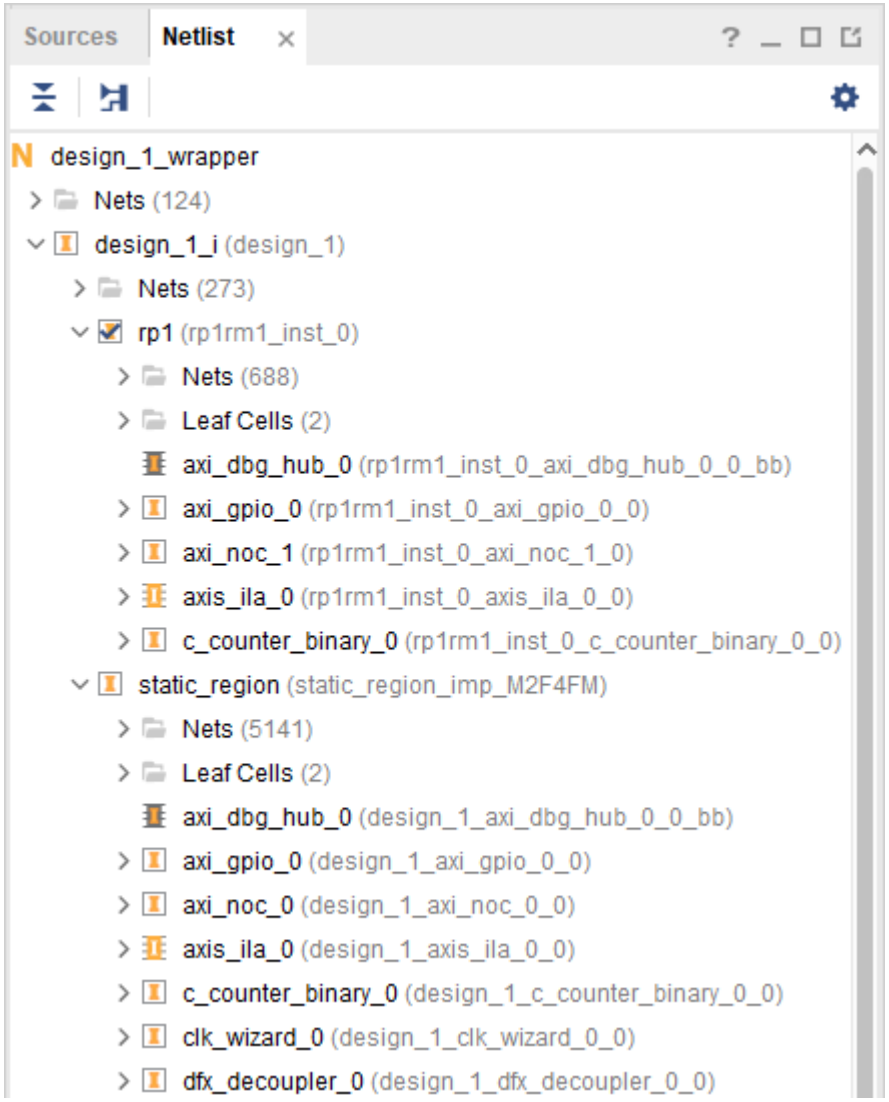
- **impl_1:** Implements a full design with the `count_up_ila` module included. Both a static black box checkpoint and an abstract shell checkpoint generate.
- **child_0_impl_1 and child_1_impl_1:** Use abstract shell to implement the `count_down_vio` and `2count_ila` modules.
- **child_gb_impl_1:** Uses the full static shell for the greybox configuration.

7. Click **Next**, then **Finish** to create these design runs. The Design Runs tab now lists your defined selections.



8. Run synthesis:
   a. In the Flow Navigator click **Run Synthesis**, then click **OK**. This runs synthesis for the top layer of each RM and for the top-level design.
   b. When synthesis completes, open the synthesized design.

c. In the Netlist view there are black boxes for the debug hubs. These are inserted later during `opt_design`.



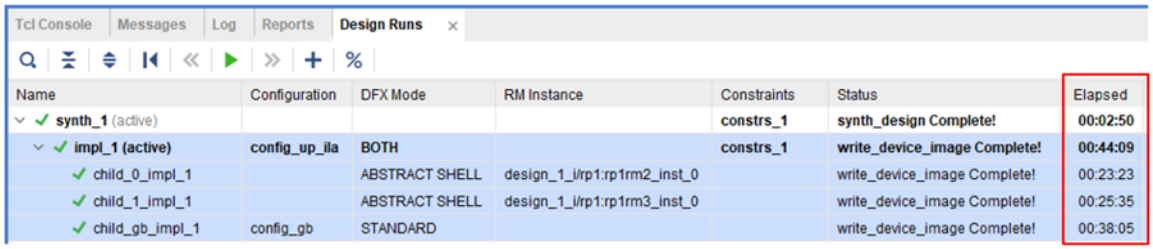
9. Launch implementation runs:
   a. In the Design Runs window, shift-click all implementation runs.
   b. Right-click in this area and select **Launch Runs**, then **OK**. This action kicks off runs in the necessary order.
   c. The parent run compiles first.
   d. After the parent run completes, each child below the parent runs in parallel.
10. Debug hub insertion:
   a. Debug hubs insert and connect to debug cores during `opt_design` in implementation.
   b. For each implementation run, observe the following lines of code in the log file.
      - Parent implementation: insertion happens in the static design.

- Child implementation: insertion happens inside a locked static design from the parent.

```
Phase 2 Generate And Synthesize Debug Cores
INFO: [Chipscope 16-329] Generating Script for core instance :
design_1_i/rp1/axi_dbg_hub_0
INFO: [IP_Flow 19-3806] Processing IP xilinx.com:ip:axi_dbg_hub:2.0
for cell rp1rm1_inst_0_axi_dbg_hub_0_0_bb.
INFO: [Chipscope 16-329] Generating Script for core instance :
design_1_i/static_region/axi_dbg_hub_0
INFO: [IP_Flow 19-3806] Processing IP xilinx.com:ip:axi_dbg_hub:2.0
for cell design_1_axi_dbg_hub_0_0_bb.
INFO: [Chipscope 16-324] Core: design_1_i/rp1/axi_dbg_hub_0 UUID:
2ba56efa-44aa-5cff-b4f7-161e160672ee
INFO: [Chipscope 16-324] Core: design_1_i/static_region/axi_dbg_hub_0
UUID: a46ff48e-6629-59e0-a9fe-ed94bf289898
```

11. Compile time for the parent run takes longer than child runs due to static design implementation and multiple checkpoint creation.
    a. Tools carve out the RP (using `update_design -black_box`) , lock static design (using `lock_design -level routing`), and create `design_1_wrapper_routed_bb.dcp`.
    b. The routed checkpoint reopens and the tools embed these two steps in `write_abstract_shell`. The additional carving and associated checks are remove the bulk of the static design, resulting in the abstract shell file `abs_shell_design_1_i_rp1.dcp`.
    c. Compare the file sizes of `design_1_wrapper_routed_bb.dcp` and `abs_shell_design_1_i_rp1.dcp`. The full shell is almost four times the size of the abstract shell for this design.
12. Standard DFX flow child runs compile slower than abstract shell runs due to larger checkpoints.
    - This is where the Abstract Shell flow provides benefit. These runs have smaller checkpoints to open and less information to process so the overall compile times are reduced.
    - In this design this is even true with the simple greybox configuration, but that is not the case in most true designs.
13. Select runs, right-click and select **Generate Device Image**.
14. The following results are generated:
    - `impl_1` generates full and partial bitstreams (complete design image).
    - `child_0_impl_1` and `child_1_impl_1` generates partial bitstreams only (abstract shell).
    - `child_gb_impl_1` generated full and partial bitstreams (uses the full static shell to implement a greybox configuration).

Figure 6: **Final Design Runs Window**



***Note*:** Compile times differ based on many factors, including Vivado tool version and operating system.

On this machine, the Abstract Shell child runs take about half the time of the Standard DFX child runs. For larger designs with multiple or small RPs, the savings are even more dramatic. The greybox run takes less time than the parent run because you do not compile the static design, but more time than the complex RM runs using abstract shells.

If you want to boot to a specific initial configuration, create a standard DFX run with your desired configuration to generate a full device bitstream. Alternatively, assemble a full design from the black box checkpoint of the parent run combined with the routed RM checkpoint from an abstract shell run. Generate the full design bitstream using `write_device_image` (Versal targets) or `write_bitstream` (FPGA targets).

Important Notes about Bitstream/PDI Generation using the Abstract Shell Flow

Parent runs are complete design images containing the complete static logic and dynamic logic, for all reconfigurable modules in the design. The final routed design checkpoint can be used to generate a full device programming image plus one partial programming image per reconfigurable partition. This is the same capability as any parent or child run when using the standard DFX flow.

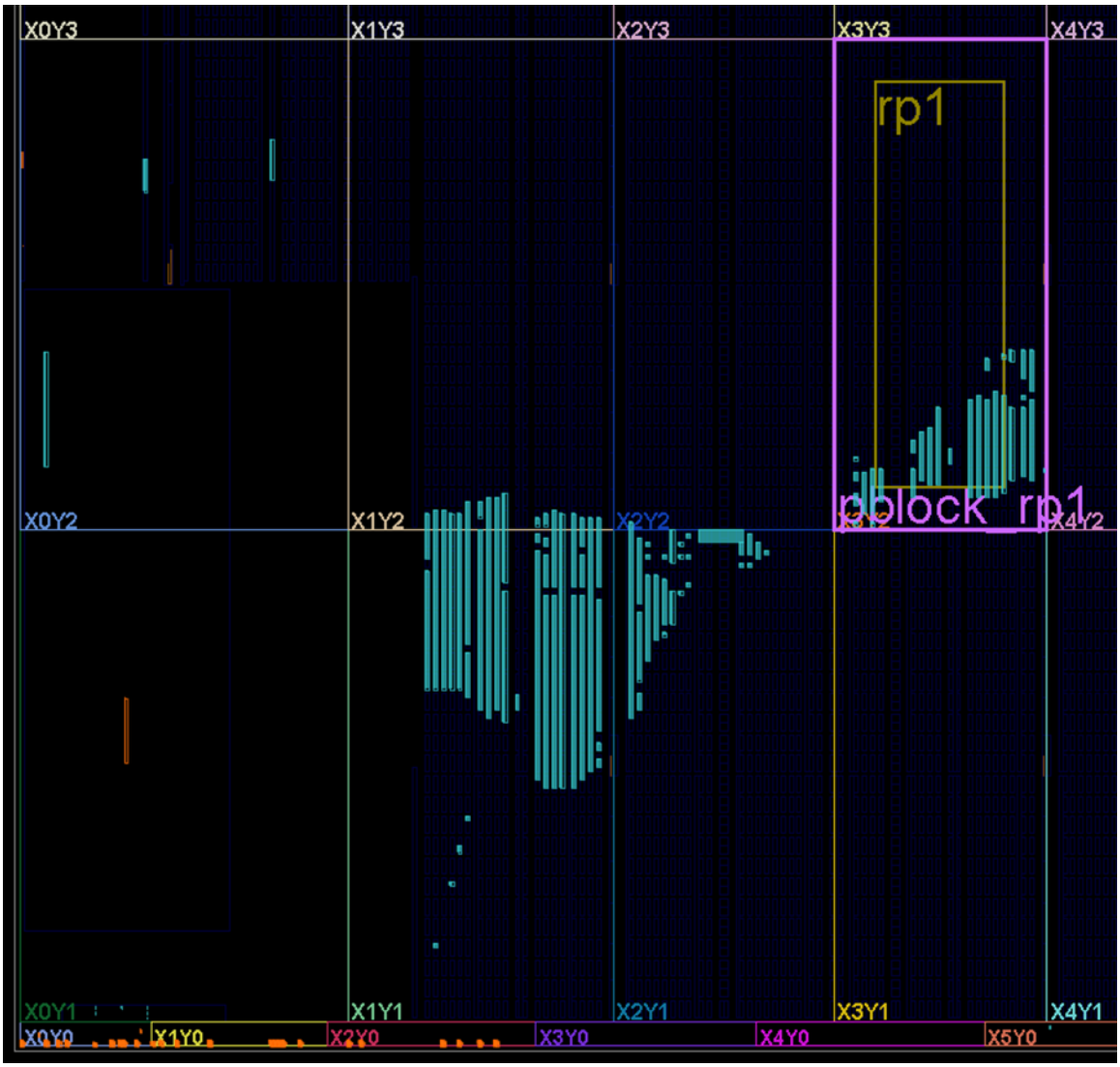
Child runs only contain the abstract shell plus the reconfigurable module implemented within that shell, so routed designs only have the ability to generate partial programming images for that reconfigurable module. A call to `write_bitstream` or `write_device_image` is done using the `-cell` option to clearly request the partial image needed (even though there is only one possible in that checkpoint).

If full design images with RMs implemented in child runs are desired, you must first reassemble target configurations by linking full shell and reconfigurable module routed checkpoints using `open_checkpoint` and `read_checkpoint -cell`, or `add_files` and `link_design`. After assembled, the current design in memory is a complete DFX design and a full bitstream can be generated using a call to `write_bitstream`.

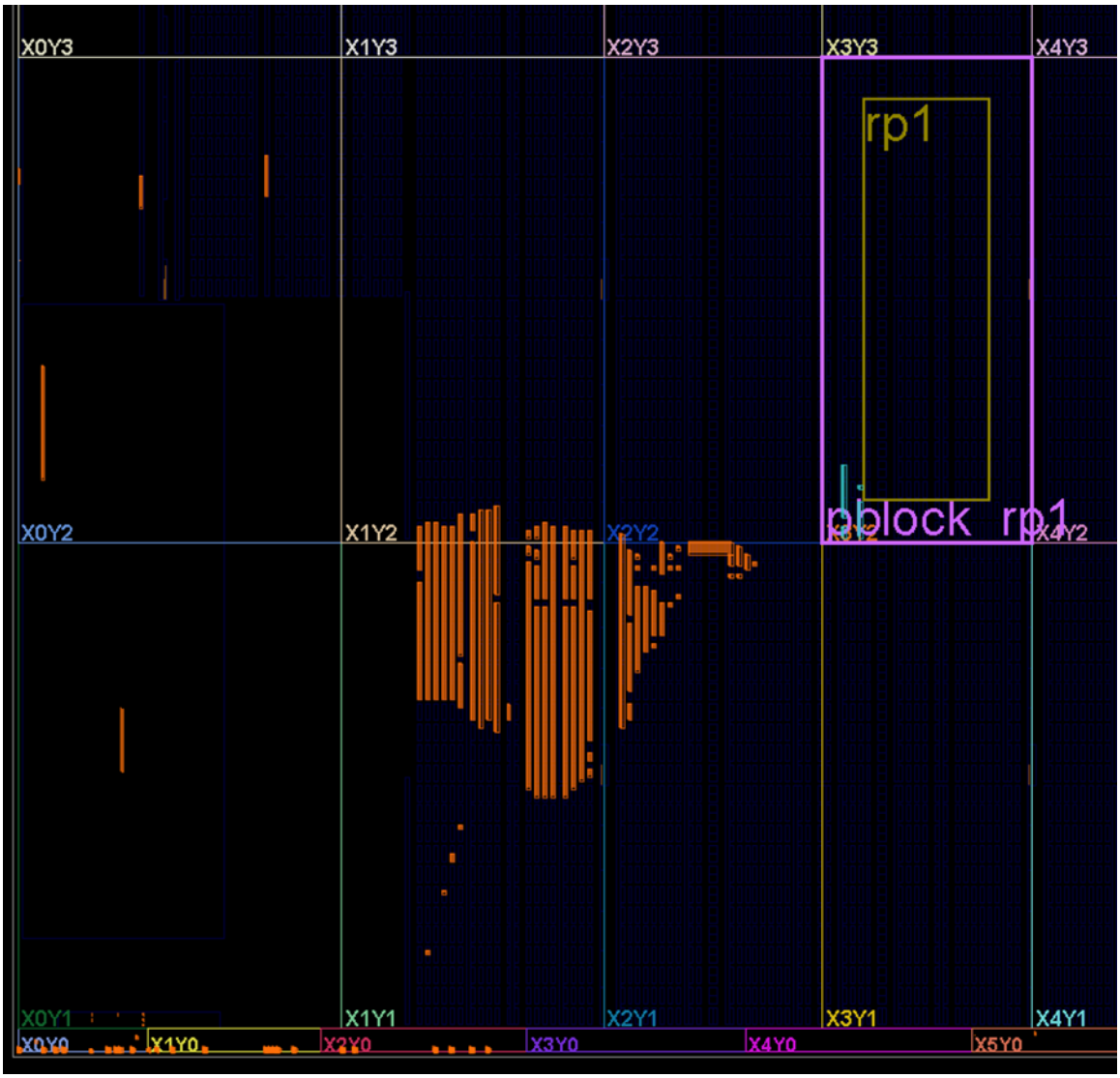
Step 4: Examine the Results

Open individual runs to examine the design results.

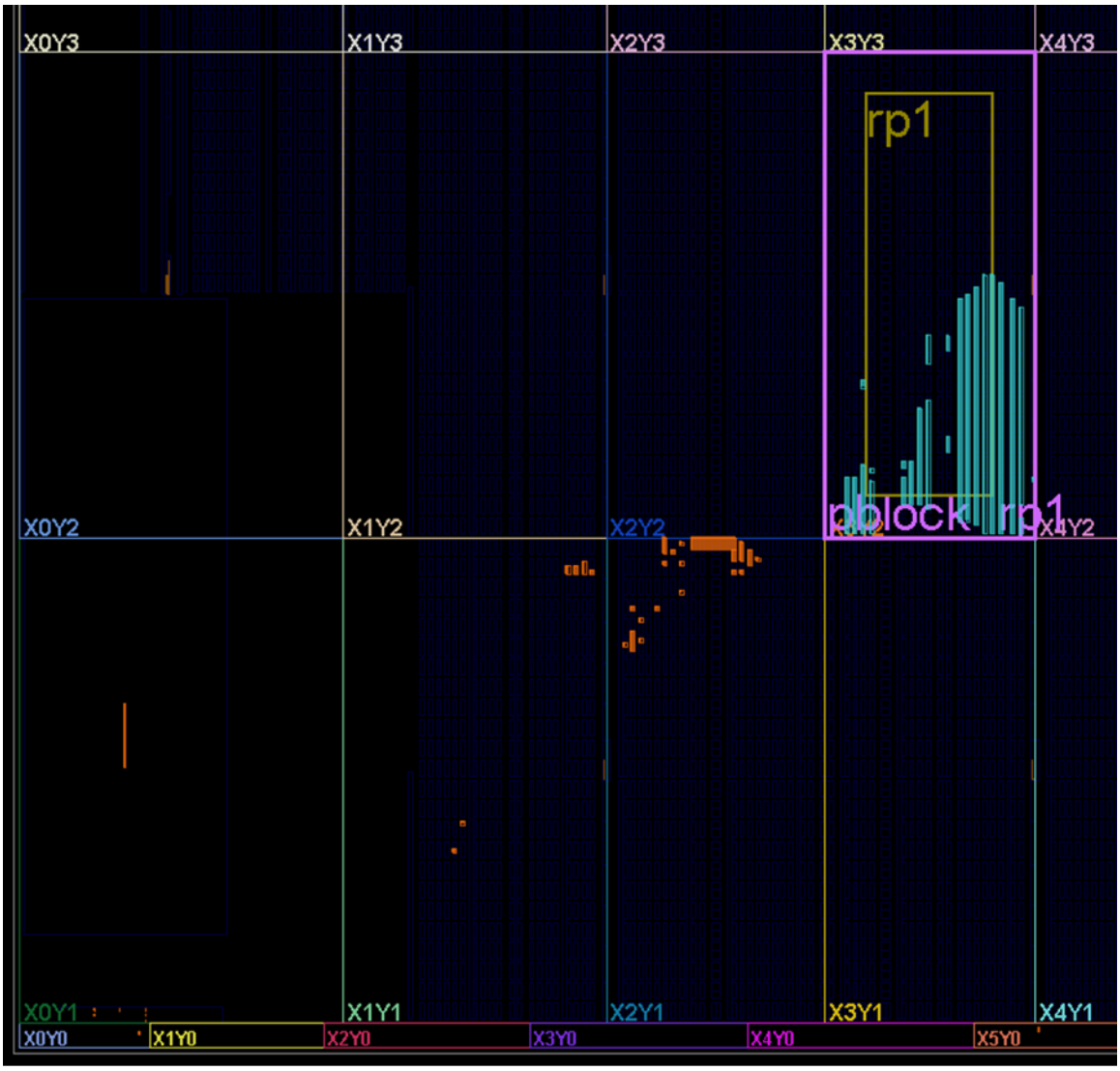
1. Open the routed checkpoint for `impl_1` to examine the parent configuration. This design image has the full static design along with the contents of rp1rm1 in the RP Pblock.



2. Open the routed design checkpoint for `child_gb_impl_1`. This has the full static design from the parent run plus a greybox as the reconfigurable module. The static logic is locked so it is shown in orange.

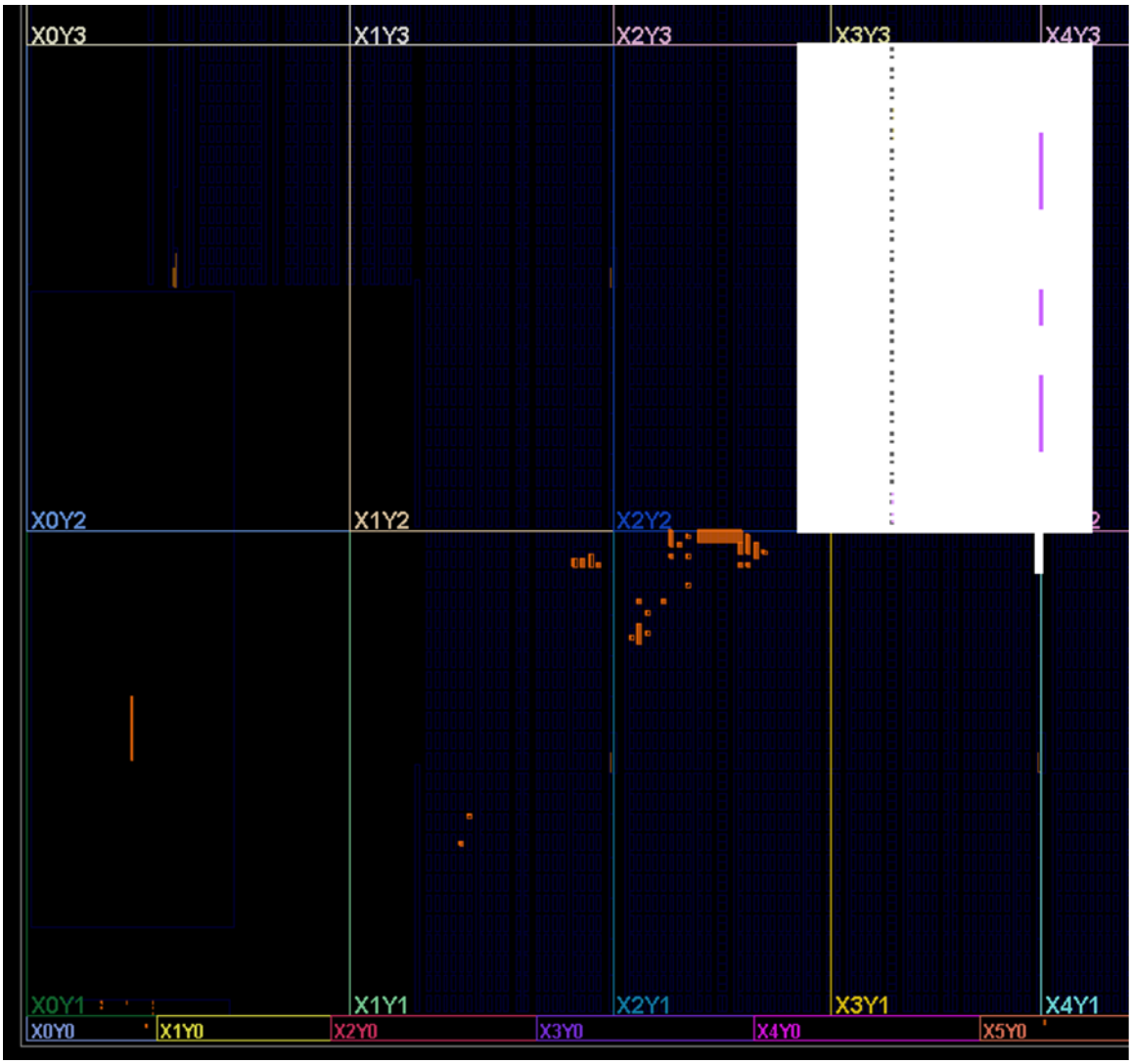


3. Open the routed design checkpoint for `child_0_impl_1`. This shows the orange locked static logic is reduced to a minimal set of logic and routing around the reconfigurable partition Pblock.



4. Finally, use the footprint visualization utility to see the expanded routing region that surrounds this Pblock.

```
select_objects [get_dfx_footprint -route -of_objects [get_cells
design_1_i/rp1]]
```



Compare this to the prior image which shows that the user-defined reconfigurable Pblock aligns to a single clock region. This highlighting shows the Pblock range expands left and right into neighboring clock regions to obtain routing resources to improve routability. It does not extend into clock regions above or below, as that would require adding two more complete clock regions to the partial bitstream, more than tripling its size.

Step 5: Hardware Validation

Hardware validation and interaction with the ChipScope debug cores can be done via XSDB.

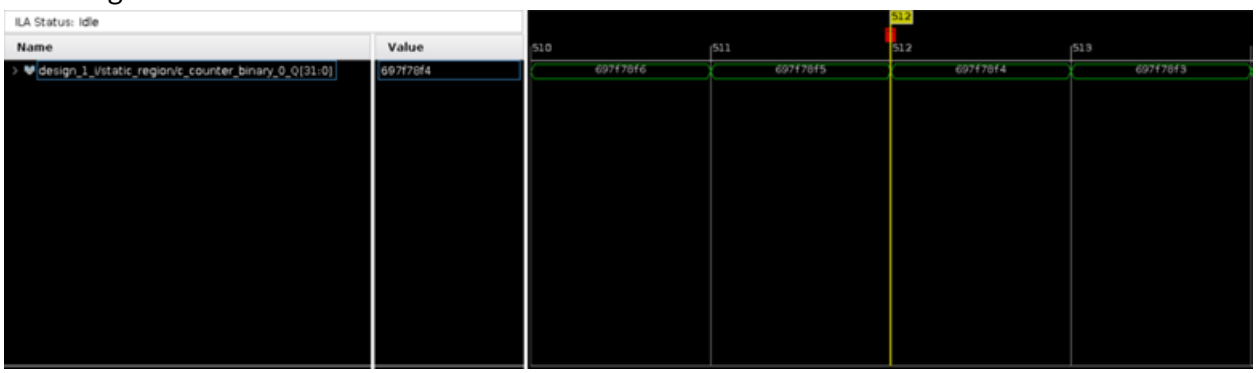
1. Download the full PDI from the parent implementation and poll the GPIOs from the static region and rp1rm1. The GPIO from the static region is connected to a 32-bit down counter and rp1rm1 is connected to a 32-bit up counter.

```
xsdb% targets
  1  Versal xcvc1902
     2  RPU (Reset)
        3  Cortex-R5 #0 (RPU PGE Reset)
        4  Cortex-R5 #1 (RPU PGE Reset)
     5  APU
        6  Cortex-A72 #0 (Power On Reset)
        7  Cortex-A72 #1 (Power On Reset)
     8  PPU
        9  MicroBlaze PPU (Sleeping)
    10  PSM
    11  PMC
    12  PL
xsdb% ta 5
xsdb% mrd -force 0x80210000
80210000:   8FE925FD

xsdb% mrd -force 0x80210000
80210000:   8560E3C5
```

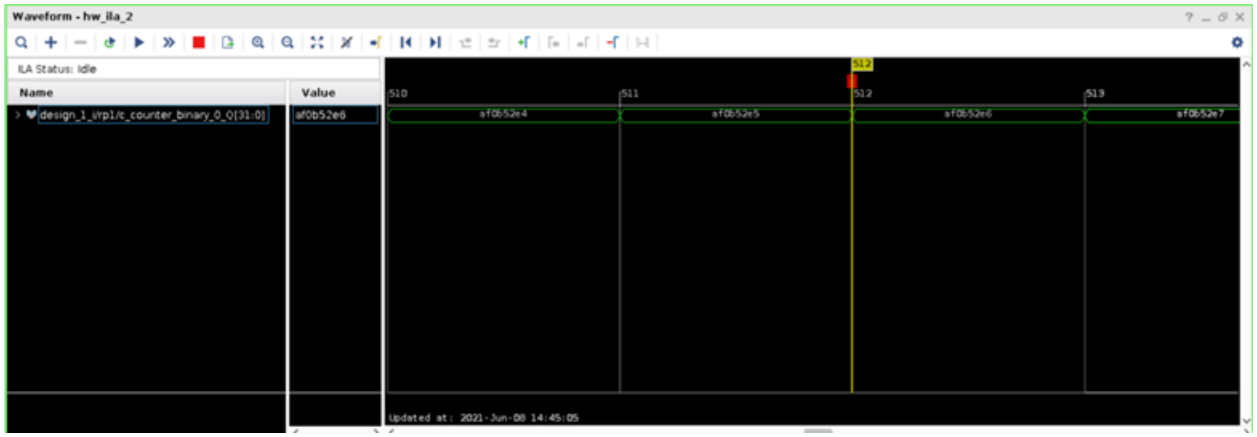
2. Read the GPIO in rp1rm1 to find the values counting up.

```
xsdb% mrd -force 0x80220000
80220000:   54FFC5A1

xsdb% mrd -force 0x80220000
80220000:   5B4B7E91
```

3. Observe the output of counters using ILAs in the static region and rp1rm1.
   - static region ILA



   - rp1rm1 ILA



4. Enable the Decouple signal before downloading the partial PDI for rp1rm2, which contains the count_down_vio module.

```
xsdb% mwr -force 0x80200000 0x01
xsdb% mrd -force 0x80200000
80200000:   00000001
```

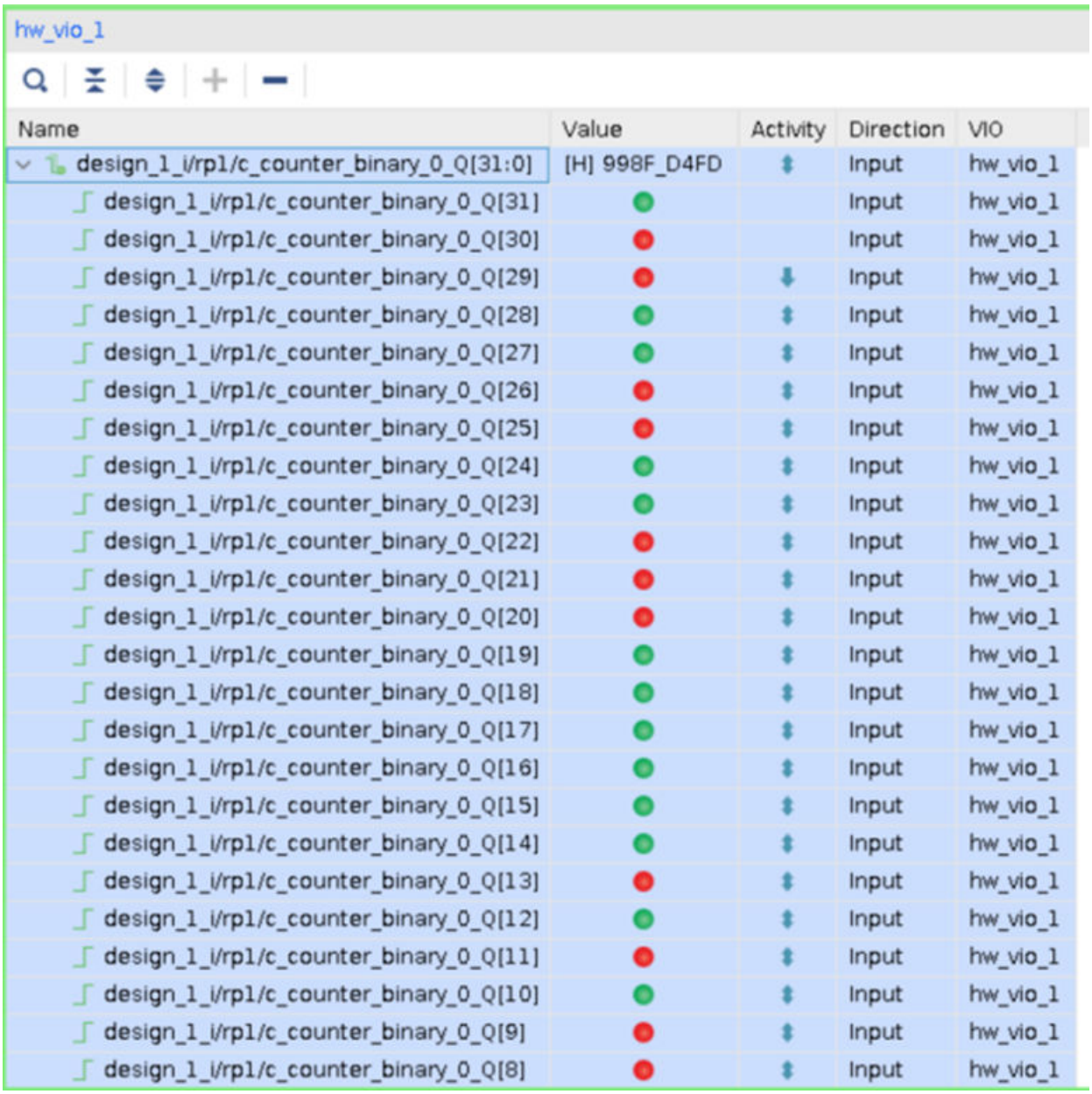
5. Download partial PDI for rp1rm2 (from child_0_impl_1) and release the Decouple signal.

```
xsdb% mwr -force 0x80200000 0x00
xsdb% mrd -force 0x80200000
80200000:   00000000
```

6. Observe the Down counter in rp1rm2.

```
xsdb% mrd -force 0x80220000
80220000:   6B5A4A7D

xsdb% mrd -force 0x80220000
80220000:   60EF29C0
```

7. Observe the rp1rm2 Down counter using VIO in AMD Vivado™ Hardware Manager.



8. Enable the Decouple signal before downloading the partial PDI for rp1rm3.

```
xsdb% mwr -force 0x80200000 0x01
xsdb% mrd -force 0x80200000
80200000:   00000001
```

9. Once rp1rm3 partial PDI is downloaded, disable the Decouple signal.

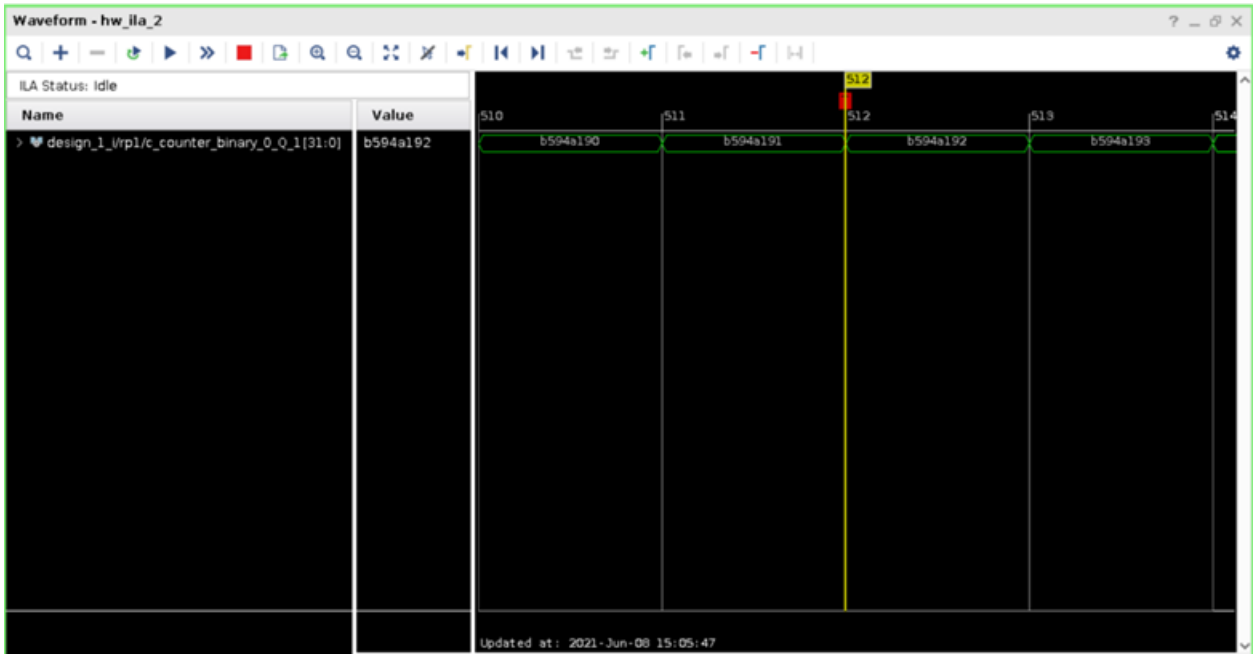
```
xsdb% mwr -force 0x80200000 0x00
xsdb% mrd -force 0x80200000
80200000:   00000000
```

10. rp1rm3 has both up and down counters connected to two separate ILAs.

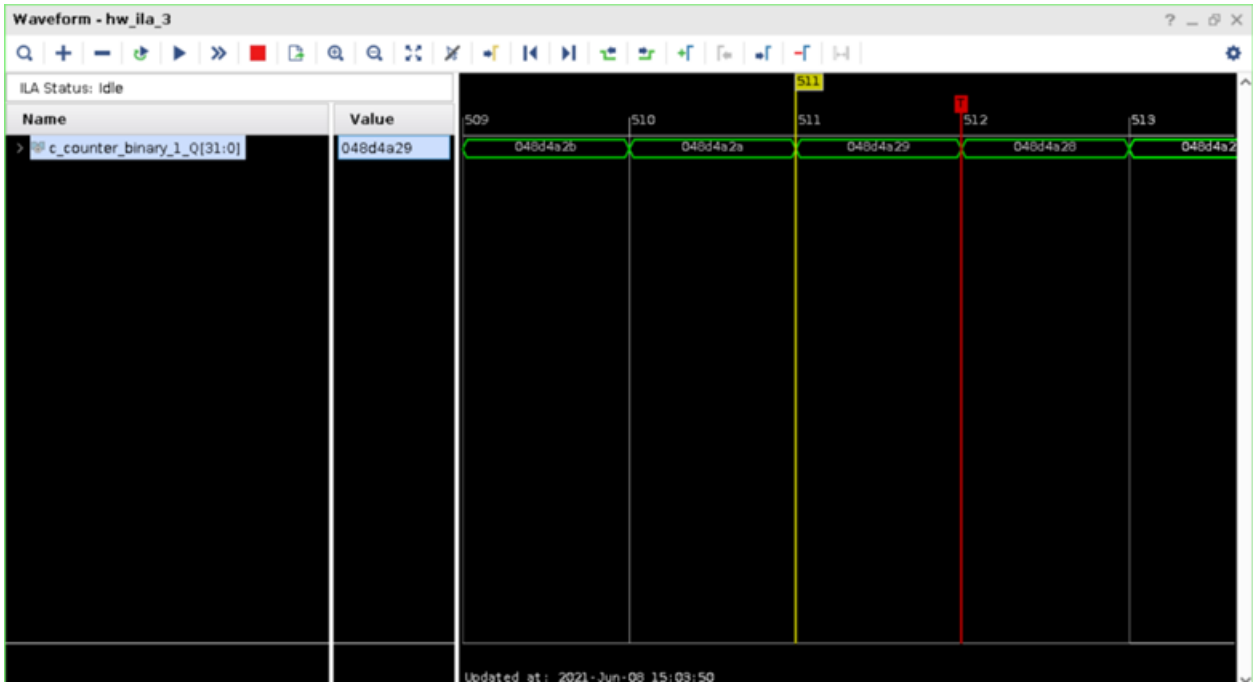
```
xsdb% mrd -force 0x80220000
80220000:   24776086

xsdb% mrd -force 0x80220000
80220000:   36D42755

xsdb% mrd -force 0x80230000
80230000:   A8B2F57F

xsdb% mrd -force 0x80230000
80230000:   9B470AEA
```

11. Observe the outputs in ILAs.

- Up Counter in rp1rm3



- Down Counter in rp1rm3



Lab 12 Conclusion

Abstract Shell for DFX

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

- Compile time for new Reconfigurable Modules is reduced for child runs, as Vivado implementation tools do not need to load or consider much of the information contained in the static part of the design.
- Static design information, including licensed IP, is hidden from view in an Abstract Shell, enhancing design security and reducing IP license requirements. This benefit is currently only viable for AMD UltraScale+™ and Versal non-project flow users,

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

Versal users working with the RTL-based Modular NoC solution, or Versal users who do not include NoC resources in dynamic regions can take advantage of both benefits by building Abstract Shell child runs that can be implemented outside the IPI project. This is what is required to avoid sharing the full parent environment (including sources) with secondary users. IP-only features such as SmartConnect IP usage, CIPS and AXI NoC IP usage and any other capability that cannot be resolved to a fixed-port netlist must be avoided in order to build Abstract Shells for non-project mode usage.

For more information on the Abstract Shell flow, see the *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909).

DFX Debug in a Versal Device

The capabilities of debug within DFX designs in Versal devices continue from the flow supported in UltraScale+. Once instantiated, debug cores can be automatically connected to the HSDP in Versal device like was done for BSCAN, as long as the ports on the Reconfigurable Partition boundary has been defined. In UltraScale+ this was done with a set of 12 explicitly named S_BSCAN ports (or tagged with properties), whereas in Versal, connection to the HSDP is accomplished via the NoC.

Appendix A

Additional Resources and Legal Notices

Finding Additional Documentation

Technical Information Portal

The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to https://docs.amd.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav, do the following:

- From the AMD Vivado™ IDE, select **Help → Documentation and Tutorials**.
- On Windows, click the **Start** button and select **AMD Adaptive SoC and FPGA Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

***Note*:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

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

- In DocNav, click the **Design Hubs View** tab.
- Go to the Design Hubs web page.

Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, refer to Support.

References

These documents provide supplemental material useful with this guide:

1. *Dynamic Function eXchange Controller IP LogiCORE IP Product Guide* (PG374)
2. *Dynamic Function eXchange Decoupler IP LogiCORE IP Product Guide* (PG375)
3. *Dynamic Function eXchange Bitstream Monitor IP LogiCORE IP Product Guide* (PG376)
4. *Dynamic Function eXchange AXI Shutdown Manager IP LogiCORE IP Product Guide* (PG377)
5. *Vivado Design Suite User Guide: Using Constraints* (UG903)
6. *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909)
7. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
8. *Silicon Labs CP210x USB-to-UART Installation Guide* (UG1033)
9. Vivado Design Suite DFX Tutorials for Versal Devices
10. Vivado Design Suite General DFX Tutorials
11. Vivado Design Suite DFX Tutorials for UltraScale+ Devices
12. AMD Documentation Navigator includes a Dynamic Function eXchange Design Hub that links these documents and other DFX-specific resources. It is also available through the AMD Adaptive Support site.
13. DFX Debug Tutorial
14. HSDP tutorial

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
07/29/2026 Version 2026.1	
Step 3: Create a Block Design Container	Updated the figure.

Section	Revision Summary
• Overview • Overview • Overview	Added Overview topics to chapters 5, 6, and 7.
Step 1: Extract the Tutorial Design Files	Updated step #3.

Please Read: Important Legal Notices

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes. THIS INFORMATION IS PROVIDED "AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AUTOMOTIVE APPLICATIONS DISCLAIMER

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

Copyright

© Copyright 2012-2026 Advanced Micro Devices, Inc. AMD, the AMD Arrow logo, Kintex, UltraScale, UltraScale+, Versal, Virtex, Vitis, Vivado, Zynq, and combinations thereof are trademarks of Advanced Micro Devices, Inc. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the US and/or elsewhere. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.