



TECHNISCHE
UNIVERSITÄT
WIEN



Institut für
Computertechnik
Institute of
Computer Technology

A REPORT ON

Dynamic Partial Reconfiguration Tutorial - The Manual

by

Andreas Dejmek

Vienna, Austria

February 2022

Table of Contents

Introduction	1
Hardware and Software Requirements	1
Tutorial Design Description	1
Lab 1: DFX Project Flow	3
Step 1: Extract the Tutorial Design Files	3
Step 2: Load Initial Design Sources	4
Step 3: Completing the Design with the DFX Wizard	8
Step 4: Synthesising and Implementing the current design	12
Step 5: Adding an additional reconfigurable module and corresponding configuration	16
Step 6: Creating and Implementing a greybox module	18
Step 7: Partially reconfiguring the FPGA	21
Conclusion	22
Lab 2: DFX Block Design	23
Step 1: Extract the Tutorial Design Files	23
Step 2: Load Initial Design Sources	24
Step 3: Completing the Design with the DFX Wizard	27
Step 4: Synthesising and Implementing the current design	31
Step 5: Adding an additional reconfigurable module and corresponding configuration	35
Step 6: Partially reconfiguring the FPGA	38
Conclusion	39
Lab 3: DFX Block Design with IP cores	40
Step 1: Extract the Tutorial Design Files	40
Step 2: Load Initial Design Sources	41
Step 3: Completing the Design with the DFX Wizard	44
Step 4: Synthesising and Implementing the current design	49

<i>Table of Contents</i>	iii
Step 5: Partially reconfiguring the FPGA	54
Conclusion	55
Bibliography	56

Acronyms

AXI Advanced Microcontroller Bus Architecture.

CPU Central Processing Unit.

DFX Dynamic Function eXchange.

FPGA Field Programmable Gate Array.

GPIO General Purpose I/O.

GUI Graphical User Interface.

I/O Input/Output.

IoT Internet of Things.

IP Intellectual Property.

LUT Lookup Table.

MPSoC Multiprocessor System on Chip.

OOC Out-of-Context.

PL Programmable Logic.

PR Partial Reconfiguration.

RM Reconfigurable Module.

RP Reconfigurable Partition.

Introduction

This tutorial covers the support of Dynamic Function eXchange (DFX) in the Vivado Design Suite release 2020.2 and complements the official tutorial for Dynamic Function eXchange from Xilinx [1].

[Lab 1: DFX Project Flow](#), [Lab 2: DFX Block Design](#) and [Lab 3: DFX Block Design with IP Cores](#) guides you through the project flow within the Vivado IDE, from the creation of the design using the DFX Wizard, to the synthesis, iteration runs, and the iteration of the design.

Hardware and Software Requirements

This tutorial requires that the 2020.2 Vivado Design Suite software release or later is installed.

The labs in this tutorial document only target the Xilinx ZCU102 Evaluation Kit [2]. It is a development platform for the Xilinx Zynq UltraScale+ Multiprocessor System on Chip (MPSoC). Unless specifically noted, this evaluation board is required to comply with the instructions in each lab.

Tutorial Design Description

The designs for the tutorial labs are available on a GitHub repository. Each lab in this tutorial has its own folder within the repository. To access the tutorial design files:

1. Download or git clone the [reference design files](#) from the GitHub repository [3].
2. Store the contents to any write-accessible location.

Lab 1: DFX Project Flow

The sample design used throughout this tutorial is called `dfx_project`. This design is very small, which helps to minimise data size and allows you to run the tutorial quickly and with minimal hardware requirements. It implements a simple LED control with two shift instances. This lab helps to illustrate that a partition definition applies to all instances of a partition type.

Lab 2: DFX Block Design

The sample design used throughout this tutorial is called `dfx_block_design`. It implements a simple block design using the Zynq UltraScale+ MPSoC and some additional resources. The reconfigurable partition is a simple RGB filter. It reads pixel data from an Advanced Microcontroller Bus Architecture (AXI) interface and filters out all channels except one. For example: The blue filter filters out everything except the blue channel. This lab is a reduced version of the project in [4].

Lab 3: DFX Block Design with IP Cores

The sample design used throughout this tutorial is called `dfx_ip_cores`. It implements a simple block design using the Zynq UltraScale+ MPSoC and some additional resources. This time, the reconfigurable partition is a Fourier Transform, with the reconfigurable modules also containing some Intellectual Property (IP) cores. Depending on the configured Reconfigurable Module (RM), a different Fourier Transform implementation is used for the transformation. The data exchange takes place via an AXI interface. This lab is a reduced version of the project in [5].

Lab 1

DFX Project Flow

This lab introduces the basic DFX flow for UltraScale and UltraScale+ devices. You will create a new project and set up all the sources and runs that define the structure of a DFX design. The design used in this lab has two instances of a RM. This shows the implications of partition definitions in the project flow.

Laboratory Objectives

By the end of this lab, you should be able to:

- Create a new project and prepare it for Dynamic Function eXchange
- Create new configurations for reconfigurable modules
- Synthesise a design bottom-up
- Create a valid floorplan
- Implement the configurations
- Create full and partial bitstreams
- Configure and partially reconfigure an Field Programmable Gate Array (FPGA)

Step 1: Extract the Tutorial Design Files

1. Download or git clone the [reference design files](#) from the GitHub repository [3].
2. Store the contents to any write-accessible location.
3. In the stored files, navigate to `\dfx_project`.

Step 2: Load Initial Design Sources

The first step in any DFX design flow (project-based or otherwise) is to define the parts of the design that will be marked as reconfigurable. This is done via context menus in the Hierarchical Source View in project mode. These steps will walk through initial project creation through the 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 Vivado IDE and select Create Project, then click **Next**.
3. Select the `<Extract_Dir>` as the Project location. Keep the Project name as `project_1`, and check the Create project subdirectory option. Click **Next**.
4. Select RTL Project and ensure the Do not specify sources checkbox is unchecked, then click **Next**.
5. Click the **Add Directories** button and select these Sources directories to be added to the design: Keep the remaining settings to default.
 - `<Extract_Dir>\Sources\hdl\top`
 - `<Extract_Dir>\Sources\hdl\shift_right`
6. Click **Next** to get to the Add Constraints window, then select this file:
 - `<Extract_Dir>\Sources\xdc\top_io.xdc`
7. Click **Next** to choose the part. In the Part selector, click on **Boards** and choose the Zynq Ultra-Scale+ ZCU102 Evaluation Board. Then click **Next** and then **Finish** to complete project creation. The Sources window shows a standard hierarchical view of the design.

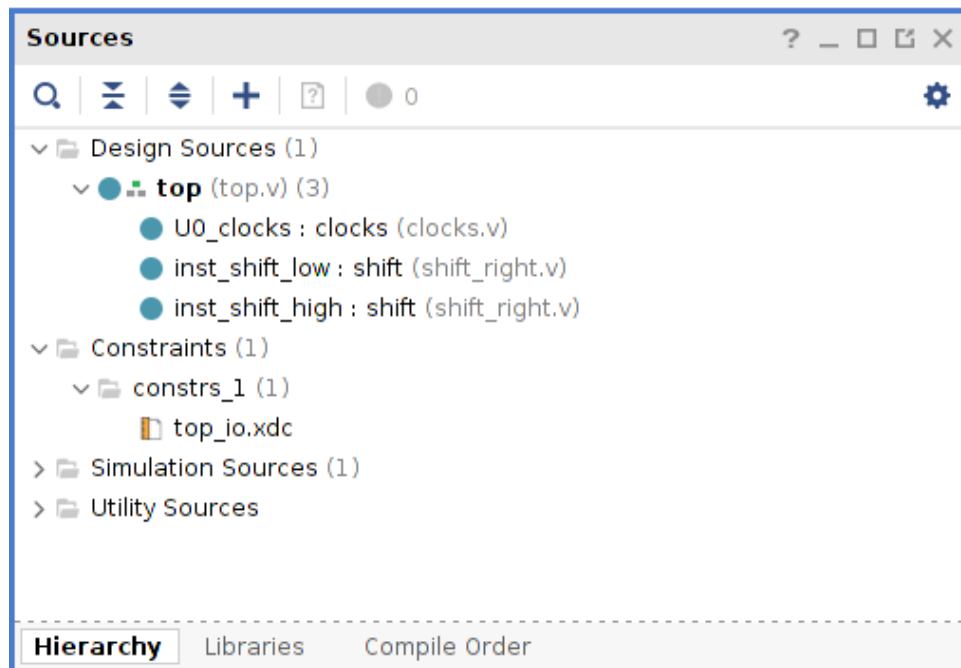


Figure 1.1: Sources view after project creation

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 action prepares the project for the DFX design flow.

Note: Once the Dynamic Function eXchange for a Vivado project is enabled, it cannot be undone, so Xilinx recommends archiving your project prior to selecting this option. The only way to return to a project that is not partially reconfigurable is to create a new one [6].

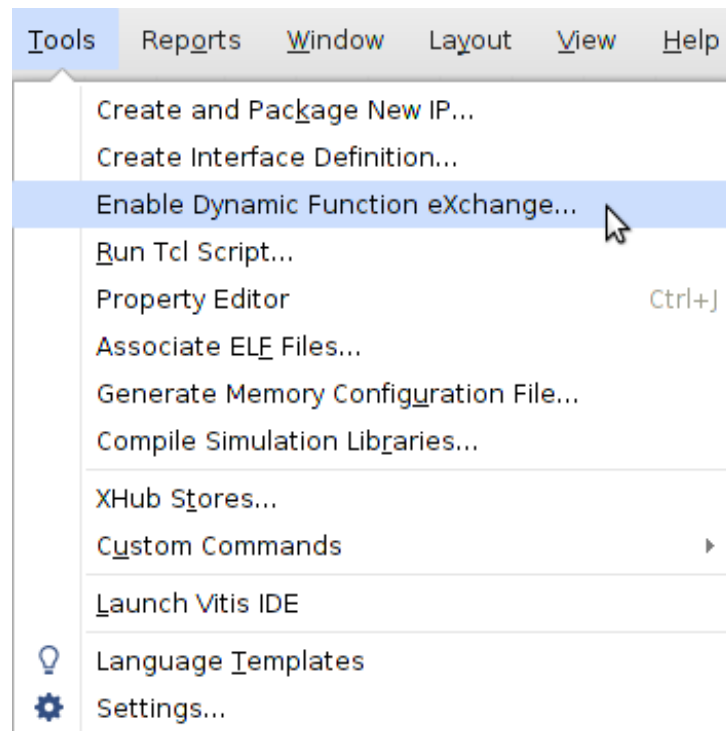


Figure 1.2: Enabling Dynamic Function eXchange

In the following pop-up window, 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 will define both `shift` instances as reconfigurable partitions in the design.

Since each instance has come from the same RTL source, they are logically identical. Bottom-up synthesis is required to keep this module separated from top. Bottom-up synthesis refers to a synthesis flow in which each module has its own synthesis project. This generally involves turning off automatic Input/Output (I/O) buffer insertion for the lower level modules to ensure no optimisation occurs across the module boundaries. To synthesise the top-level, a netlist with a black box for each reconfigurable partition must be available. This requires the top-level synthesis to have module or entity declarations for the partitioned instances, but no logic. The top-level synthesis also infers or instantiates I/O buffers on all top-level ports.

10. In the dialogue box that appears, enter names for 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 a suitable 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**.

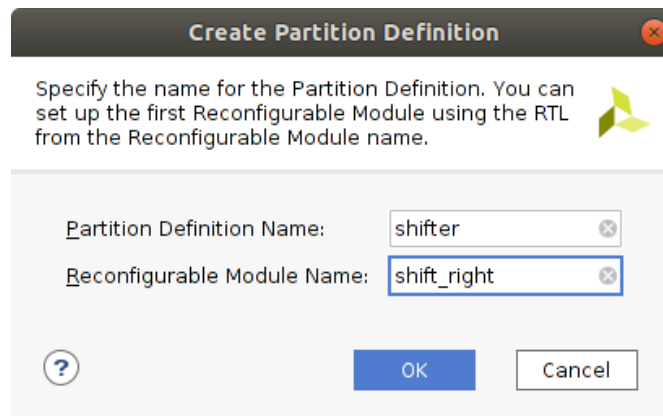


Figure 1.3: Creating the shifter partition definition

The Sources view has now changed slightly, with both shift instances now shown with a yellow diamond, indicating they are partitions. You will 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 synthesising the `shift_right` module.

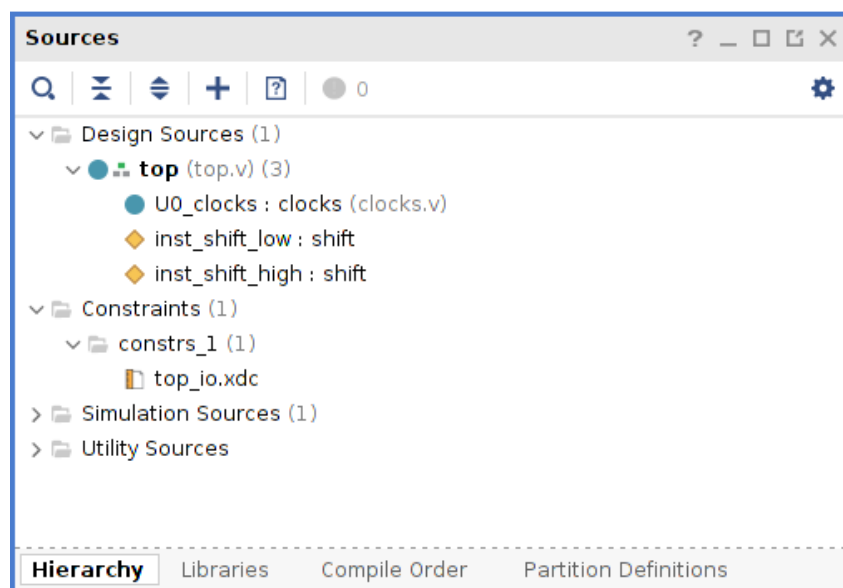


Figure 1.4: Sources view after defining shifter partition

At this point, more reconfigurable module sources may be added. This is done via the Dynamic Function eXchange Wizard.

Note: Once the partitions are defined, any additional RMs must be added via the DFX Wizard, and the management of RM sources, configurations, and runs must also be done via this wizard.

Step 3: Completing the Design with the DFX Wizard

1. Launch the DFX 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. Note that they would need to be scoped to the level of hierarchy for this partition.

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

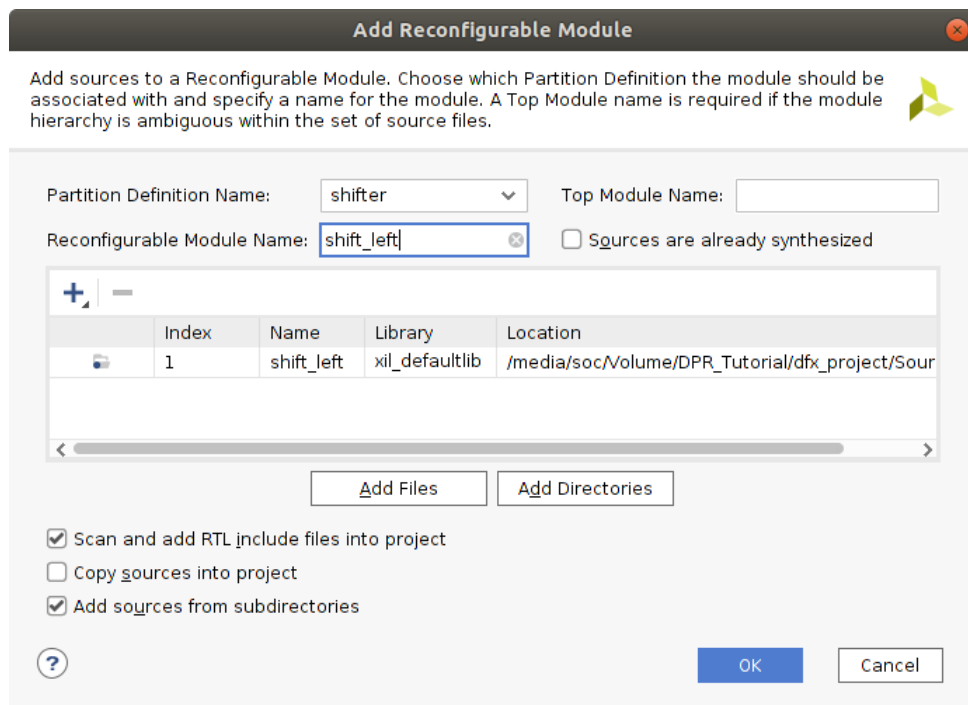


Figure 1.5: Add a new RM with the DFX Wizard

Two reconfigurable modules are now available for the `shifter` reconfigurable partition.

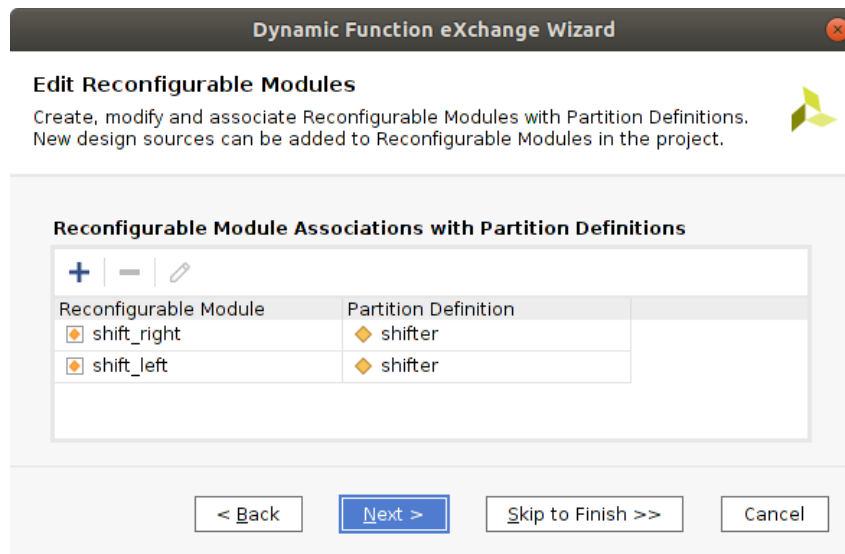


Figure 1.6: DFX Wizard with two RMs defined

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

4. Let the Wizard create the configurations by selecting the **automatically create configurations** link.

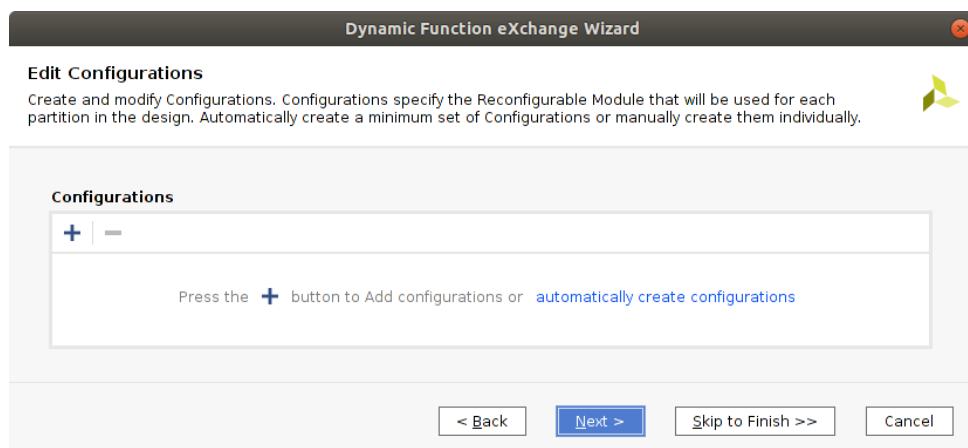


Figure 1.7: DFX Wizard Configurations page

After selecting this option, the minimum set of two configurations has been created. Each shift instance has been given `shift_right` in the first configuration and `shift_left` in the second configuration. Note that the Configuration Name is editable – in the example below, the names have been updated to `config_right` and `config_left` to reflect the reconfigurable modules contained within each one.

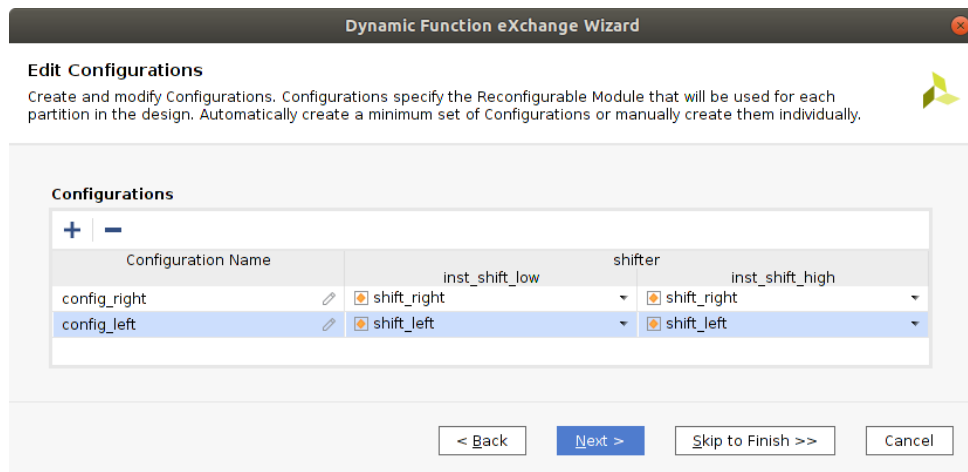


Figure 1.8: Auto-generated minimum set of configurations

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.

- 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.
- Click on the **automatically create configuration run** link to populate the Configuration Runs page with the minimum set of runs.

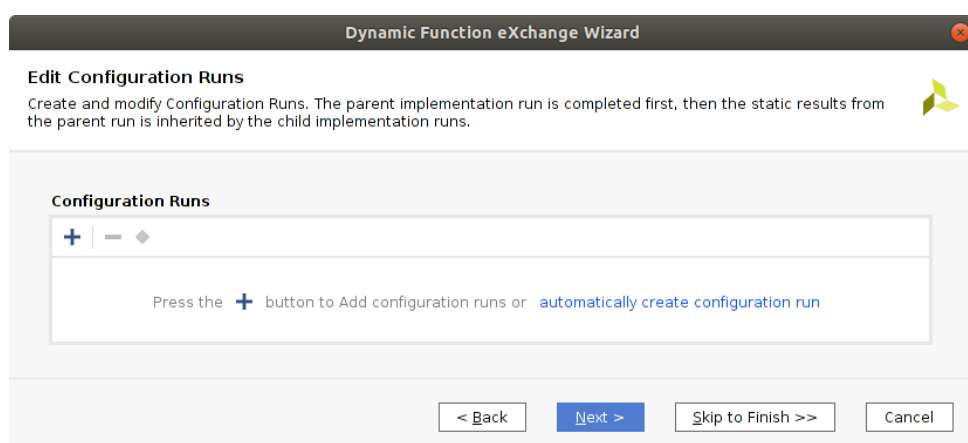


Figure 1.9: DFX Wizard Configuration Runs page

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 constraints 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.

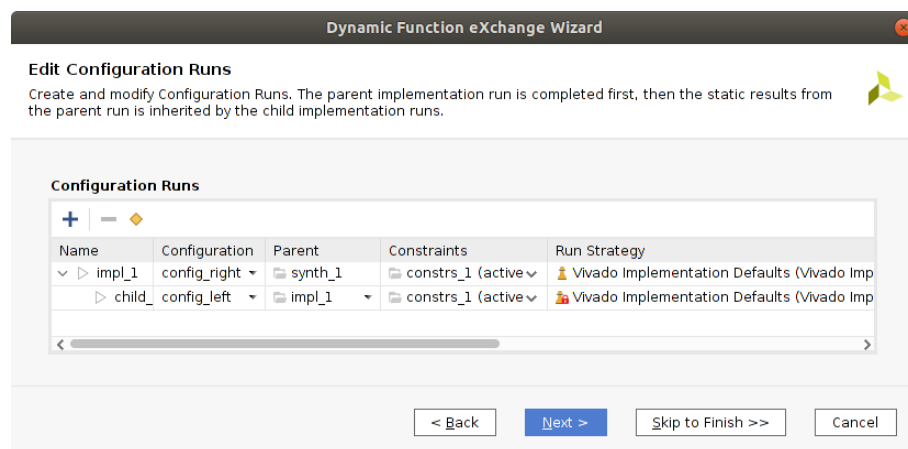


Figure 1.10: Auto-generated minimum set of configuration runs

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

Note: 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, 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.

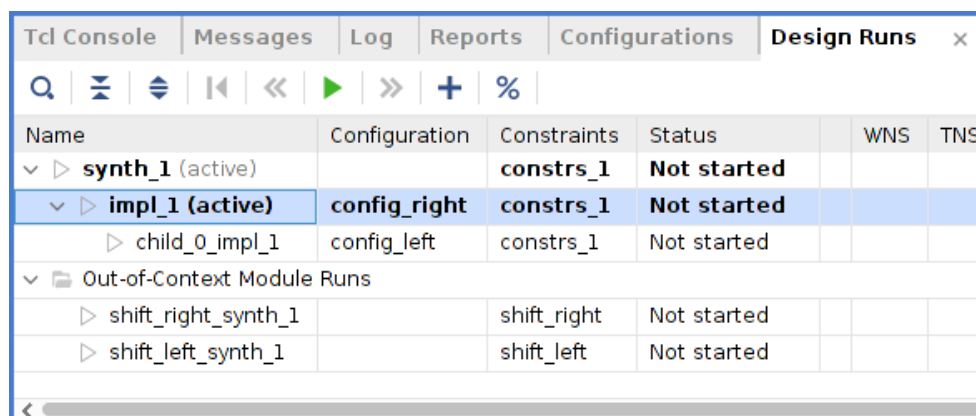


Figure 1.11: Design Runs window showing all synthesis and implementation ready to launch

Step 4: Synthesising and Implementing the current design

With the design from above open in the 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 in Run Synthesis or Run Implementation pulls the design through only these runs, as well as the Out-of-Context (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**. This action will open a new window, where you can specify the launch options. For this tutorial, we will keep the remaining settings to default. The number of jobs allows you to specify how many Central Processing Unit (CPU) cores should be used for this launch. Then click **OK**.

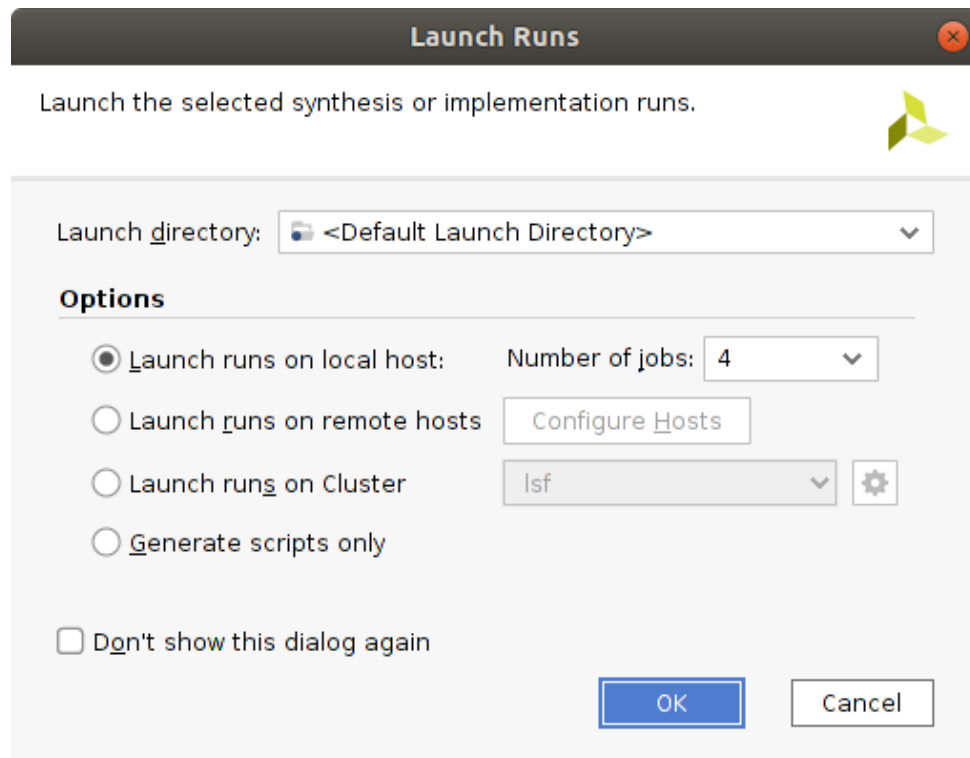


Figure 1.12: Launch options window

Now this will synthesise all OOC modules, followed by synthesis of the top-level design. This is no different from any design with OOC modules (IP or otherwise).

2. After the synthesis is successfully completed, select **Open Synthesized Design**.

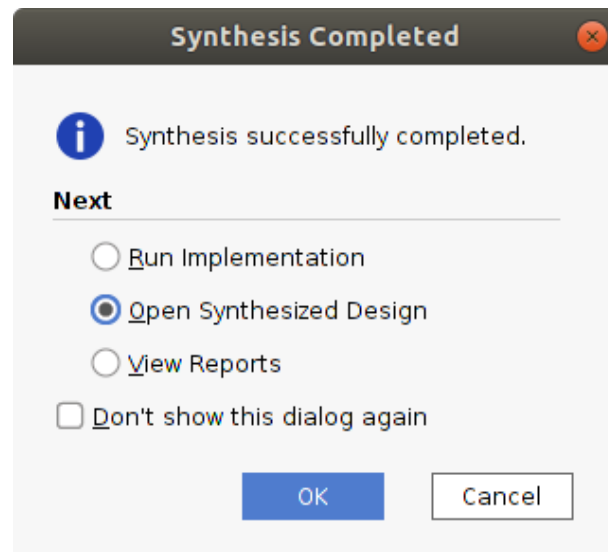


Figure 1.13: Synthesis successfully completed

Now the post-synthesis design will open. Since no Pblocks existed in the design sources, they can be created in this step. This can be done by right-clicking in an `inst_shift` instance in the design hierarchy to select **Floorplanning** → **Draw Pblock**. Each instance will require its own unique Pblock.

Note: If you look at the statistic page of the cell properties of an instance, you can see what resources are needed by that instance. Since all partially configurable modules must fit into this Pblock, it must contain sufficient resources.

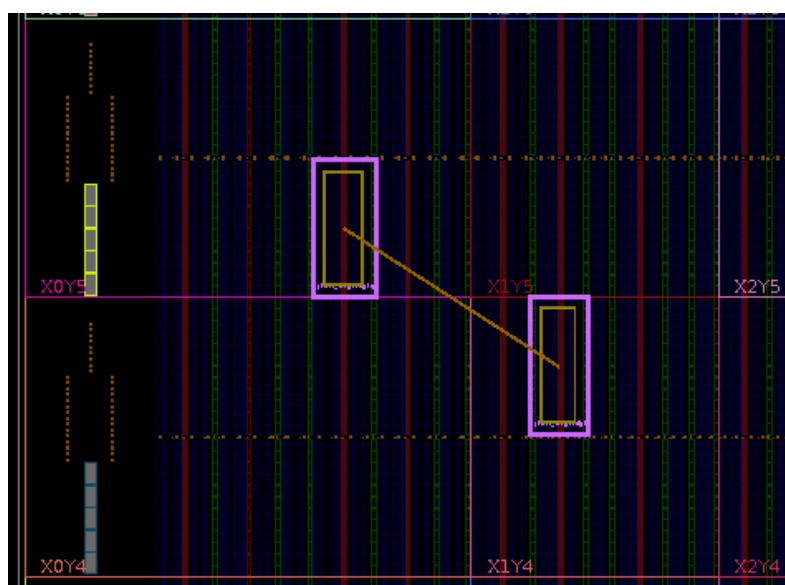


Figure 1.14: Floorplan with two reconfigurable partitions (Pblocks)

3. Select one of the two Pblocks in the floorplan and note its properties. The last property listed is `SNAPPING_MODE`, a property specific to DFX. Note that this option has been enabled in the Pblocks xdc.
4. 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.

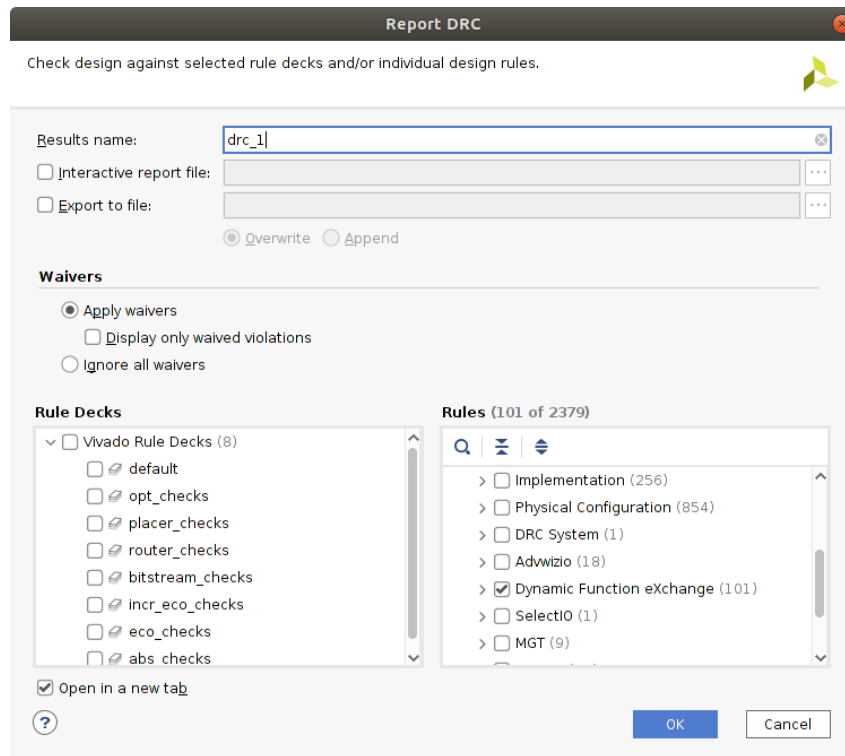


Figure 1.15: Checking DFX DRCs

If the DRC checks have been reported some issues, fix them before moving on. Note that both modules will require BRAM resources, and remember that `SNAPPING_MODE` will resolve any errors related to horizontal or vertical alignment. For certain devices, hints can be given on how to improve the quality of the specified Pblocks. These can be ignored for this simple design.

TIP: Run DFX Design Rule Checks early and often.

The created Pblocks can be saved to a separate xdc file, so they can be used in different designs. Saving the current constraints to the target project may cause your synthesis to go out-of-date. To avoid re-running synthesis, you can force the design up-to-date by selecting the run in the Design Tab, right-clicking, and selecting **Force Up-to-Date**.

5. In the Flow Navigator, select **Run Implementation** to run place and route on all configurations. Again, a window will open, where you can specify the launch options.

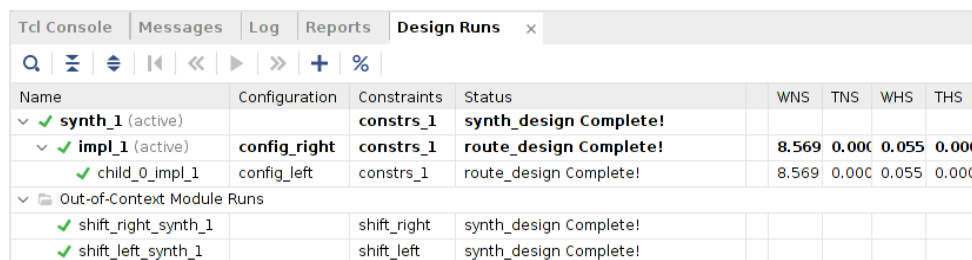
This action runs implementation first for `impl_1` and then for `child_0_impl_1`. Behind the scenes, Vivado takes care of all the details. In addition to running place and route for the two runs with all the DFX requirements in place, it does a few more tasks specific to DFX. After `impl_1` completes, 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 only specific configuration runs are desired, these can be individually selected within the Design Runs window. Note that 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.

6. When implementation completes, click **Cancel** in the resulting pop-up dialogue.



Name	Configuration	Constraints	Status	WNS	TNS	WHS	THS
✓ synth_1 (active)		constrs_1	synth_design Complete!				
✓ impl_1 (active)	config_right	constrs_1	route_design Complete!	8.569	0.000	0.055	0.000
✓ child_0_impl_1	config_left	constrs_1	route_design Complete!	8.569	0.000	0.055	0.000
Out-of-Context Module Runs							
✓ shift_right_synth_1		shift_right	synth_design Complete!				
✓ shift_left_synth_1		shift_left	synth_design Complete!				

Figure 1.16: All configurations routed

At this point, there are two steps remaining. The first is running Partial Reconfiguration (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.

7. In the Flow Navigator, click **Generate Bitstream**. This action launches bitstream generation on the active parent runs, 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 Graphical User Interface (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` then click **OK**.
4. Enter `shift_right_slow` for the Reconfigurable Module Name and then click **OK** and **Next**.

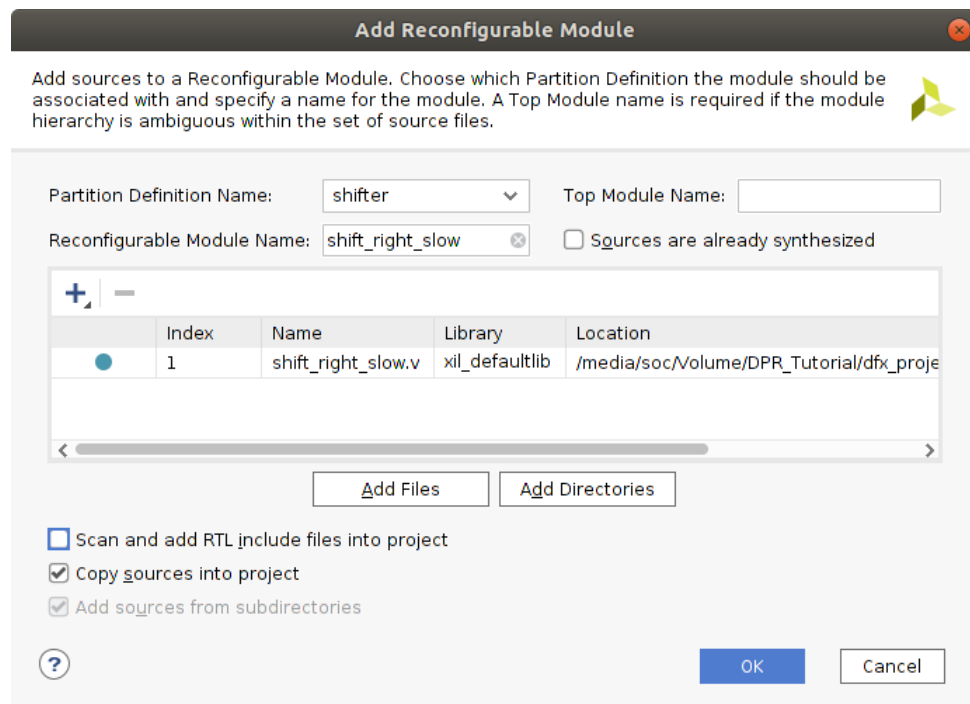


Figure 1.17: Adding a new reconfigurable module

Note that 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 will recreate all configurations and remove all existing results.

5. Create a new configuration by clicking the + button, entering the name `config_right_slow`, then hitting **Enter**. Select `shift_right_slow` for each reconfigurable partition instance.

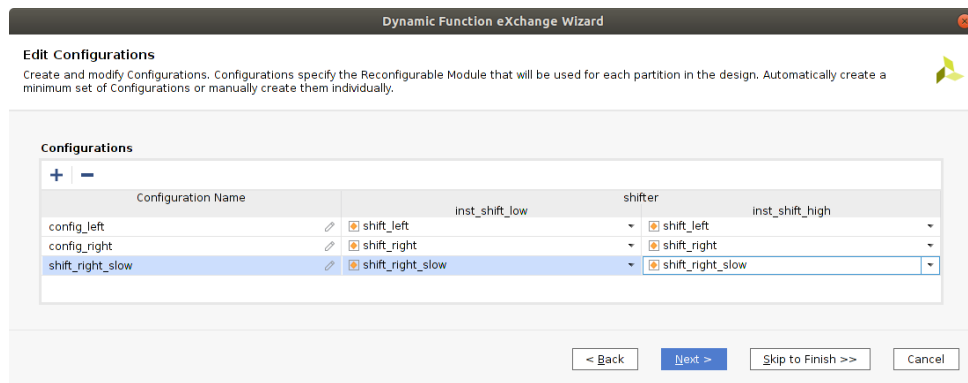


Figure 1.18: Creating new configuration

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` – this simply matches the existing convention
 - Parent: `impl_1` – this makes this configuration a child run of the existing parent run
 - Configuration: `config_right_slow` – this is the one with the new RMs that was just defined
7. Click **OK** to add the new Configuration Run.

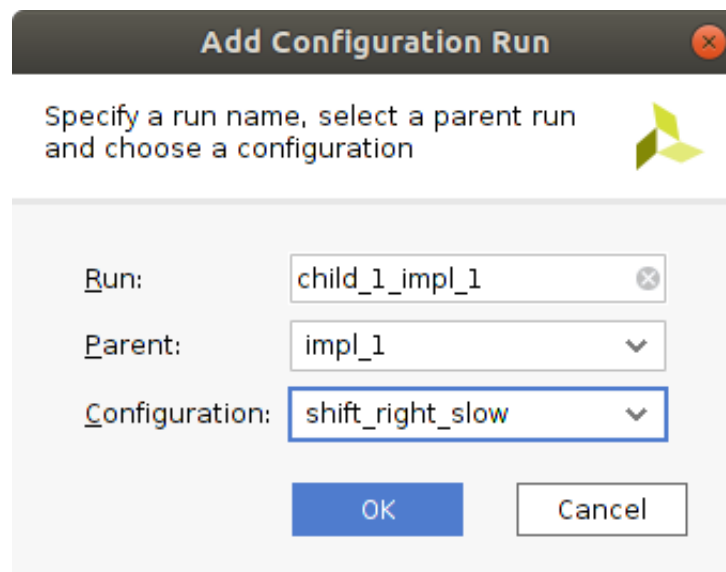


Figure 1.19: Creating a new Configuration Run

This new configuration, as a child of the existing `impl_1`, will reuse the static design implementation results, just 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.

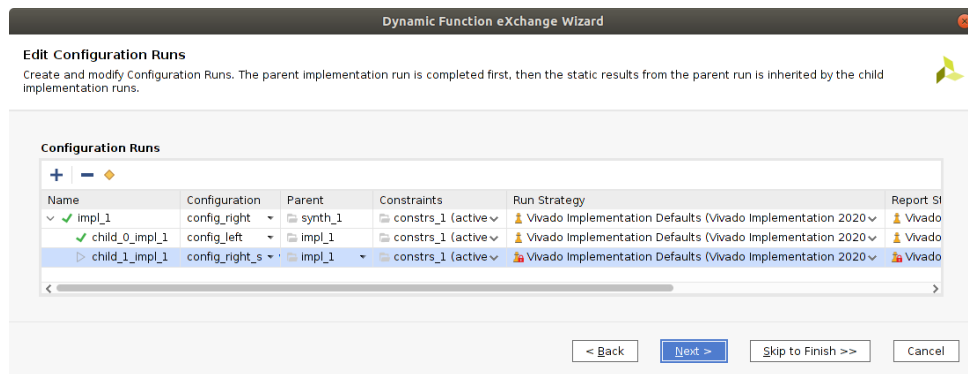


Figure 1.20: New configuration added as a new child run

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

Design Runs							
Name	Configuration	Constraints	Status	WNS	TNS	WHS	THS
✓ synth_1 (active)		constrs_1	synth_design Complete!				
✓ impl_1 (active)	config_right	constrs_1	write_bitstream Complete!	8.569	0.000	0.055	0.000
✓ child_0_impl_1	config_left	constrs_1	write_bitstream Complete!	8.569	0.000	0.055	0.000
▶ child_1_impl_1	config_right_slc	constrs_1	Not started				
Out-of-Context Module Runs							
✓ shift_right_synth_1		shift_right	synth_design Complete!				
✓ shift_left_synth_1		shift_left	synth_design Complete!				
▶ shift_right_slow_synth_1		shift_right_slow	Not started				

Figure 1.21: New OOC synthesis run and configuration run added

9. Select this new child implementation run, right-click and select **Launch Runs**. This will run OOC synthesis on the `shift_right_slow` module, then implement this module within the context of the locked static design.
10. After the implementation run is successfully completed, generate the bitstreams. Therefore, select the new child implementation run, right-click and select **Generate Bitstream**. Leave all checkboxes unchecked and click **OK**.

Step 6: Creating and Implementing a greybox module

For some designs, the desired initial configuration of the device may be an image with no function resident in a reconfigurable partition. Or perhaps there are no reconfigurable modules available to implement yet. A greybox configuration can be used to implement just the static design without real RM netlists available.

A greybox is a module that starts off as a blackbox, but then has Lookup Tables (LUTs) automatically inserted for all ports. 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. Training scripts are available to create timing budgets for this greybox image, optimising the implementation

results of the static design. A configuration with greybox RMs can be the parent run, but this is only recommended when no other RMs exist and/or when budgeting constraints are used to optimise the PR interface placement.

1. Open the Dynamic Function eXchange Wizard and move to the Configurations page – no new reconfigurable modules need to be defined in this case, as this is a dedicated feature. Create a new configuration, enter a name of `config_greybox`, and enter `<greybox>` for each reconfigurable partition instance.

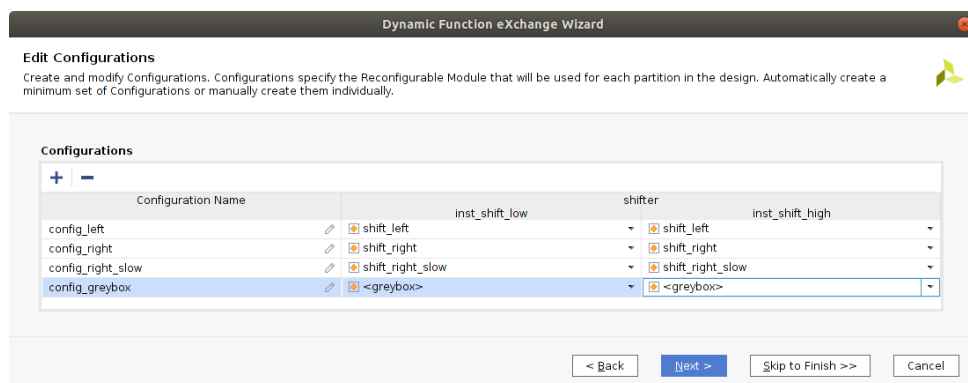


Figure 1.22: Adding the new greybox configuration

2. Click **Next** to get to the Configuration Runs page, then create another new configuration run, this time for the greybox configuration.
 - Run: `impl_greybox`
 - Parent: `synth_1` – this makes this configuration a new parent, starting from the synthesised top-level design
 - Configuration: `config_greybox` – the RMs consist only of LUT tie-offs

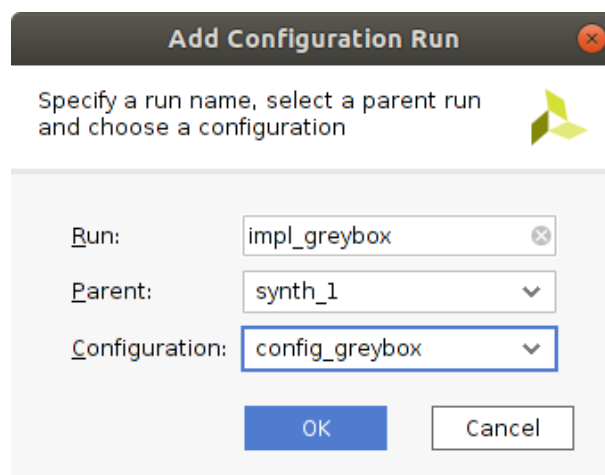


Figure 1.23: Creating new greybox Configuration Run

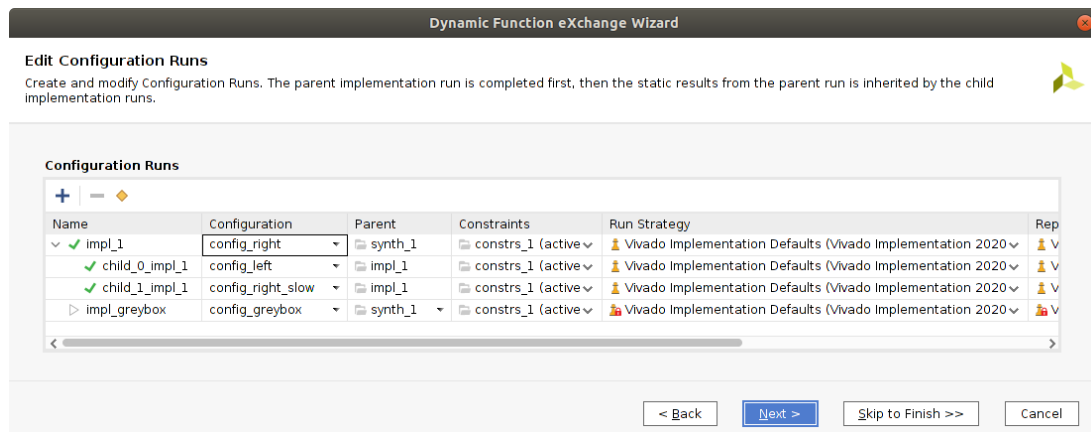


Figure 1.24: Creating an independent greybox Configuration Run

- Click **Next** then **Finish** to create this new run. Now there are four implementation runs and three Out-of-Context runs shown in the Design Runs window. Note that the greybox module does not require synthesis – it is an embedded feature in the DFX solution.

Tcl Console Messages Log Design Runs x								
Name	Configuration	Constraints	Status	WNS	TNS	WHS	THS	
✓ synth_1 (active)		constrs_1	synth_design Complete!					
✓ impl_1 (active)	config_right	constrs_1	write_bitstream Complete!	8.569	0.000	0.055	0.000	
✓ child_0_impl_1	config_left	constrs_1	write_bitstream Complete!	8.569	0.000	0.055	0.000	
✓ child_1_impl_1	config_right_slow	constrs_1	route_design Complete!	8.569	0.000	0.055	0.000	
▶ impl_greybox	config_greybox	constrs_1	Not started					
Out-of-Context Module Runs								
✓ shift_right_synth_1		shift_right	synth_design Complete!					
✓ shift_left_synth_1		shift_left	synth_design Complete!					
✓ shift_right_slow_synth_1		shift_right_slow	synth_design Complete!					

Figure 1.25: Greybox implementation ready to run

At this point the greybox configuration can be implemented.

- Select the `impl_greybox` design run, right-click and select **Launch Runs**. The Flow Navigator will not launch this run as it is not the active parent.

IMPORTANT: Because `impl_1` and `impl_greybox` are both parents, their static design results will be different, and their resulting bitstreams will **NOT** be compatible in hardware. Only bitstreams derived from a single parent (and subsequently confirmed using PR Verify) should ever be delivered through DFX to a device.

- To also create the bitstream again, select the `impl_greybox` design run, right-click and select **Generate Bitstream**.

Step 7: Partially reconfiguring the FPGA

The current design targets only the Xilinx ZCU102 Evaluation Kit.

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 and pick the target device, here xczu9_0.
4. Navigate to the bitstream folder to select `top.bit`, then click **OK** to program the device.

You should now see the bank of General Purpose I/O (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.

1. Select **Program device** on the green banner again. Navigate to the bitstream folder to select `inst_shift_high_shift_left_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.

2. Select **Program device** on the green banner again. Navigate to the bitstream folder to select `inst_shift_low_shift_left_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 bit files in the `impl_1` folder to return to the original configuration.

Conclusion

This concludes Lab 1. By now you should be able to:

- Created a new project and prepared it for Dynamic Function eXchange
- Created two configurations for reconfigurable modules
- Synthesised a design bottom-up to prepare for DFX implementation
- Created a valid floorplan for a DFX design
- Implemented these two configurations
- Created full and partial bitstreams
- Configured and partially reconfigured an FPGA

The Dynamic Function eXchange Project Flow offers a high degree of flexibility and allows users to manage their design environment and explore different options. Users need to be careful when tracking implementation results and bitstreams to ensure that only compatible bitstreams created from a single fixed static image are downloaded to the target device.

Lab 2

DFX Block Design

This lab provides an introduction to how the DFX flow can be used in a project design based on a block design. You will run an automated script that creates a new project and set up all the sources. Then you will modify the top-level design to create a DFX design. After that, you will finally set up all the runs that define the structure of a DFX design. The design used in this lab has one instance of a RM.

Laboratory Objectives

By the end of this lab, you should be able to:

- Create a new project based on a block design using a TCL script and prepare it for Dynamic Function eXchange
- Create new configurations for reconfigurable modules
- Synthesise a design bottom-up
- Create a valid floorplan
- Implement the configurations
- Create full and partial bitstreams

Step 1: Extract the Tutorial Design Files

1. Download or git clone the [reference design files](#) from the GitHub repository [3].
2. Store the contents to any write-accessible location.
3. In the stored files, navigate to `\dfx_block_design`.

Step 2: Load Initial Design Sources

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

1. Extract the design from the archive. The `dfx_block_design` data directory is referred to in this tutorial as the `<Extract_Dir>`.
2. Open the Vivado IDE and run the automated project creation script. This is done by calling `source ./scripts/create_hw_project.tcl` in the TCL Console.

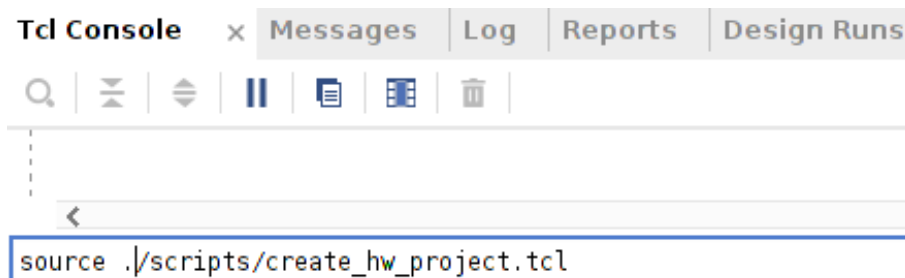


Figure 2.1: TCL Console project creation

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

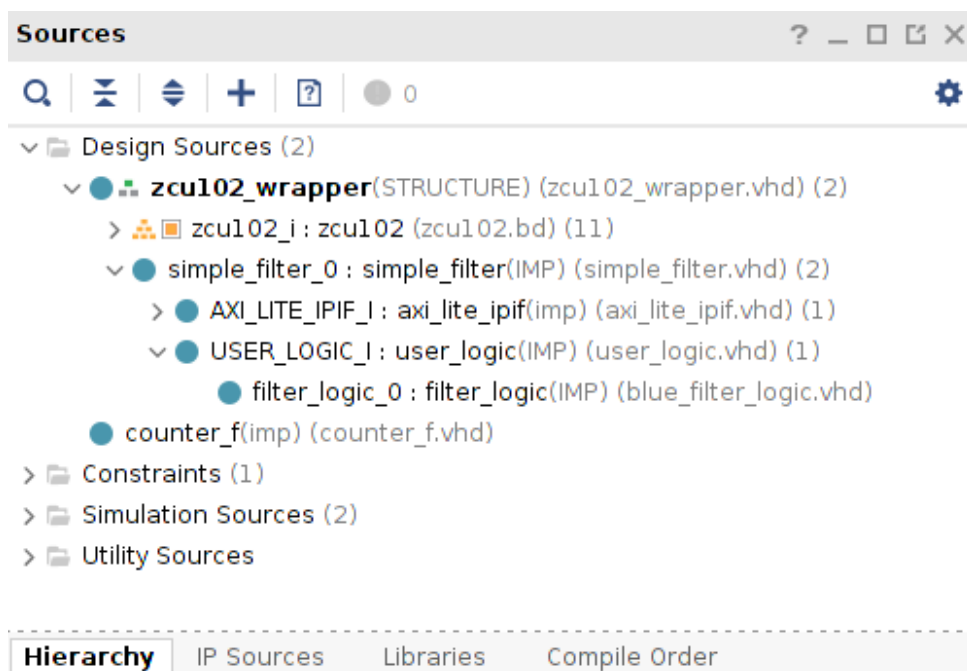


Figure 2.2: Sources view after project creation

3. Before we can continue, the created block design must be generated. In the Flow Navigator, click **Generate Block Design**. Now this will synthesise all block design OOC modules.

Note: Since Vivado does not offer DFX support for block designs in the latest version of the Vivado Design Suite, design adjustments must be made. The partially reconfigurable partition cannot be included in the block design, so the PR partition must be connected in the top-level design. Therefore, the required interfaces have to be exported to the top-level design. In the top-level design (here the `zcu102_wrapper.vhd` file) the partition has to be manually integrated. This step has already been taken in this lab.

4. Select **Tools** → **Enable Dynamic Function eXchange**. This action prepares the project for the DFX design flow.

Note: Once the Dynamic Function eXchange for a Vivado project is enabled, it cannot be undone, so Xilinx recommends archiving your project prior to selecting this option. The only way to return to a project that is not partially reconfigurable is to create a new one [6].

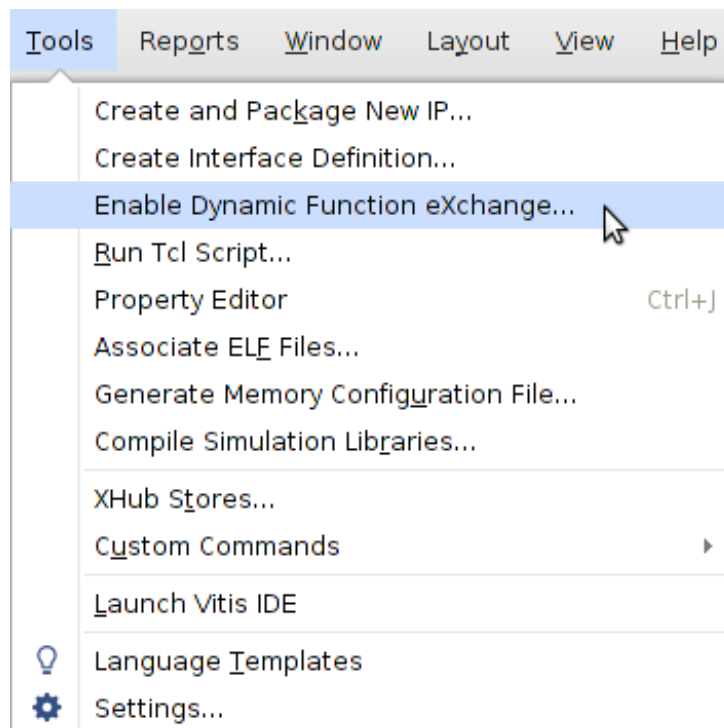


Figure 2.3: Enabling Dynamic Function eXchange

In the following pop-up window, click **Convert** to turn this project into a DFX project.

5. In the Sources view, right-click on the `filter_logic_0` instances and select the **Create Partition Definition...** option. This action will define the `filter_logic` instance as a reconfigurable partition in the design.

Bottom-up synthesis is required to keep this module separated from top. Bottom-up synthesis refers to a synthesis flow in which each module has its own synthesis project. This generally involves turning off automatic I/O buffer insertion for the lower level modules to ensure no optimisation occurs across the module boundaries. To synthesise the top-level, a netlist with a black box for each reconfigurable partition must be available. This requires the top-level synthesis to have module or entity declarations for the partitioned instances, but no logic. The top-level synthesis also infers or instantiates I/O buffers on all top-level ports.

6. In the dialogue box that appears, enter names for 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 a suitable name, such as `filter_logic`. The reconfigurable module refers to this specific RTL instance, so give it a name that references its functionality, such as `blue_filter`, then click **OK**.

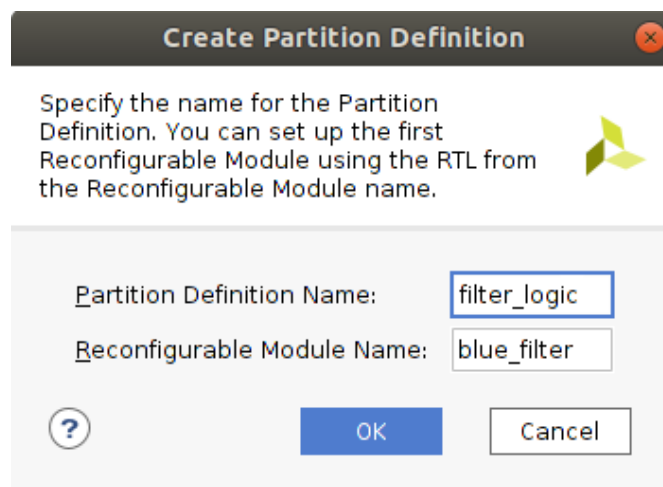


Figure 2.4: Creating the `filter_logic` partition definition

The Sources view has now changed slightly, with the `filter_logic` instance now shown with a yellow diamond, indicating it is a partition. You will 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 synthesising the `blue_filter` module.

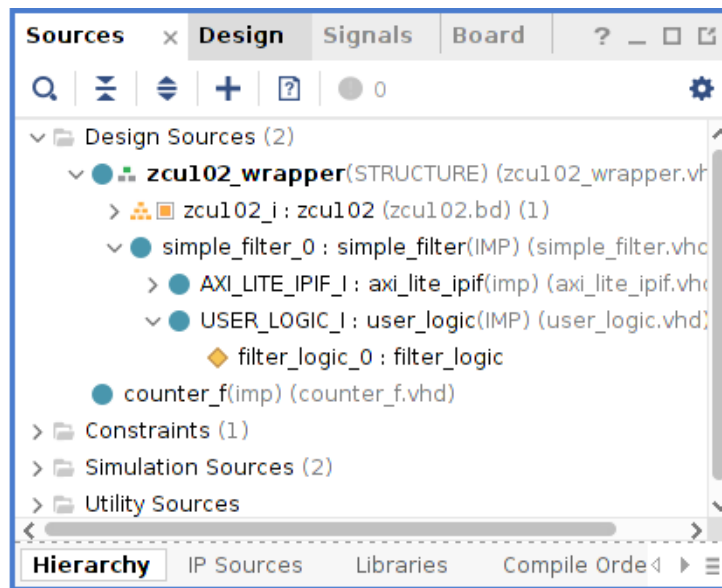


Figure 2.5: Sources view after defining filter_logic partition

At this point, more reconfigurable module sources may be added. This is done via the Dynamic Function eXchange Wizard.

Note: Once the partitions are defined, any additional RMs must be added via the DFX Wizard, and the management of RM sources, configurations, and runs must also be done via this wizard.

Step 3: Completing the Design with the DFX Wizard

1. Launch the DFX 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 `blue_filter` 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 `green_filter` folder:

- `<Extract_Dir>\Sources\hdl\green_filter`

Or use the **Add Files** button to select the `green_filter_logic.vhd` file residing in this directory. If module-level constraints were needed, they would be added here. Note that they would need to be scoped to the level of hierarchy for this partition.

Fill in the Reconfigurable Module Name to be `green_filter`. Set the Partition Definition to be `filter_logic`, leave Top Module Name empty and the Sources are already synthesized

check box unchecked. Click **OK** to create the new module.

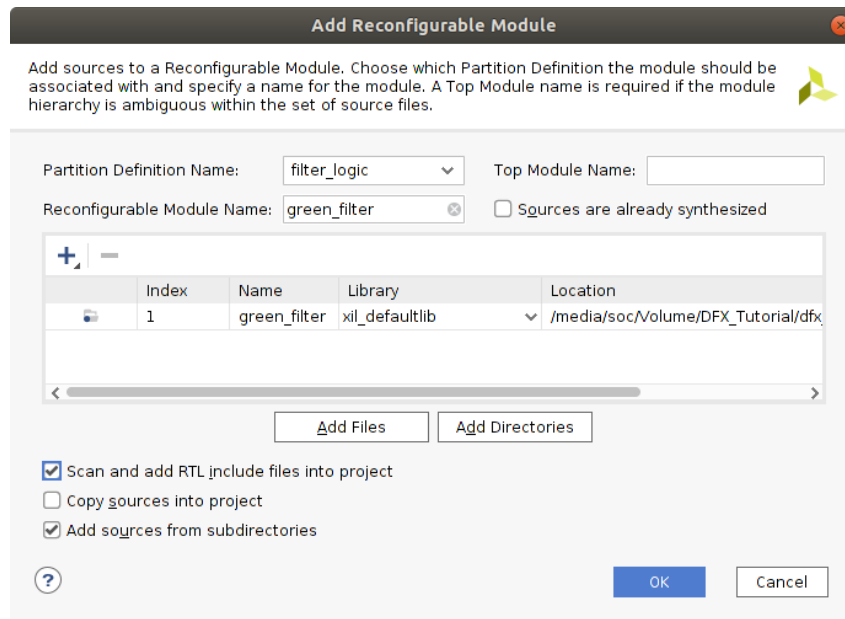


Figure 2.6: Add a new RM with the DFX Wizard

Two reconfigurable modules are now available for the `filter_logic` reconfigurable partition.

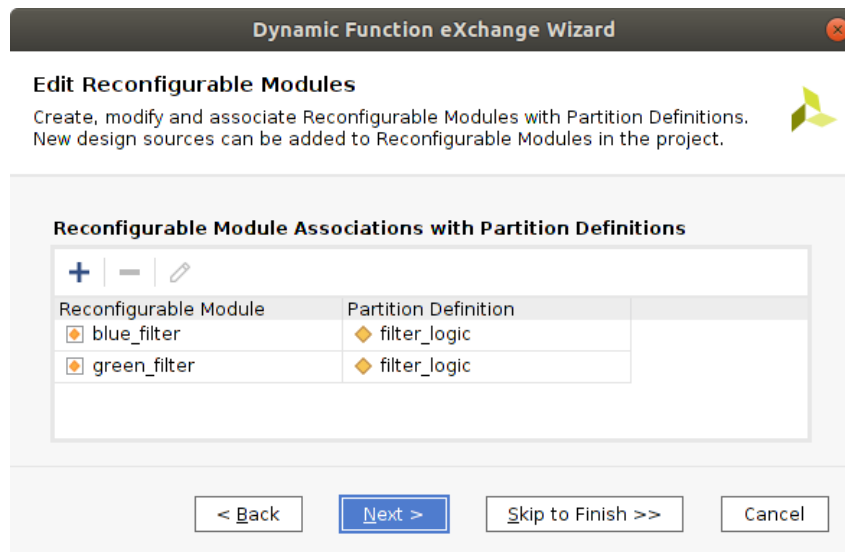


Figure 2.7: DFX Wizard with two RMs defined

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. Let the Wizard create the configurations by selecting the **automatically create configurations** link.

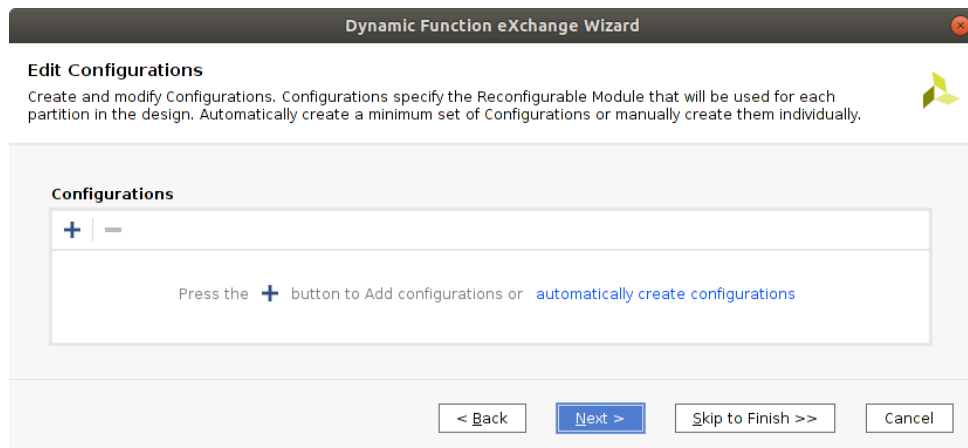


Figure 2.8: DFX Wizard Configurations page

After selecting this option, the minimum set of two configurations has been created. Each `logic_filter` instance has been given `blue_filter` in the first configuration and `green_filter` in the second configuration. Note that the Configuration Name is editable – in the example below, the names have been updated to `config_blue` and `config_green` to reflect the reconfigurable modules contained within each one.

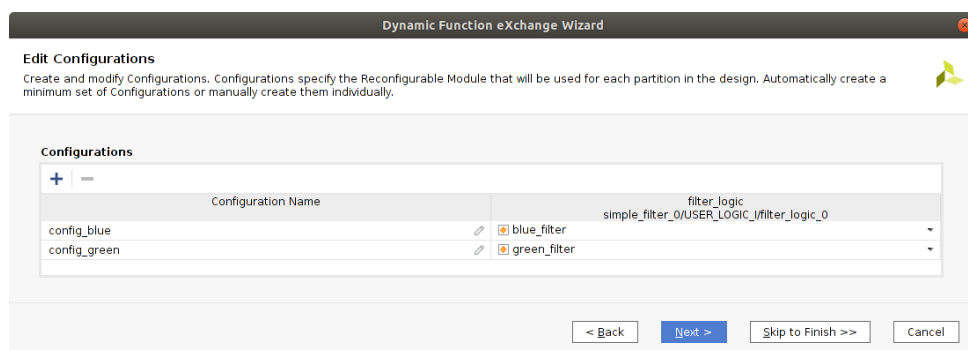


Figure 2.9: Auto-generated minimum set of configurations

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.

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. Click on the **automatically create configuration run** link to populate the Configuration Runs page with the minimum set of runs.

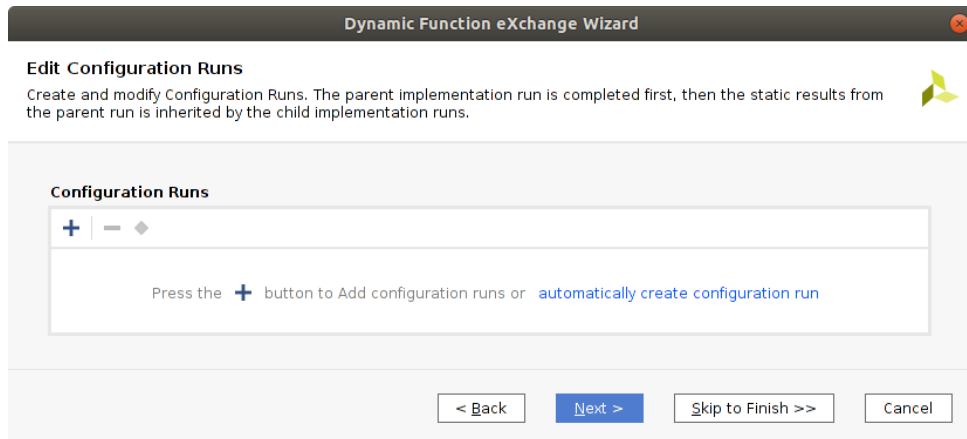


Figure 2.10: DFX Wizard Configuration Runs page

This creates two runs, consisting of one parent configuration (`config_blue`) and one child configuration (`config_green`). Any number of independent or related runs can be created within this wizard, with options for using different strategies or constraints 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.

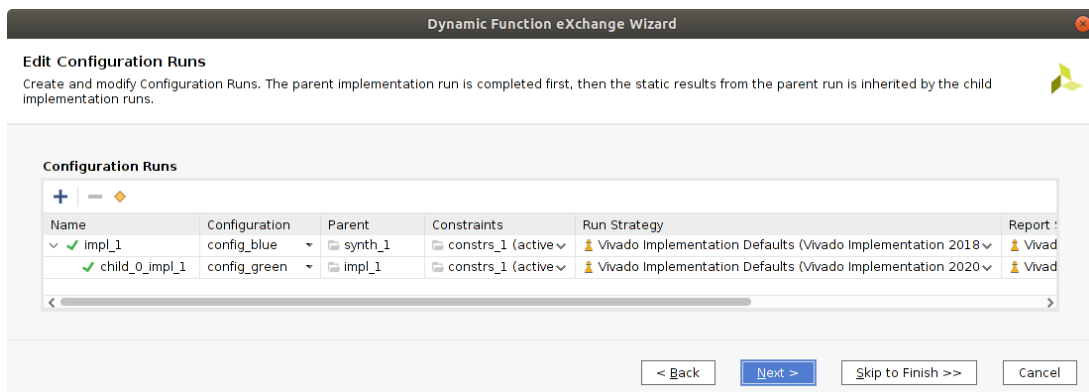


Figure 2.11: Auto-generated minimum set of configuration runs

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

Note: 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, you will see that the Design Runs window has been updated. A second Out-of-Context synthesis run has been added for the `green_filter` 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.

Tcl Console Messages Log Reports Design Runs x									
Name	Configuration	Constraints	Status	WNS	TNS	WHS	THS	TPWS	
synth_1 (active)		constrs_1	Not started						
impl_1 (active)	config_blue	constrs_1	Not started	2.158	0.000	0.005	0.000	0.000	
child_0_impl_1	config_green	constrs_1	Not started	2.158	0.000	0.005	0.000	0.000	
Out-of-Context Module Runs									
zcu102			Submodule Runs Complete						
blue_filter_synth_1		blue_filter	Not started						
green_filter_synth_1		green_filter	Not started						

Figure 2.12: Design Runs window showing all synthesis and implementation ready to launch

Step 4: Synthesising and Implementing the current design

With the design from above open in the 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 in 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**. This action will open a new window, where you can specify the launch options. For this tutorial, we will keep the remaining settings to default. The number of jobs allows you to specify how many CPU cores should be used for this launch. Then click **OK**.

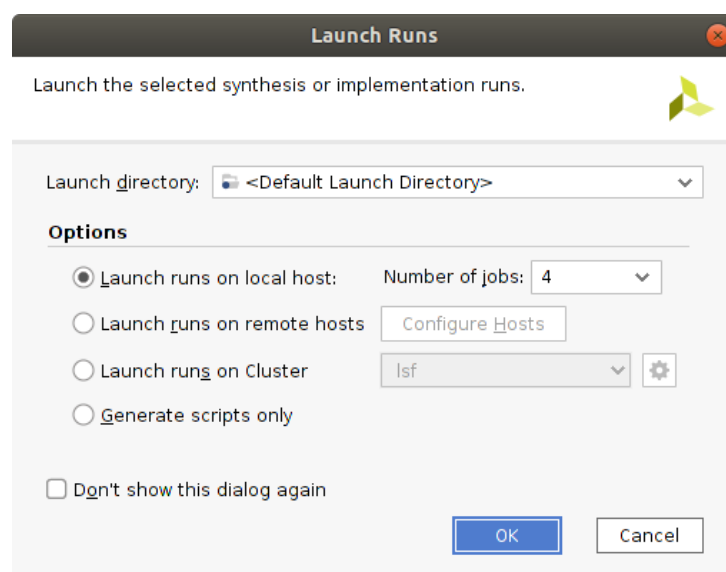


Figure 2.13: Launch options window

Now this will synthesise all OOC modules, followed by synthesis of the top-level design. This is no different from any design with OOC modules (IP or otherwise).

2. After the synthesis is successfully completed, select **Open Synthesized Design**.

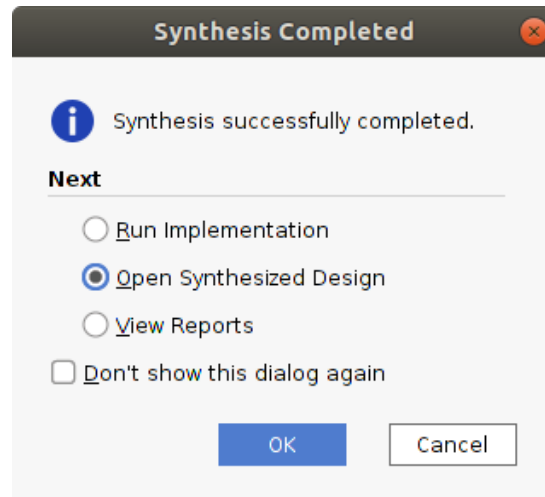


Figure 2.14: Synthesis successfully completed

Now the post-synthesis design will open. Since no Pblocks existed in the design sources, they can be created in this step. This can be done by right-clicking in the `filter_logic_0` instance in the design hierarchy to select **Floorplanning** → **Draw Pblock**.

Note: If you look at the statistic page of the cell properties of an instance, you can see what resources are needed by that instance. Since all partially configurable modules must fit into this Pblock, it must contain sufficient resources.

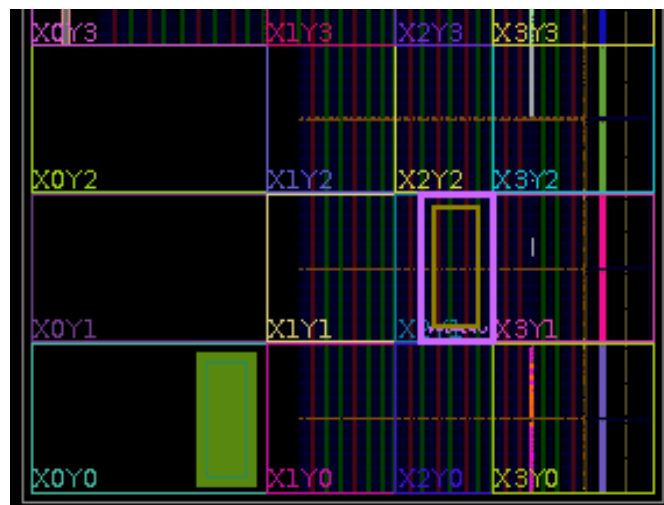


Figure 2.15: Floorplan with a reconfigurable partition (Pblock)

3. Select the Pblock in the floorplan and note its properties. The last property listed is SNAPPING_MODE, a property specific to DFX. Note that this option has been enabled in the Pblocks xdc.
4. 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.

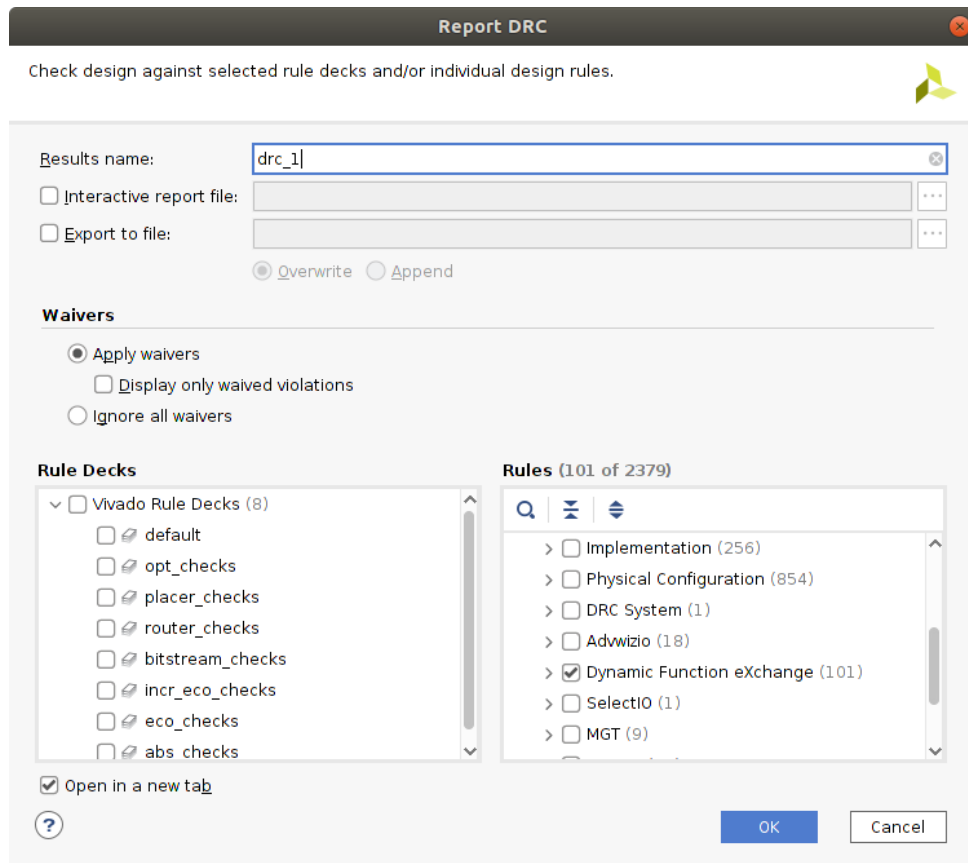


Figure 2.16: Checking DFX DRCs

If the DRC checks have been reported some issues, fix them before moving on. Remember that SNAPPING_MODE will resolve any errors related to horizontal or vertical alignment. For certain devices, hints can be given on how to improve the quality of the specified Pblocks. These can be ignored for this simple design.

TIP: Run DFX Design Rule Checks early and often.

The created Pblock can be saved to a separate xdc file, so it can be used in different designs. Saving the current constraints to the target project may cause your synthesis to go out-of-date. To avoid re-running synthesis, you can force the design up-to-date by selecting the run in the Design Tab, right-clicking, and selecting **Force Up-to-Date**.

5. In the Flow Navigator, select **Run Implementation** to run place and route on all configurations. Again, a window will open, where you can specify the launch options.

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 `filter_logic` 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, a module-level checkpoint is created for the routed `green_filter` RM. A locked static design image would be identical to the parent, so this step is not necessary.

If only specific configuration runs are desired, these can be individually selected within the Design Runs window. Note that 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.

6. When implementation completes, click **Cancel** in the resulting pop-up dialogue.

Name	Configuration	Constraints	Status	WNS	TNS	WHS	THS	TPWS
✓ synth_1 (active)		constrs_1	synth_design Complete!					
✓ impl_1 (active)	config_blue	constrs_1	route_design Complete!	2.158	0.000	0.005	0.000	0.000
✓ child_0_impl_1	config_red	constrs_1	route_design Complete!	2.158	0.000	0.005	0.000	0.000
Out-of-Context Module Runs								
> ✓ zcu102			Submodule Runs Complete					
✓ blue_filter_synth_1		blue_filter	synth_design Complete!					
✓ green_filter_synth_1		green_filter	synth_design Complete!					

Figure 2.17: All configurations routed

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.

7. In the Flow Navigator, click **Generate Bitstream**. This action launches bitstream generation on the active parent runs, 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 `red_filter_logic.vhd` file in `<Extract_Dir>\Sources\hdl\red_filter` then click **OK**.
4. Enter `red_filter` for the Reconfigurable Module Name and then click **OK** and **Next**.

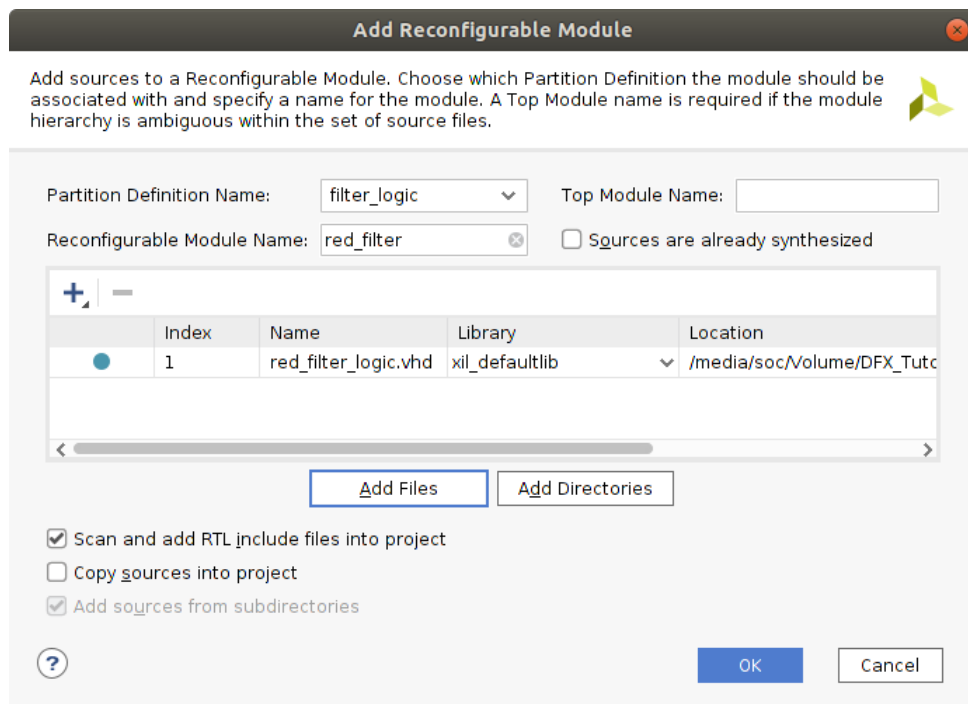


Figure 2.18: Adding a new reconfigurable module

Note that 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 will recreate all configurations and remove all existing results.

5. Create a new configuration by clicking the + button, entering the name `config_red`, then hitting **Enter**. Select `red_filter` for the reconfigurable partition instance.

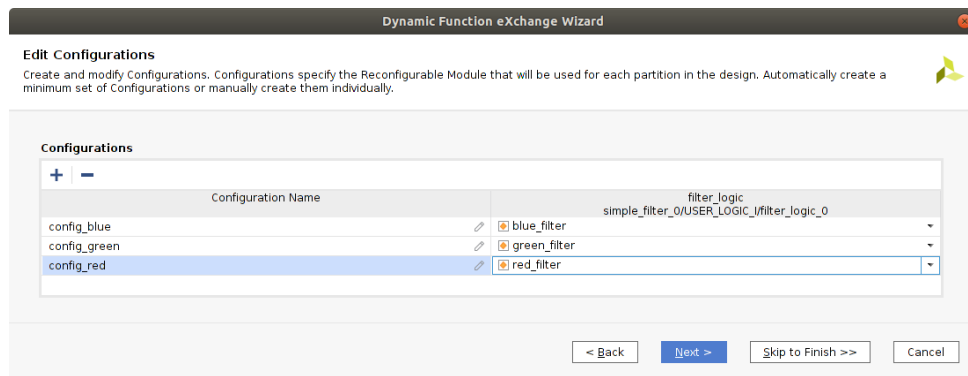


Figure 2.19: Creating new configuration

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` – this simply matches the existing convention
 - Parent: `impl_1` – this makes this configuration a child run of the existing parent run
 - Configuration: `config_red` – this is the one with the new RM that was just defined
7. Click **OK** to add the new Configuration Run.

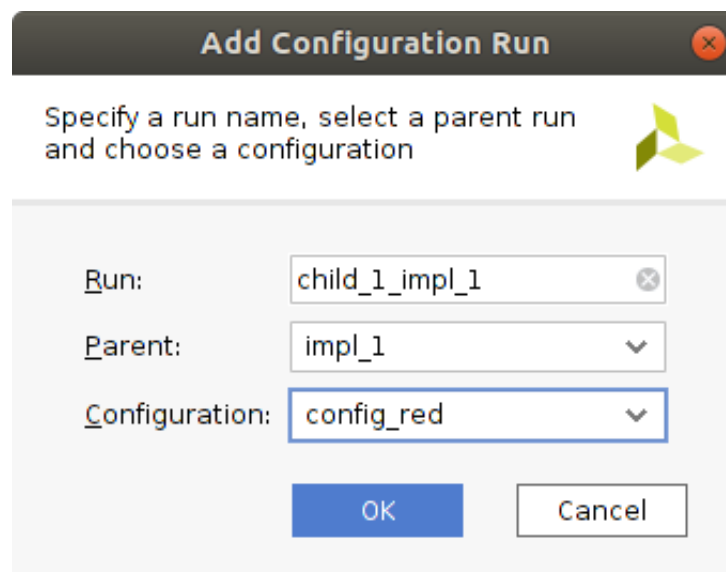


Figure 2.20: Creating a new Configuration Run

This new configuration, as a child of the existing `impl_1`, will reuse the static design implementation results, just like `config_green` 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.

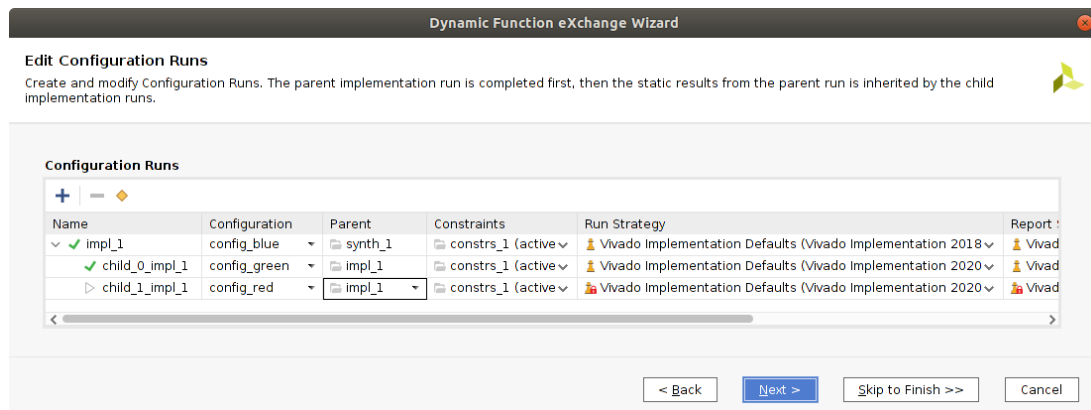


Figure 2.21: New configuration added as a new child run

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

Tcl Console Messages Log Reports Design Runs x									
<div> Q ≡ ⏮ ⏪ ▶ ⏩ + % </div>									
Name	Configuration	Constraints	Status	WNS	TNS	WHS	THS	TPWS	
✓ synth_1 (active)		constrs_1	synth_design Complete!						
✓ impl_1 (active)	config_blue	constrs_1	write_bitstream Complete!	2.158	0.000	0.005	0.000	0.000	
✓ child_0_impl_1	config_green	constrs_1	write_bitstream Complete!	2.158	0.000	0.005	0.000	0.000	
▶ child_1_impl_1	config_red	constrs_1	Not started						
Out-of-Context Module Runs									
▶ zcu102			Submodule Runs Complete						
✓ blue_filter_synth_1		blue_filter	synth_design Complete!						
✓ green_filter_synth_1		green_filter	synth_design Complete!						
▶ red_filter_synth_1		red_filter	Not started						

Figure 2.22: New OOC synthesis run and configuration run added

- Select this new child implementation run, right-click and select **Launch Runs**. This will run OOC synthesis on the `red_filter` module, then implement this module within the context of the locked static design.
- After the implementation run is successfully completed, generate the bitstreams. Therefore, select the new child implementation run, right-click and select **Generate Bitstream**. Leave all checkboxes unchecked and click **OK**.

Step 6: Partially reconfiguring the FPGA

Further steps are necessary to test the currently created bitstreams on the FPGA. The current design targets only the Xilinx ZCU102 Evaluation Kit.

The purpose of this project is to implement a partial reconfigurable Internet of Things (IoT) device, which can exchange parts of the synthesised hardware during runtime for a suitable application. Android is set up to run on the Xilinx Zynq software processing unit. An application is presented that applies filters to given images. These filters may be subject to updates in the future, and the app is able to download such updates and apply them to the Programmable Logic (PL) using dynamic partial reconfiguration.

If you are interested in this project or want to test the bitstreams, please follow the instructions in [4].

Conclusion

This concludes Lab 2. By now you should be able to:

- Created a new project based on a block design using a TCL script and prepared it for Dynamic Function eXchange
- Applied the changes in the top-level design to be able to integrate any partial partition (this has already been done)
- Created two configurations for a reconfigurable module
- Synthesised a design bottom-up to prepare for DFX implementation
- Created a valid floorplan for a DFX design
- Implemented these two configurations
- Created full and partial bitstreams

Lab 3

DFX Block Design with IP cores

This lab provides an introduction to how the DFX flow can be used in a project design based on a block design where the partially reconfigurable modules use IP cores. You will run an automated script that creates a new project and set up all the sources. Then you will modify the top-level design to create a DFX design. After that, you will finally set up all the runs that define the structure of a DFX design. The design used in this lab has one instance of a RM.

Laboratory Objectives

By the end of this lab, you should be able to:

- Create a new project based on a block design using a TCL script and prepare it for Dynamic Function eXchange
- Create new configurations for reconfigurable modules that use IP cores
- Synthesise a design bottom-up
- Create a valid floorplan
- Implement the configurations
- Create full and partial bitstreams

Step 1: Extract the Tutorial Design Files

1. Download or git clone the [reference design files](#) from the GitHub repository [3].
2. Store the contents to any write-accessible location.
3. In the stored files, navigate to `\dfx_ip_cores`.

Step 2: Load Initial Design Sources

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

1. Extract the design from the archive. The `dfx_ip_cores` data directory is referred to in this tutorial as the `<Extract_Dir>`.
2. Open the Vivado IDE and run the automated project creation script. This is done by calling `source ./scripts/create_hw_project.tcl` in the TCL Console.

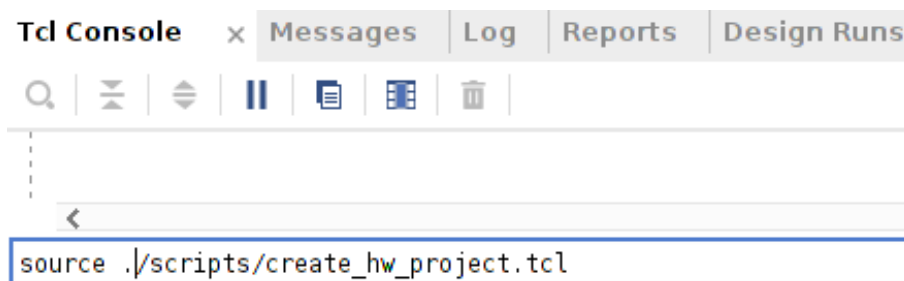


Figure 3.1: TCL Console project creation

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

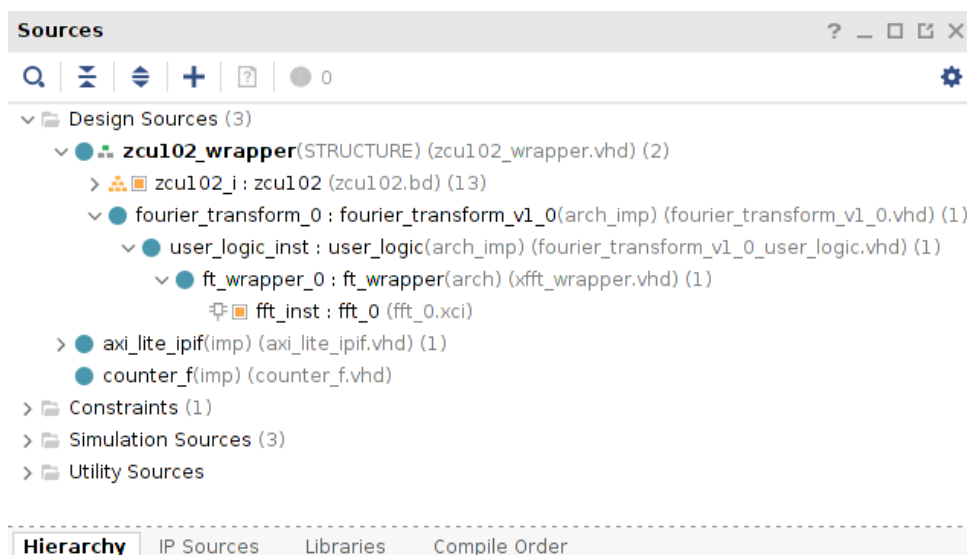


Figure 3.2: Sources view after project creation

3. Before we can continue, the created block design must be generated. In the Flow Navigator, click **Generate Block Design**. Now this will synthesise all block design OOC modules.

Note: Since Vivado does not offer DFX support for block designs in the latest version of the Vivado Design Suite, design adjustments must be made. The partially reconfigurable partition cannot be included in the block design, so the PR partition must be connected in the top-level design. Therefore, the required interfaces have to be exported to the top-level design. In the top-level design (here the `zcu102_wrapper.vhd` file) the partition has to be manually integrated. This step has already been taken in this lab.

4. Select **Tools** → **Enable Dynamic Function eXchange**. This action prepares the project for the DFX design flow.
-

Note: Once the Dynamic Function eXchange for a Vivado project is enabled, it cannot be undone, so Xilinx recommends archiving your project prior to selecting this option. The only way to return to a project that is not partially reconfigurable is to create a new one [6].

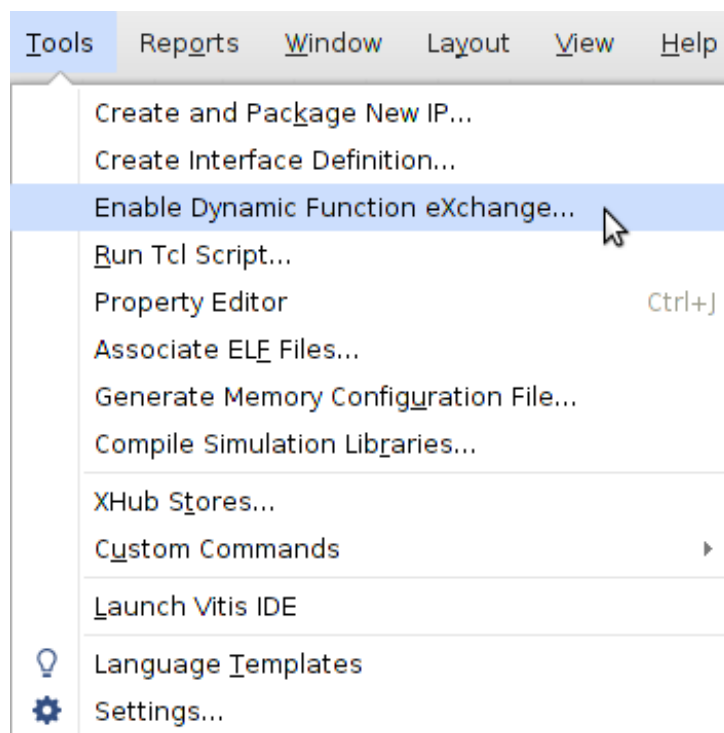


Figure 3.3: Enabling Dynamic Function eXchange

In the following pop-up window, click **Convert** to turn this project into a DFX project.

5. In the Sources view, right-click on the `ft_wrapper_0` instances and select the **Create Partition Definition...** option. This action will define the `ft_wrapper` instance as a reconfigurable

partition in the design.

Bottom-up synthesis is required to keep this module separated from top. Bottom-up synthesis refers to a synthesis flow in which each module has its own synthesis project. This generally involves turning off automatic I/O buffer insertion for the lower level modules to ensure no optimisation occurs across the module boundaries. To synthesise the top-level, a netlist with a black box for each reconfigurable partition must be available. This requires the top-level synthesis to have module or entity declarations for the partitioned instances, but no logic. The top-level synthesis also infers or instantiates I/O buffers on all top-level ports.

6. In the dialogue box that appears, enter names for 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 a suitable name, such as `ft_wrapper`. The reconfigurable module refers to this specific RTL instance, so give it a name that references its functionality, such as `xfft`, then click **OK**.

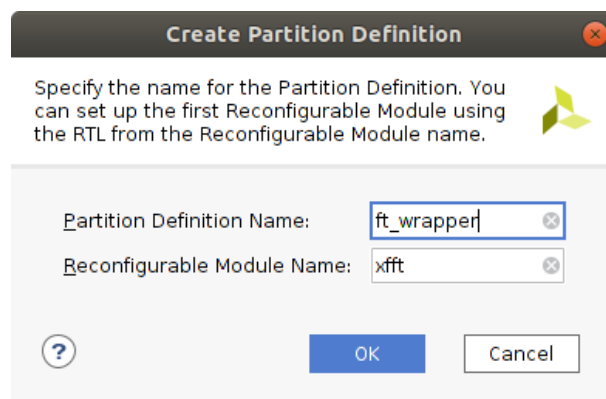


Figure 3.4: Creating the `ft_wrapper` partition definition

The Sources view has now changed slightly, with the `ft_wrapper` instance now shown with a yellow diamond, indicating it is a partition. You will 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 synthesising the `xfft` module.

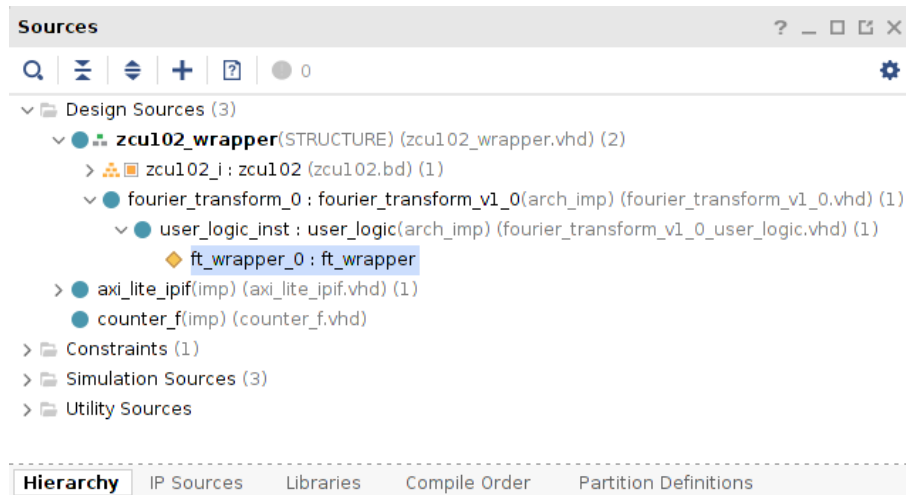


Figure 3.5: Sources view after defining ft_wrapper partition

At this point, more reconfigurable module sources may be added. This is done via the Dynamic Function eXchange Wizard.

Note: Once the partitions are defined, any additional RMs must be added via the DFX Wizard, and the management of RM sources, configurations, and runs must also be done via this wizard.

Step 3: Completing the Design with the DFX Wizard

1. Launch the DFX 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 `xffft` 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 `xdfft` folder:

- `<Extract_Dir>\Sources\hdl\xdfft`

This action will include the `xdfft_wrapper.vhd` file and all underlying IP cores for this RM. If module-level constraints were needed, they would be added here. Note that they would need to be scoped to the level of hierarchy for this partition.

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

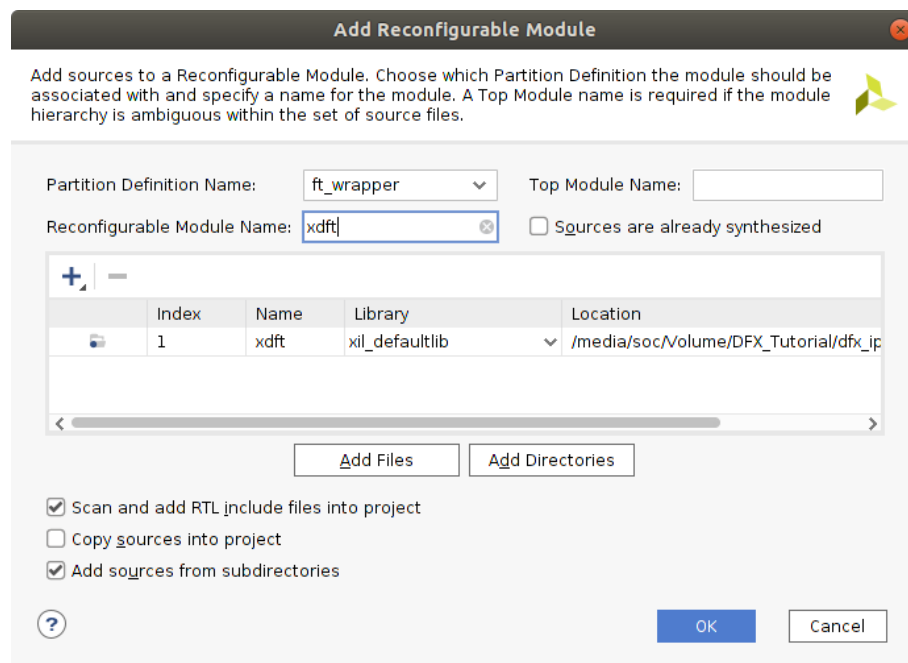


Figure 3.6: Add a new RM with the DFX Wizard

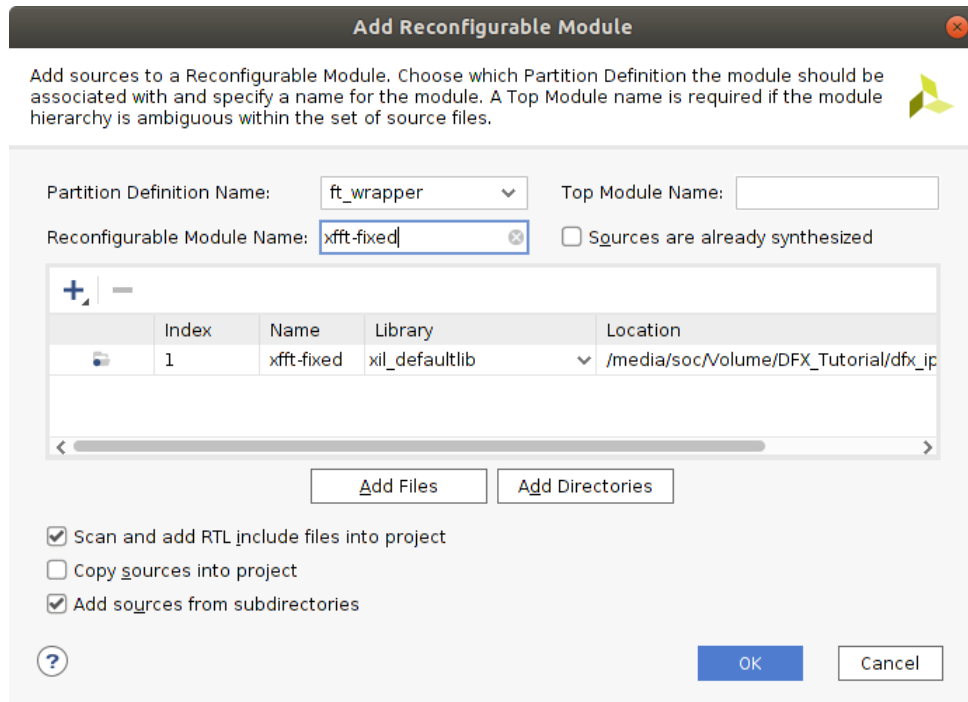
4. This time we will include a third reconfigurable module for this partition. Click on the blue + icon again to add a new RM.
5. Click the **Add Directories** button to select the `xfft-fixed` folder:

- `<Extract_Dir>\Sources\hdl\xfft-fixed`

This action will include the `xfft-fixed_wrapper.vhd` file and all underlying IP cores for this RM. If module-level constraints were needed, they would be added here. Note that they would need to be scoped to the level of hierarchy for this partition.

Note: Since the different RMs partly use the same IP cores, an adjustment has to be made here. Vivado does not allow you to use the same IP core in different RMs. The IP cores must have a unique name. If a IP core is copied and the name is changed, everything is fine. So if you look at the IP cores in the Sources folder, you will notice that each IP core already has a unique name. In this case, a prefix specifying the reconfigurable module is included.

Fill in the Reconfigurable Module Name to be `xfft-fixed`. Set the Partition Definition to be `ft_wrapper`, leave Top Module Name empty and the Sources are already synthesized check box unchecked. Click **OK** to create the new module.



Add Reconfigurable Module

Add sources to a Reconfigurable Module. Choose which Partition Definition the module should be associated with and specify a name for the module. A Top Module name is required if the module hierarchy is ambiguous within the set of source files.

Partition Definition Name: `ft_wrapper` Top Module Name:

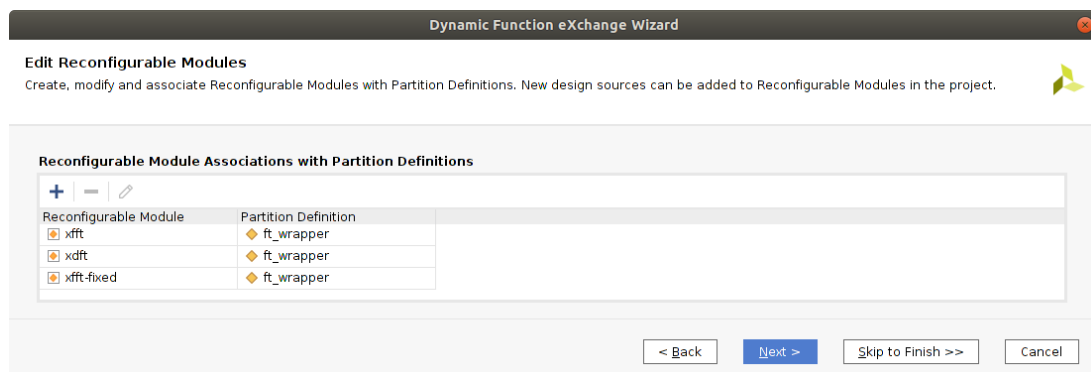
Reconfigurable Module Name: `xfft-fixed` ☐ Sources are already synthesized

	Index	Name	Library	Location
	1	xfft-fixed	xil_defaultlib	/media/soc/Volume/DFX_Tutorial/dfx_ip

☒ Scan and add RTL include files into project
☐ Copy sources into project
☒ Add sources from subdirectories

Figure 3.7: Add a new RM with the DFX Wizard

Three reconfigurable modules are now available for the `ft_wrapper` reconfigurable partition.



Dynamic Function eXchange Wizard

Edit Reconfigurable Modules
 Create, modify and associate Reconfigurable Modules with Partition Definitions. New design sources can be added to Reconfigurable Modules in the project.

Reconfigurable Module Associations with Partition Definitions

Reconfigurable Module	Partition Definition
xfft	ft_wrapper
xdft	ft_wrapper
xfft-fixed	ft_wrapper

Figure 3.8: DFX Wizard with three RMs defined

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.

- Let the Wizard create the configurations by selecting the **automatically create configurations** link.

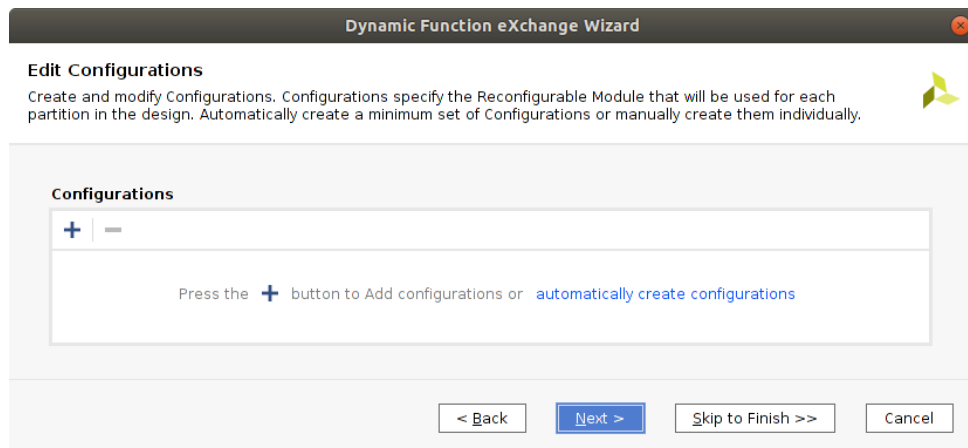


Figure 3.9: DFX Wizard Configurations page

After selecting this option, the minimum set of three configurations has been created. The `ft_wrapper` instance has been given `xfft` in the first configuration, `xdft` in the second configuration, and `xfft-fixed` in the third configuration. Note that the Configuration Name is editable – in the example below, the names have been updated to `config_xfft`, `config_xdft`, and `config_xfft-fixed` to reflect the reconfigurable modules contained within each one.

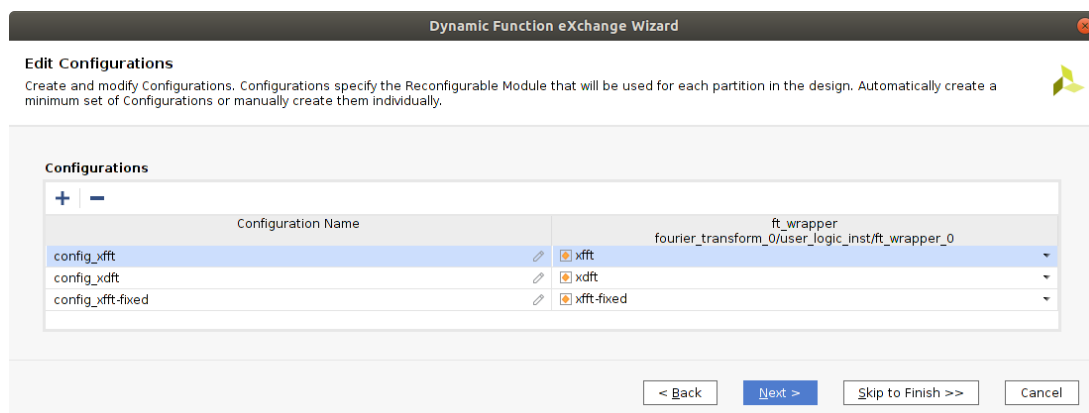


Figure 3.10: Auto-generated minimum set of configurations

7. 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.
8. Click on the **automatically create configuration run** link to populate the Configuration Runs page with the minimum set of runs.

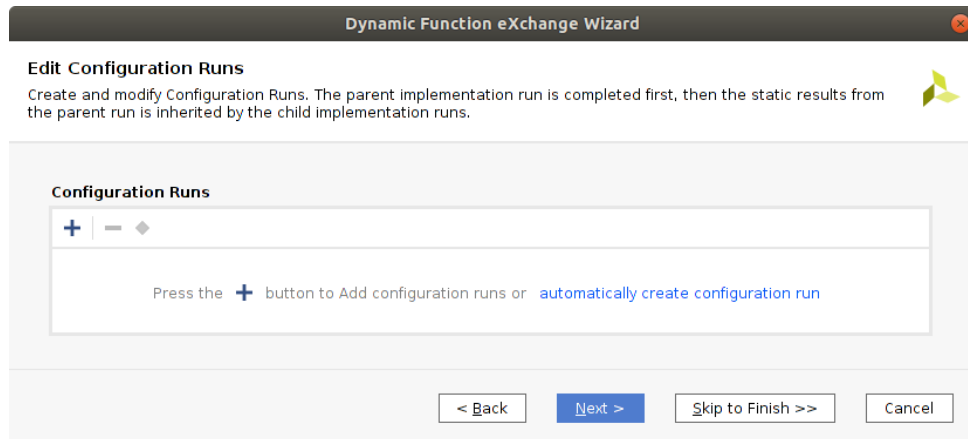


Figure 3.11: DFX Wizard Configuration Runs page

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

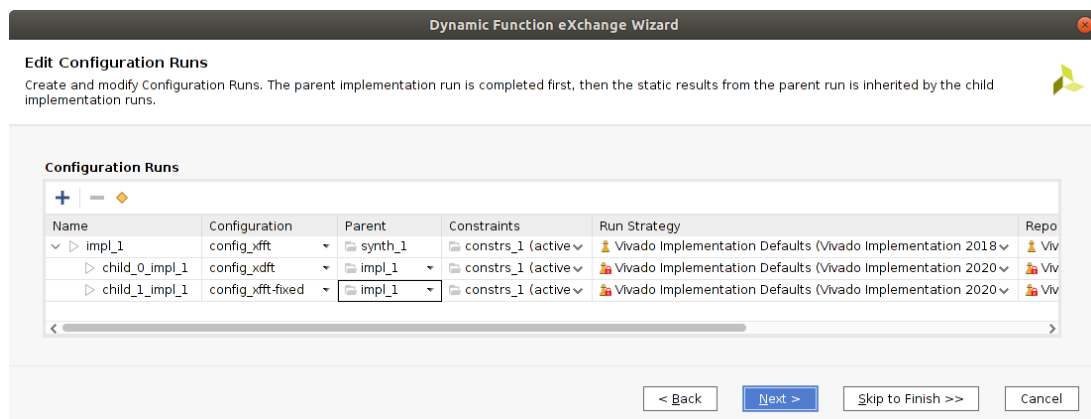


Figure 3.12: Auto-generated minimum set of configuration runs

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

Note: 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, you will see that the Design Runs window has been updated. There

are now three Out-of-Context synthesis runs available, one for each RM. Two child implementation runs (`child_0_impl_1` and `child_1_impl_1`) were also created under the parent (`impl_1`). You are now ready to process the design.

Tcl Console

Messages

Log

Reports

Configurations

Design Runs

🔍

⏏

⏴

⏵

⏮

⏭

⏪

⏩

+

%

Name	Configuration	Constraints	Status
<div> <div>▼</div> <div>▷</div> <div>synth_1 (active)</div> </div>		constrs_1	Not started
<div> <div>▼</div> <div>▷</div> <div>impl_1 (active)</div> </div>	config_xfft	constrs_1	Not started
<div> <div>▷</div> <div>child_0_impl_1</div> </div>	config_xdft	constrs_1	Not started
<div> <div>▷</div> <div>child_1_impl_1</div> </div>	config_xfft-fixed	constrs_1	Not started
<div> <div>▼</div> <div>📁</div> <div>Out-of-Context Module Runs</div> </div>			
<div> <div>></div> <div>✓</div> <div>zcu102</div> </div>			Submodule Runs Complete
<div> <div>▷</div> <div>xfft_synth_1</div> </div>		xfft	Not started
<div> <div>▷</div> <div>xdft_synth_1</div> </div>		xdft	Not started
<div> <div>▷</div> <div>xfft-fixed_synth_1</div> </div>		xfft-fixed	Not started

Figure 3.13: Design runs window showing all synthesis and implementation ready to launch

Step 4: Synthesising and Implementing the current design

With the design from above open in the 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 in 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**. This action will open a new window, where you can specify the launch options. For this tutorial, we will keep the remaining settings to default. The number of jobs allows you to specify how many CPU cores should be used for this launch. Then click **OK**.

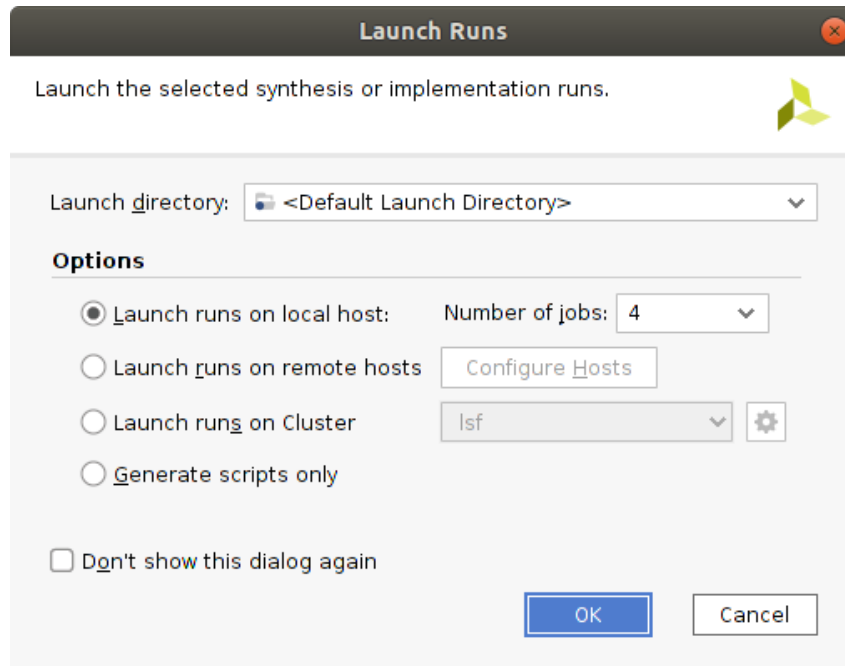


Figure 3.14: Launch options window

Now this will synthesise all OOC modules, followed by synthesis of the top-level design. This is no different from any design with OOC modules (IP or otherwise).

2. After the synthesis successfully completed, select **Open Synthesized Design**.

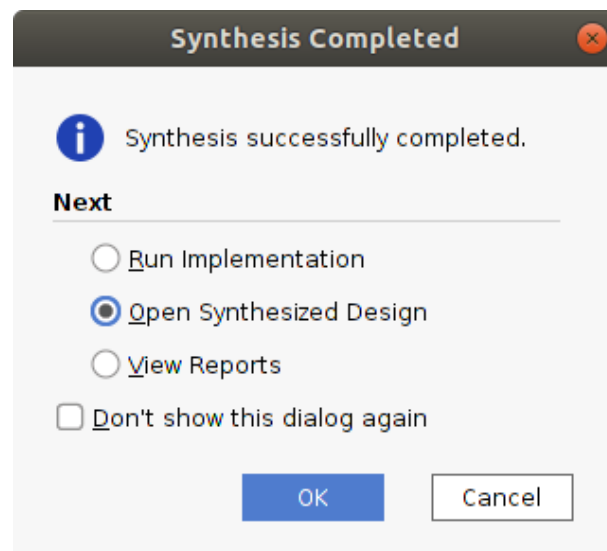


Figure 3.15: Synthesis successfully completed

Now the post-synthesis design will open. Since no Pblocks existed in the design sources, they can be created in this step. This can be done by right-clicking in the `ft_wrapper_0` instance in the design hierarchy to select **Floorplanning** → **Draw Pblock**.

Note: If you look at the statistic page of the cell properties of an instance, you can see what resources are needed by that instance. Since all partially configurable modules must fit into this Pblock, it must contain sufficient resources.

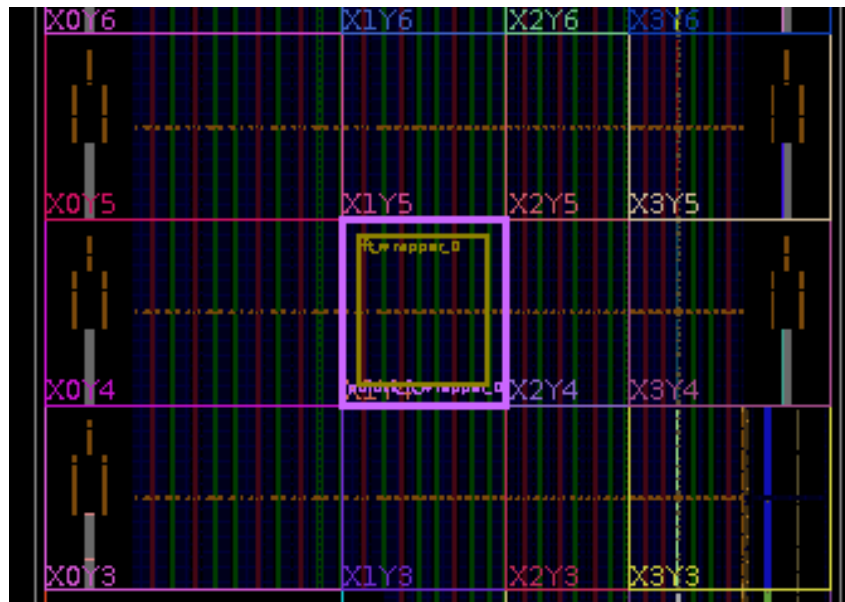


Figure 3.16: Floorplan with a reconfigurable partition (Pblock)

3. Select the Pblock in the floorplan and note its properties. The last property listed is SNAPPING_MODE, a property specific to DFX. Note that this option has been enabled in the Pblocks xdc.
4. 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.

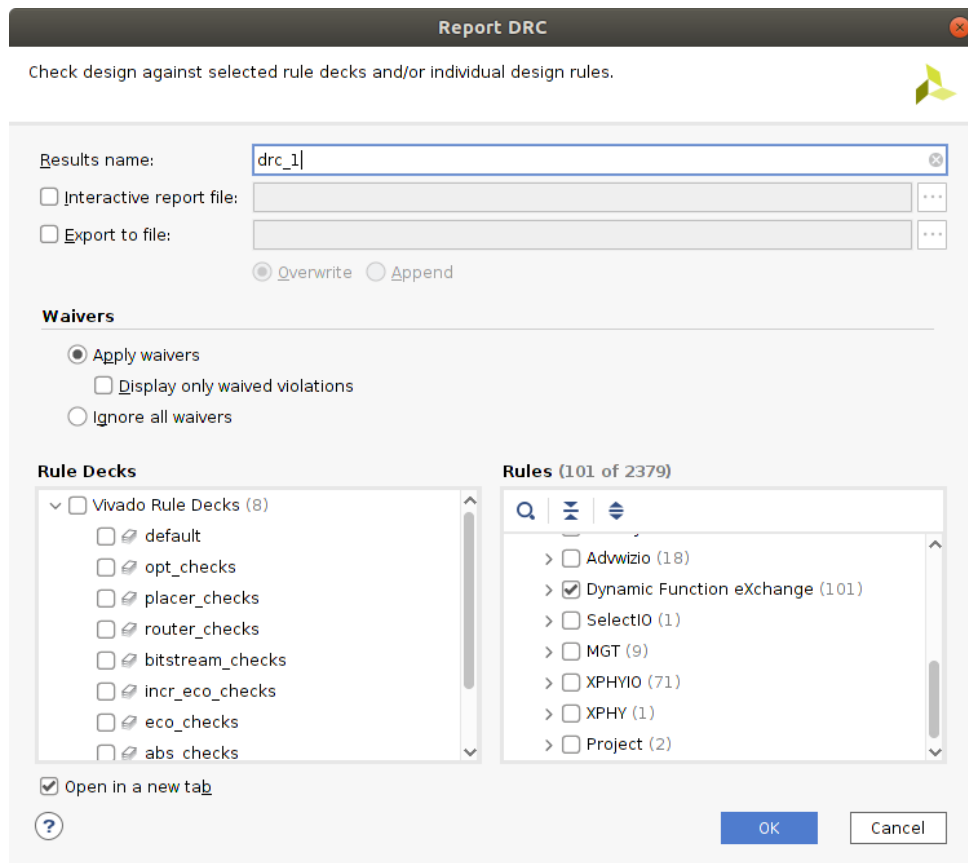


Figure 3.17: Checking DFX DRCs

If the DRC checks have been reported some issues, fix them before moving on. Remember that SNAPPING_MODE will resolve any errors related to horizontal or vertical alignment. For certain devices, hints can be given on how to improve the quality of the specified Pblocks. These can be ignored for this simple design.

TIP: Run DFX Design Rule Checks early and often.

The created Pblock can be saved to a separate xdc file, so it can be used in different designs. Saving the current constraints to the target project may cause your synthesis to go out-of-date. To avoid re-running synthesis, you can force the design up-to-date by selecting the run in the Design Tab, right-clicking, and selecting **Force Up-to-Date**.

5. In the Flow Navigator, select **Run Implementation** to run place and route on all configurations. Again, a window will open, where you can specify the launch options.

This action runs implementation first for impl_1 and then for child_0_impl_1 and child_1_impl_1. Behind the scenes, Vivado takes care of all the details. In addition to running place and route for the three 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 `ft_wrapper` 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, a module-level checkpoint is created for the routed RM. A locked static design image would be identical to the parent, so this step is not necessary. If only specific configuration runs are desired, these can be individually selected within the Design Runs window. Note that 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.

6. When implementation completes, click **Cancel** in the resulting pop-up dialogue.

Name	Configuration	Constraints	Status	WNS	TNS	WHS	THS	TPWS
✓ synth_1 (active)		constrs_1	synth_design Complete!					
✓ impl_1 (active)	config_xfft	constrs_1	route_design Complete!	2.113	0.000	0.010	0.000	0.000
✓ child_0_impl_1	config_xdft	constrs_1	route_design Complete!	2.113	0.000	0.007	0.000	0.000
✓ child_1_impl_1	config_xfft-fixed	constrs_1	route_design Complete!	2.113	0.000	0.010	0.000	0.000
Out-of-Context Module Runs								
> ✓ zcu102			Submodule Runs Complete					
✓ xfft_synth_1		xfft	synth_design Complete!					
✓ xdft_synth_1		xdft	synth_design Complete!					
✓ xfft-fixed_synth_1		xfft-fixed	synth_design Complete!					

Figure 3.18: All configurations routed

At this point, there are two steps remaining. The first is running PR Verify to compare the three 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.

7. In the Flow Navigator, click **Generate Bitstream**. This action launches bitstream generation on the active parent runs, 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: Partially reconfiguring the FPGA

Further steps are necessary to test the currently created bitstreams on the FPGA. The current design targets only the Xilinx ZCU102 Evaluation Kit.

The purpose of this project is to implement a partial reconfigurable IoT device, which can exchange parts of the synthesised hardware during runtime for a suitable application. Android is set up to run on the Xilinx Zynq software processing unit. An application is presented that applies filters to given images. These filters may be subject to updates in the future, and the app is able to download such updates and apply them to the PL using dynamic partial reconfiguration.

If you are interested in this project or want to test the bitstreams, please follow the instructions in [5].

Conclusion

This concludes Lab 3. By now you should be able to:

- Created a new project based on a block design using a TCL script and prepared it for Dynamic Function eXchange
- Applied the changes in the top-level design to be able to integrate any partial partition and ensure all IP cores have a unique name (this has already been done)
- Created three configurations for a reconfigurable module that use IP cores
- Synthesised a design bottom-up to prepare for DFX implementation
- Created a valid floorplan for a DFX design
- Implemented these three configurations
- Created full and partial bitstreams

Bibliography

- [1] Xilinx, “Vivado design suite tutorial - dynamic function exchange,” *UG947 (v2020.2)*, Feb. 2021. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2020_2/ug947-vivado-partial-reconfiguration-tutorial.pdf
- [2] —. (2021) Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. [Xilinx Website]. Last Accessed: February 27, 2022. [Online]. Available: <https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html>
- [3] A. Dejmek. (2022, Feb. 27) Github dynamic function exchange tutorial repository. [GitHub]. Last Accessed: February 27, 2022. [Online]. Available: https://github.com/1chor/DFX_Tutorial
- [4] —. (2020, Dec. 19) Github SoC project repository. [GitHub]. Last Accessed: February 27, 2022. [Online]. Available: <https://github.com/1chor/SoC-project>
- [5] —. (2022, Jan. 30) Github master thesis repository. [GitHub]. Last Accessed: February 27, 2022. [Online]. Available: <https://github.com/1chor/master-thesis>
- [6] Xilinx, “Vivado design suite user guide - dynamic function exchange,” *UG909 (v2020.2)*, Feb. 2021. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug909-vivado-partial-reconfiguration.pdf