Dynamic Function eXchange Tool Flow

2022.2

Abstract

This lab covers the basic Dynamic Function eXchange (DFX) tool flow in the Vivado™ Design Suite.

This lab should take approximately 45 minutes.

CloudShare Users Only

You are provided with three attempts to access a lab, and the time allotted to complete each lab is twice the time expected to complete the lab. Once the timer starts, you cannot pause the timer. Each lab attempt will reset the previous attempt—that is, your work from a previous attempt is not saved.

Objectives

After completing this lab, you will be able to:

- Describe the basic steps of the Dynamic Function eXchange (DFX) tool flow
- Create a DFX project, create different configurations using multiple RMs, and generate full and partial bitstreams in DFX non-project mode
- Effectively prepare scripts for multiple configurations
- Program the FPGA with full and partial bitstreams to evaluate the design

Introduction

The sample design used throughout this lab is called *led shift_count*. This design is very small, which helps to minimize data size and allows you to run the lab quickly with minimal hardware requirements.

The design has two reconfigurable partitions (RPs) (**shift** and **count**). Each reconfigurable partition (RP) has two reconfigurable modules (RM), resulting in four total configurations (**shift_right**, **shift_left**, **count_up**, and **count_down**) for this design.

However, two of the four would be sufficient to consider all cases—one configuration with **count_up** and **shift_left** RM variations and the other configuration with the **count_down** and **shift_right** RM variations.

The DFX flow is now supported in both the Vivado Design Suite project-mode and non-project batch mode. This lab focuses on the DFX non-project tool flow.

Note that all the figures in this lab are in respect to the ZCU104 board.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run on a Linux platform.

One environment variable is required: `TRAINING_PATH`, which points to where the lab files are located. This variable comes configured in the CloudShare/CustEd_VM environments.

Some tools can use this environment variable directly (that is, `$TRAINING_PATH` is expanded), and some tools require manual expansion (`/home/amd/training` for the CloudShare/CustEd_VM environments). The lab instructions describe what to do for each tool. Other environments require the definition of this variable for the scripts to work properly.

Both the Vivado Design Suite and the Vitis platform offer a Tcl environment that is used in many labs. When the tool is launched, it starts with a clean Tcl environment with none of the procs or variables remaining from any previous launch of the tools.

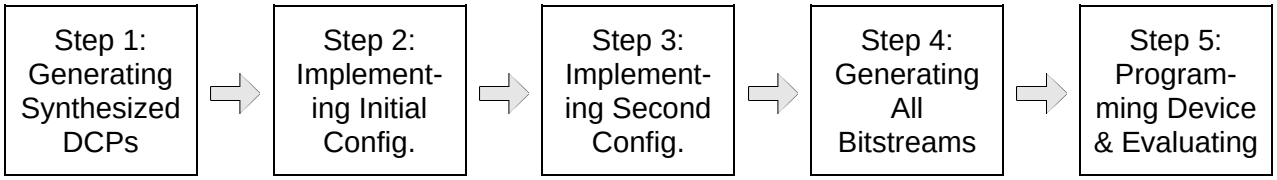
If you sourced a Tcl script or manually set any Tcl variables and you closed the tool, when you reopen the tool, you will need to re-source the Tcl script and set any variables that the lab requires. This is also true of terminal windows—any variable settings will be cleared when a new terminal opens.

Nomenclature

Formal nomenclature is used to explain how different arguments are used. The following are some of the more commonly used symbols:

Symbol	Description	Example	Explanation
<`text`**>**	Indicates a field	`cd <dir>`	`<dir>` represents the name of the directory. The < and > symbols are NOT entered. If the directory to change to is `XYZ`, then you would enter **`cd XYZ`** into the environment.
[`text`**]**	Indicates an optional argument	`ls [ | more]`	This could be interpreted as `ls <Enter>` or `ls | more <Enter>`. The first instance lists the files in the current Linux directory, and the second lists the files in the current Linux directory, but additionally runs the output through the `more` tool, which paginates the output. Here, the pipe symbol (`|`) is a Linux operator.
|	Indicates choices	`cmd <ZCU104 | VCK190>`	The `cmd` command takes a single argument, which could be `ZCU104` OR `VCK190`. You would enter either `cmd ZCU104` or `cmd VCK190`.

General Flow



Generating Synthesized Design Checkpoints Step 1

The first step in this lab is to run synthesis and store the synthesized design checkpoints by using the provided script files.

Because there are two reconfigurable partitions (RPs), each with two possible configurations (excluding the black box), four netlists must be generated. The netlist for each RP must be named identically so that it can connect to the static logic; hence, the separate directories.

1-1. Launch the Vivado Design Suite Tcl shell.

1-1-1. Open a Linux terminal window (press <**Ctrl** + **Alt** + **T**>) and enter the following commands:

```
[host]$ source /opt/amd/Vivado/2022.2/settings64.sh
[host]$ vivado -mode tcl
```

Note: The installation path (`/opt/amd`) is valid for the CustEd VM and CloudShare environments. Modify the path as necessary for your environment.

This opens the Tcl interactive shell in the terminal itself.

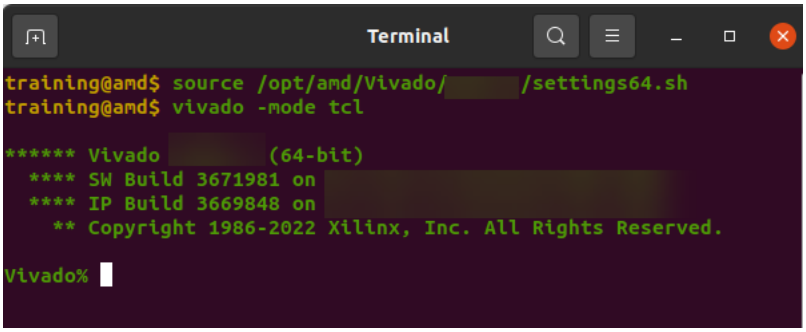


Figure 1-1: Vivado Tcl Interactive Shell

1-1-2. Enter the following command to change to the lab working directory:

```
cd $::env(TRAINING_PATH)/dfx_flow/lab/ZCU104
```

1-1-3. Enter **`pwd`** to verify the present working directory.

1-2. Review the provided source files.

1-2-1. Browse to the `$TRAINING_PATH/dfx_flow/lab/ZCU104/Sources` directory.

This folder has two sub-folders: `hdl` and `xdc`. As implied by the names, the `hdl` folder contains HDL source files used by this lab and the `xdc` folder contains constraints files for the design.

1-2-2. Open the **`hdl`** folder.

This directory has five sub-folders. One folder is for the top module of the design. The top module has two reconfigurable modules (shift and count) in it, and each has two configurations. Hence, there are four folders of reconfigurable modules.

1-2-3. Open the **`top`** folder.

1-2-4. Open and review **`top.v`** using your preferred text editor.

Note that there are black-box definitions for two reconfigurable modules (RMs) at lines 121 and 129. When the top module is synthesized, the two RMs will be implemented as black boxes (without optimizing the boundaries for black-box defined modules) and the Vivado Design Suite expects the netlists for these RMs during the `link_design` stage of the implementation.

1-2-5. Open and review **`shift_left.v`** in the `Sources/hdl/shift_left` folder.

Note that the module name is shift and the module is implemented via the RAMB36E2 primitive.

1-2-6. Open and review **`shift_right.v`** in the `Sources/hdl/shift_right` folder.

Question 1

What are the module names in *shift_left.v* and *shift_right.v*?

1-2-7. Open and review the **count_up.v** and **count_down.v** files in the `Sources/hdl/count_up` and `Sources/hdl/count_down` folders, respectively.

1-2-8. Open and review the **top_io.xdc** file in the `Sources/xdc` folder.

Note that the *top_io.xdc* file contains timing and physical constraints for this design.

1-2-9. Close all opened files in the text editor.

Do not save anything if prompted.

1-3. Generate the static checkpoint.

You will run the synthesis for the static design by using the `synth_design` command with the `-flatten_hierarchy rebuilt` option. This option allows the synthesis tool to flatten the hierarchy, perform synthesis, and then rebuild the hierarchy based on the original RTL.

1-3-1. Browse to the `$TRAINING_PATH/dfx_flow/lab/ZCU104` directory.

1-3-2. Open **`run_synthesis.tcl`** using your preferred text editor.

Note that this sources multiple Tcl files for generating the synthesized design checkpoints for static and RM logic.

1-3-3. Open and review the **`run_synth_top_static.tcl`** and **`run_synth_count_up_rm.tcl`** files using your preferred text editor.

Note that `-mode out_of_context` is used when RM logic is synthesized. This option ensures that synthesis does not insert I/O buffers into that module.

Question 2

Where will the synthesized design checkpoints be stored?

1-3-4. Enter the following command in the Vivado Tcl shell to run synthesis and generate the synthesized design checkpoints for this design:

```
vivado -mode batch -source run_synthesis.tcl
```

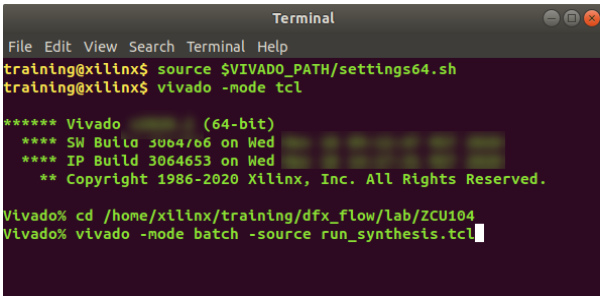


Figure 1-2: Sourcing run_synthesis.tcl

This process will take approximately 2–3 minutes.

Note that the LOG and JOU files will be saved in the launch directory.

The synthesized design checkpoints will be stored in their respective directories in the `SynOutputDir` directory.

1-3-5. Close all opened files in the text editor.

Do not save anything if prompted.

Implementing the Initial Configuration Step 2

Here you will complete the script for the initial configuration and run the script to generate the routed design checkpoint for initial configuration. You will also lock down the static routed design checkpoint to be used with the rest of the configuration implementations.

The `run_impl_config1.tcl` script is for the initial configuration implementation (with count_up and shift_left reconfigurable module variations). Running the initial configuration with timing-critical RM variations is recommended.

2-1. Review `run_impl_config1.tcl` and complete the script for initial configuration.

2-1-1. Browse to the `$TRAINING_PATH/dfx_flow/lab/ZCU104` directory.

2-1-2. Open **`run_impl_config1.tcl`** using your preferred text editor.

2-1-3. Uncomment line 13 to open the static synthesized design checkpoint:

```
open_checkpoint $SynOutputDir/post_synth_top/post_synth_top.dcp
```

2-1-4. Uncomment line 16 to read the count_up synthesized design checkpoint for the inst_count cell:

```
read_checkpoint -cell inst_count $SynOutputDir/
post_synth_count_up/post_synth_count_up.dcp
```

2-1-5. Uncomment line 17 to read the shift_left synthesized design checkpoint for the inst_shift cell:

```
read_checkpoint -cell inst_shift $SynOutputDir/
post_synth_shift_left/post_synth_shift_left.dcp
```

Question 3

What is the difference between the `open_checkpoint` and `read_checkpoint` commands?

2-1-6. Uncomment lines 20 and 21 to turn on the HD.RECONFIGURABLE property on the inst_count and inst_shift cells:

```
set_property HD.RECONFIGURABLE true [get_cells inst_count]
set_property HD.RECONFIGURABLE true [get_cells inst_shift]
```

These commands specify that the inst_count and inst_shift cells are going to be reconfigurable.

2-1-7. Uncomment line 24 to read the `pblocks.xdc` file:

```
read_xdc $srcDir/xdc/pblocks.xdc
```

2-1-8. Open and review the **pblocks.xdc** file in the `Sources/xdc` folder.

This file contains the floorplanning constraints (the `create_pblock`, `add_cells_to_pblock`, and `resize_pblock` commands) for the inst_count and inst_shift cells.

2-1-9. Uncomment lines 27, 28, and 29 to run the implementation processes in the `run_impl_config1.tcl` file:

```
opt_design

place_design

route_design
```

2-1-10. Uncomment line 32 to write the routed design checkpoint point for this implementation configuration:

```
write_checkpoint -force $OutputDir/
config1_routed_count_up_shift_left.dcp
```

This routed design checkpoint will be used to generate bitstreams for this configuration.

2-1-11. Uncomment lines 33 and 34 to write the routed design checkpoints for the RMs:

```
write_checkpoint -cell inst_count -force $OutputDir/
config1_inst_count_a_route_design.dcp

write_checkpoint -cell inst_shift -force $OutputDir/
config1_inst_shift_a_route_design.dcp
```

Writing these checkpoints is optional. These can be used to analyze how the placement and routing are performed for just the RMs.

2-1-12. Uncomment lines 37, 38, and 39 to prepare the static routed design checkpoint:

```
update_design -cell inst_count -black_box

update_design -cell inst_shift -black_box

lock_design -level routing
```

The `update_design` command will replace the RMs with a black box; that is, it will remove the RM place and route data from the fully routed design checkpoint.

The `lock_design -level routing` command will lock down the placement and routing results (preserve) of the static logic (in memory at this stage) from the first configuration.

2-1-13. Uncomment line 42 to write the static routed design checkpoint:

```
write_checkpoint -force $OutputDir/config1_static_routed.dcp
```

The static implementation checkpoint of the static logic will be used by subsequent configurations.

2-1-14. Save the **run_impl_config1.tcl** file.

2-2. Run `run_impl_config1.tcl`.

2-2-1. Enter the following command in the Vivado Tcl shell to run the script:

```
vivado -mode batch -source run_impl_config1.tcl
```

This process will take approximately 2–3 minutes.

Note that the LOG and JOU files will be saved in the launch directory.

The implemented checkpoints will be stored in the `./Checkpoints` directory.

2-2-2. Browse to the `$TRAINING_PATH/dfx_flow/lab/ZCU104/Checkpoints` directory and verify that all four routed checkpoints are generated.

2-2-3. Enter the following command in the Vivado Tcl shell to open the routed checkpoint for the `config1_routed_count_up_shift_left.dcp` configuration1 file:

```
open_checkpoint ./Checkpoints/
config1_routed_count_up_shift_left.dcp
```

2-2-4. Enter **`start_gui`** in the Vivado Tcl shell.

The Vivado IDE opens with the *config1_routed_count_up_shift_left* checkpoint loaded.

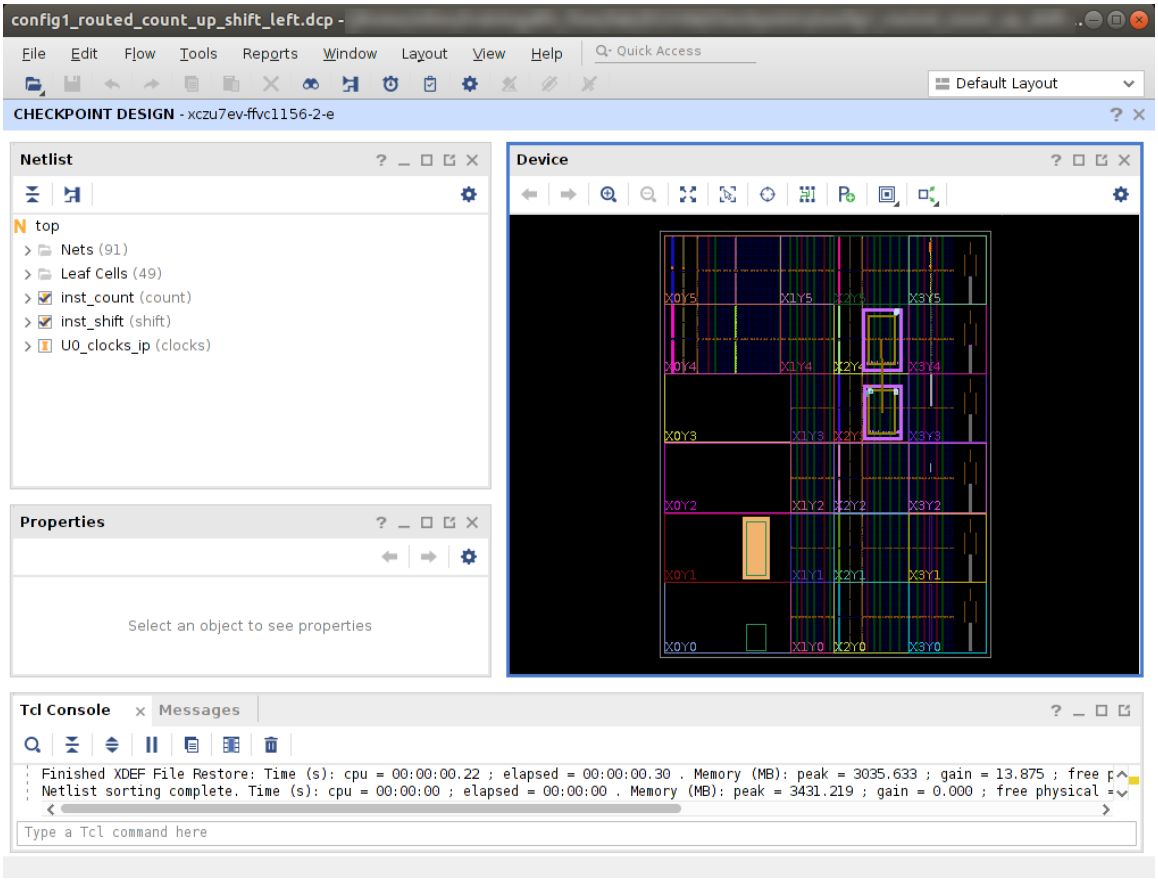


Figure 1-3: Vivado GUI with Initial Configuration Implemented Checkpoint

2-3. Analyze the placement and routing of the initial configuration.

2-3-1. Select the **inst_count** Pblock in the Device view to view the properties of this Pblock.

Note: Do not select inst_count from the Netlist tab because it provides the properties of the inst_count instance, not the Pblock.

2-3-2. Select **Window** > **Properties** to open the Properties window for the selected Pblock.

2-3-3. Select the **General** tab in the Properties window.

2-3-4. Observe that although this reconfigurable module only requires CLB resources, RAMB18 and RAMB36 are also included in the Pblock.

This allows the routing resources for these block types to be included in the reconfigurable region.

2-3-5. Select the **Properties** tab in the Pblock Properties window.

2-3-6. Observe that SNAPPING_MODE in set to **ON**.

The Pblock property SNAPPING_MODE automatically resizes the Pblock to make sure that the Pblocks are not splitting the paired interconnect tiles. Splitting the paired interconnect tiles is called a back-to-back violation. The SNAPPING_MODE property helps to avoid this back-to-back violation.

2-3-7. Select **Reports** > **Report DRC** to run DRC checks on the initial configuration.

2-3-8. Select only the **Dynamic Function eXchange** option in the Rules section of the Report DRC dialog box.

2-3-9. Click **OK** in the dialog box to run the DRC checks.

You may see some advisory messages, but you can still continue.

2-3-10. Enter **`close_project`** in the Tcl Console of the Vivado IDE.

This command will close the active design from memory.

2-3-11. Enter **`stop_gui`** in the Tcl Console of the Vivado IDE to close the Vivado IDE.

2-3-12. Close all opened files in the text editor.

Do not save anything if prompted.

Implementing the Second Configuration Step 3

In this step, you will review the second configuration script and run the script to generate the routed design checkpoint for the second configuration.

The `run_impl_config2.tcl` script is for the second configuration implementation (with the count_down and shift_right RM variations).

3-1. Review `run_impl_config2.tcl`.

3-1-1. Open **`run_impl_config2.tcl`** using your preferred text editor.

Note that `config1_static_routed.dcp` is opened instead of all the synthesized design points being read. In the DFX flow, for the second and subsequent configurations, opening the static routing locked checkpoint of the initial configuration is recommended. This will ensure that the static design remains completely identical from configuration to configuration.

3-1-2. Close the **`run_impl_config2.tcl`** file.

Do not save anything if prompted.

3-2. Run `run_impl_config2.tcl`.

3-2-1. Enter the following command in the Vivado Tcl shell to run the script:

```
vivado -mode batch -source run_impl_config2.tcl
```

This process will take approximately 2–3 minutes.

This script implements the second configuration and then generates the routed checkpoint (`./Checkpoints/config2_routed_count_down_shift_right.dcp` file).

Generating All Bitstreams Step 4

Here, you will verify the two configurations and generate the full and partial bitstreams using the provided Tcl scripts.

Once all the configurations have been completely placed and routed, a final verification check using the `pr_verify` command can be performed to validate the consistency between these configurations.

The provided `generate_bitstreams.tcl` script will generate the bitstreams for both configurations.

4-1. Verify the configurations.

4-1-1. Enter the following commands in the Vivado Tcl shell:

```
set OutputDir "./Checkpoints"

pr_verify -full_check $OutputDir/
config1_routed_count_up_shift_left.dcp $OutputDir/
config2_routed_count_down_shift_right.dcp -file $OutputDir/
pr_verify_c1_c2.log
```

4-1-2. Browse to the `./Checkpoints` folder.

4-1-3. Open and review the **`pr_verify_c1_c2.log`** file using your preferred text editor.

Note that the both configurations are compatible.

Tip: If you have more than two configurations, use the `pr_verify` command with the `-initial` and `-additional` options:

```
pr_verify -full_check -initial config1.dcp -additional
{config2.dcp, config3.dcp, etc} -file multi_config_verify.log
```

4-2. Review `generate_bitstreams.tcl`.

4-2-1. Browse to the `$TRAINING_PATH/dfx_flow/lab/ZCU104` directory.

4-2-2. Open **`generate_bitstreams.tcl`** using your preferred text editor.

4-2-3. Review the script.

4-2-4. Close the **`generate_bitstreams.tcl`** file.

Do not save anything if prompted.

4-3. Run `generate_bitstreams.tcl`.

4-3-1. Enter the following command in the Vivado Tcl shell to run the script:

```
vivado -mode batch -source generate_bitstreams.tcl
```

This process will take approximately 7–8 minutes. It will take approximately 2–3 minutes to generate the bitstream for each configuration.

Programming the Device and Evaluating the Design Step 5

This final step is the actual downloading of both the full bitstream and the partial bitstreams. Because this design has two relatively independent sections, you will notice the non-reconfiguring design continuing to run while the new partial bitstream is loaded into the other RP.

Set up and connect the ZCU104 evaluation board. Verify that this has been done properly before turning on the power.

For CloudShare users, if you have the evaluation board locally available, copy the project directory to the local path and proceed with the steps below. You can also review the "How to Perform Board Labs on Your Local Machine" document available in the Overview tab of the CloudShare environment.

Otherwise, you can just review (without performing) the connecting the board, programming the device, and verifying the design on the hardware sections shown below.

5-1. Set the switches on the ZCU104 to boot from JTAG.

5-1-1. Ensure that the four positions on SW6 are as shown in the figure below to boot from JTAG.

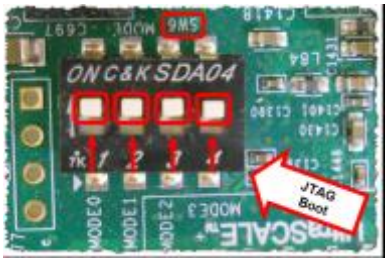


Figure 1-4: JTAG Boot Settings

5-2. Connect the board to your machine.

5-2-1. Connect the USB cable to the USB JTAG connector on the evaluation board.

5-2-2. Ensure that the power cord is plugged in and turn on the evaluation board.

5-2-3. Make sure that the board settings are correct.

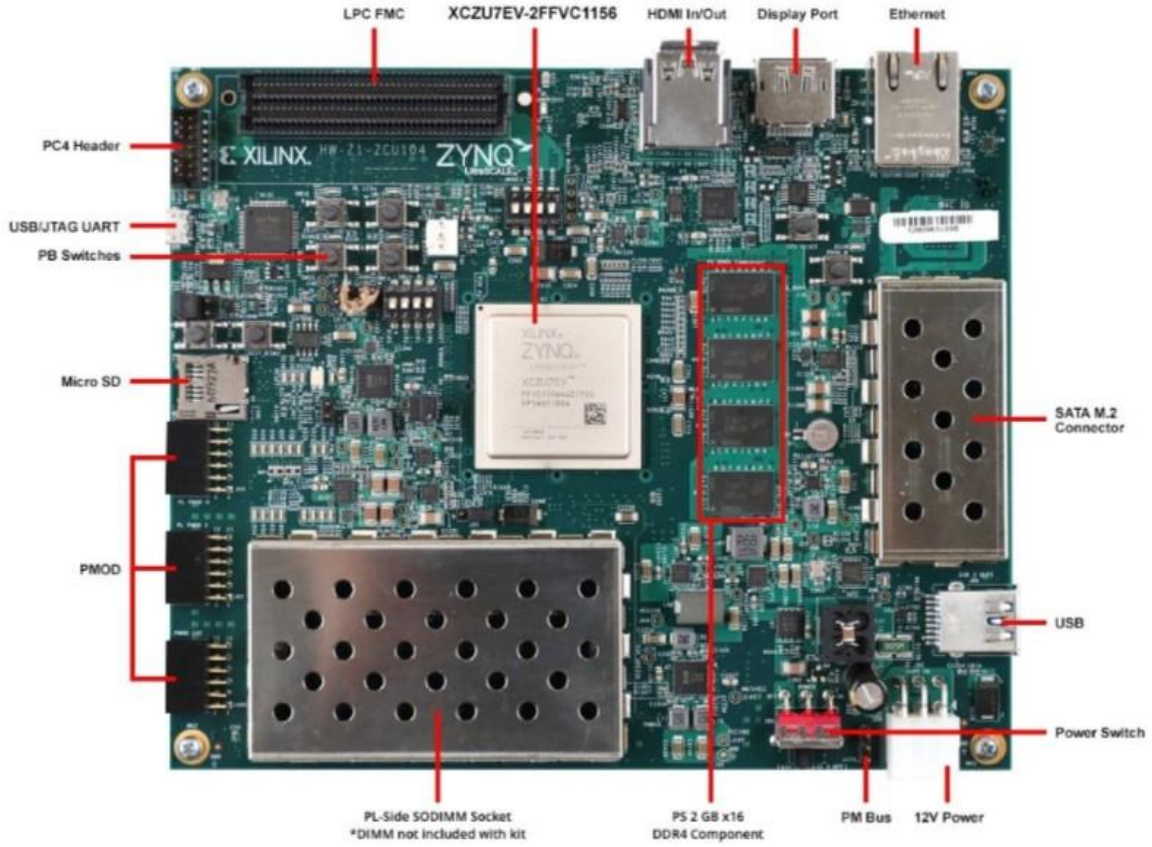


Figure 1-5: ZCU104 Evaluation Board

A hardware manager is the portion of the Vivado Design Suite that enables the monitoring of cores that were added to a design.

5-3. Launch the Vivado hardware manager.

5-3-1. Enter **`start_gui`** in the Vivado Tcl shell.

5-3-2. Click **Open Hardware Manager** under the Tasks section in the Vivado Design Suite Welcome page.

The Hardware Manager window opens.

The hardware needs to be connected and the information bar invites you to open an existing or a new target.

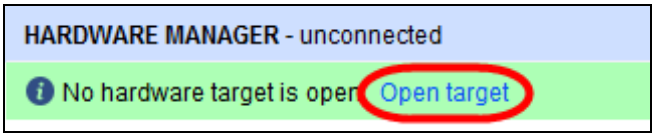


Figure 1-6: Opening a Hardware Target

5-4. Connect the target through the New Target Wizard to guide you through the process.

5-4-1. Click **Open target** > **Open New Target**.

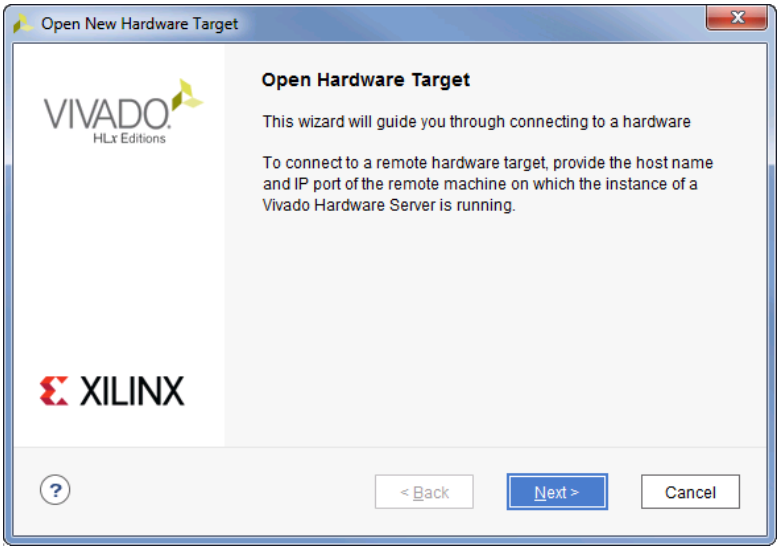


Figure 1-7: Open Hardware Target Dialog Box

5-4-2. Click **Next** to set the hardware server settings.

5-4-3. Enter a name for the server.

Typically, this is left at its default value.

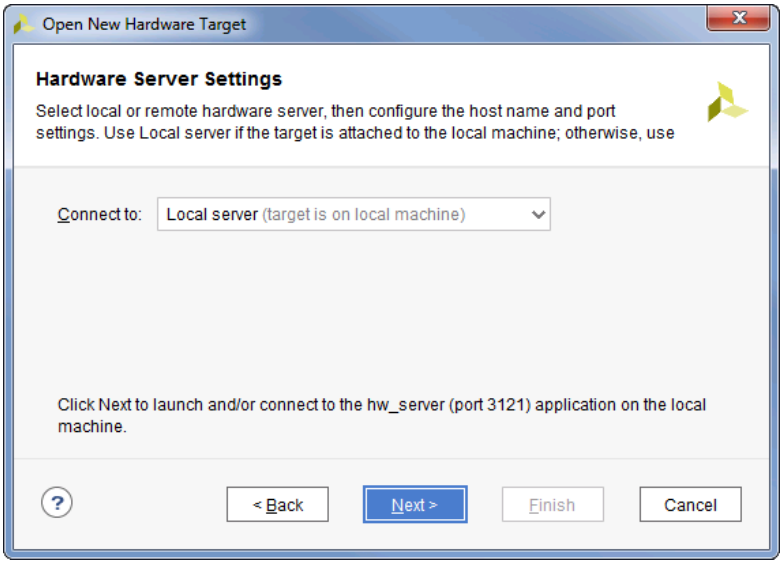


Figure 1-8: Setting the Server Name for the New Hardware Target

5-4-4. Click **Next** to select the hardware target.

5-4-5. Verify the hardware target.

This becomes important when there are multiple targets connected to the PC.

You can change the frequency of the JTAG cable if you are experiencing communications problems.

[Linux users]: If you receive an error saying that no active target is found, check the USB connections by selecting **Devices** > **USB** in the VirtualBox toolbar at the top.

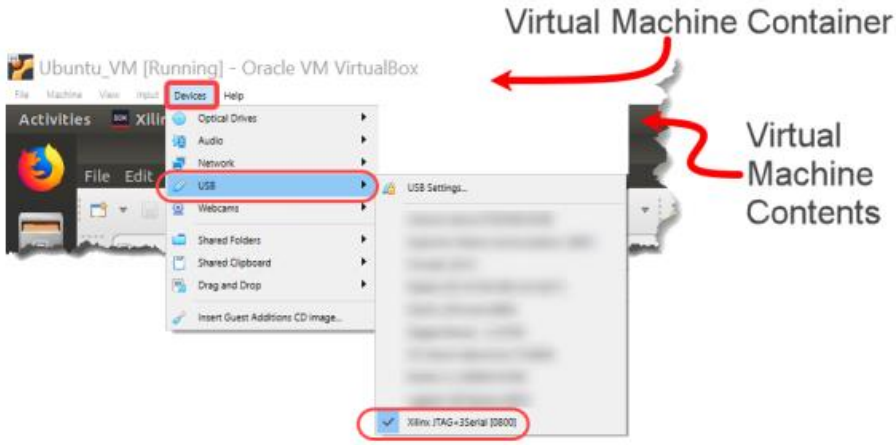


Figure 1-9: Selecting the Hardware Target

5-4-6. Leave the frequency at its default value.

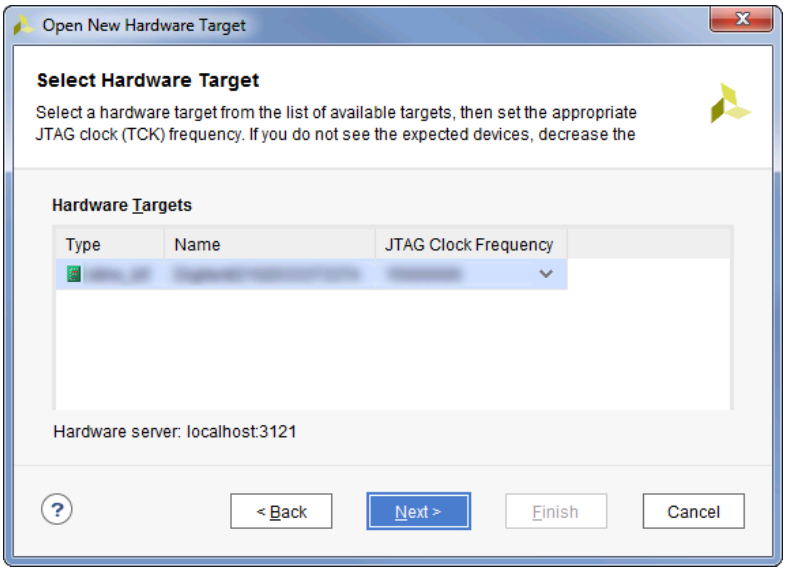


Figure 1-10: Selecting the Hardware Target

5-4-7. Click **Next** to view the hardware target summary.

A summary of the connection is displayed.

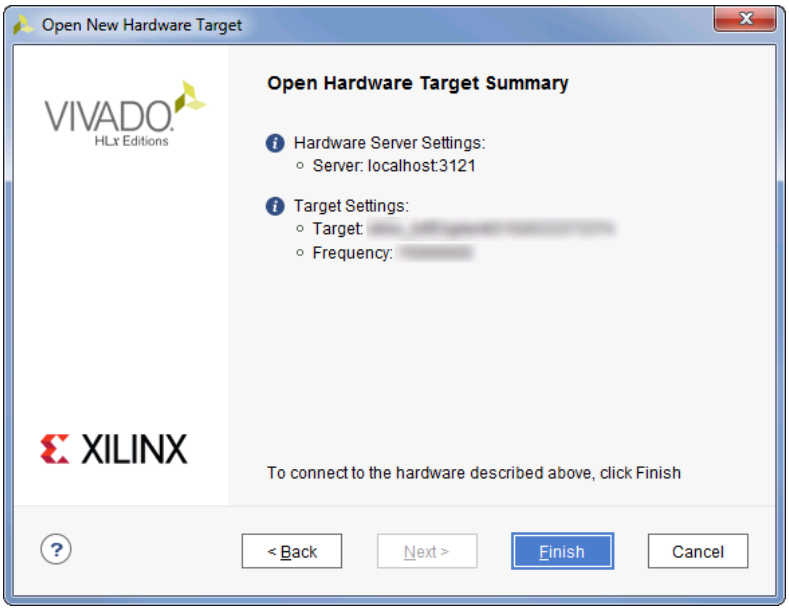


Figure 1-11: Summary of the Open Hardware Target Settings

5-4-8. Click **Finish** to connect to the new hardware target.

5-5. Program the device connected from the hardware session.

5-5-1. Right-click **xczu7_0** and select **Program Device**.

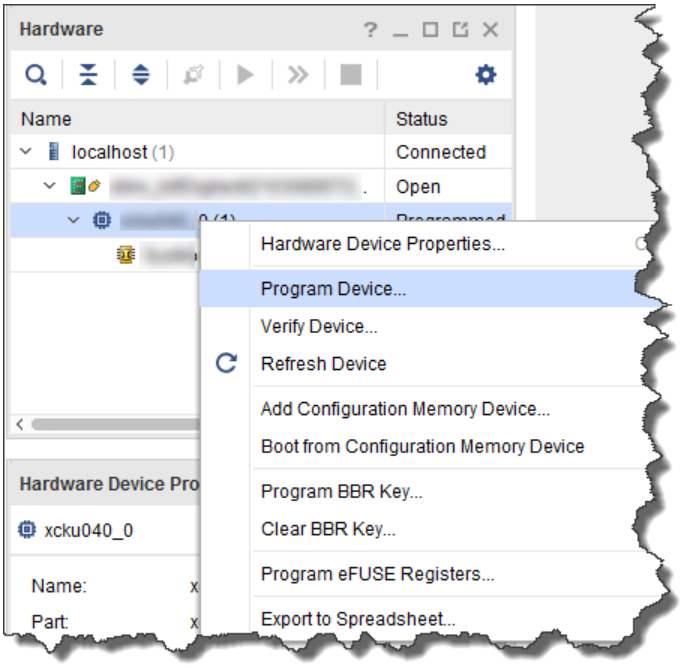


Figure 1-12: Programming the Device

The Program Device dialog box opens.

5-5-2. Browse to the `./Checkpoints` directory in the Program Device dialog box and select **`config1_count_up_shift_left.bit`**.

This is the full design bitstream that is used to configure the device from power up.

5-5-3. Click **OK** to select the bitstream.

5-5-4. Leave the debug probes file field empty as there are no debug cores in the design.

5-5-5. Click **Program**.

A progress bar appears and, when complete, the dialog box closes.

Notice that two GPIO LEDs are performing up counter operation on LEDs 2-3 and other two LEDs are performing shift left operation on the LEDs 0-1. Note the time taken to configure full device.

5-6. Program the FPGA with partial bitstreams and verify the functionality.

5-6-1. Repeat the previous steps to program the FPGA with the the `config2_count_down_shift_right_pblock_shift_partial.bit` partial bitstream.

Note that the time taken for programming with the partial bitstream is much less.

5-6-2. Verify on the GPIO LEDS 0-1 that the shift pattern has changed from left to right.

5-6-3. Program the FPGA with the `config2_count_down_shift_right_pblock_count_partial.bit` partial bitstream.

5-6-4. Verify on the GPIO LEDS 2-3 that the up counter changed to a down counter.

5-6-5. Verify other partial bitstreams by configuring one by one into the FPGA.

5-6-6. Enter **`stop_gui`** in the Tcl console.

5-7. Close the Vivado Tcl shell.

5-8. Power off the board.

Some systems (particularly VMs) may be memory constrained. Removing the workspace frees a portion of the disk space, allowing other labs to be performed.

You can delete the directory containing the lab you just ran by using the graphical interface or the command-line interface. You can choose either mechanism. Both processes will recursively delete all the files in the `$TRAINING_PATH/dfx_flow` directory.

5-9. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

5-9-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$TRAINING_PATH/dfx_flow`.

5-9-2. Select **`dfx_flow`**.

5-9-3. Press <**Delete**>.

-- OR --

Using the command line:

5-9-4. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>).

5-9-5. Enter the following command to delete the contents of the workspace:

[Windows users]: **`rd /s /q $TRAINING_PATH/dfx_flow`**

[Linux users]: **`rm -rf $TRAINING_PATH/dfx_flow`**

Summary

Through completing this lab you became more familiar with the DFX flow in the Vivado Tcl shell and saw how a simple Tcl script file can be used to assemble the netlists in a consistent fashion.

You created multiple configurations through implementation, validated routed DCPs, and generated bitstreams. You then used the Vivado hardware manager to download both full and partial bitstreams to the board.

Answers

1. What are the module names in *shift_left.v* and *shift_right.v*?

   The module name is **shift** and it is same in both reconfigurable modules. The Dynamic Function eXchange flow requirement is that the name of the block (module) must be the same in each instance, and all the properties of the interfaces (names, widths, direction) must also be identical.

2. Where will the synthesized design checkpoints be stored?

   The synthesized checkpoint will be stored in the `./SynOutputDir/post_synth_<RM_variant_name>` directory.

3. What is the difference between the `open_checkpoint` and `read_checkpoint` commands?

   The `open_checkpoint` command reads in the static design checkpoint and opens it in active memory.

   The `read_checkpoint` command simply reads the associated checkpoint file (similar to other `read_*` commands, such as `read_verilog`, `read_xdc`, etc.) without opening a design or project in memory.