Reconfiguring Processor Peripheral Lab

Introduction

In this lab, you will use Vivado IPI and Software Development Kit to create a reconfigurable peripheral using ARM Cortex-A9 processor system on Zynq. You will use Vivado IPI to create a top-level design, which includes the Zynq processor system as a sub-module. During the PR flow, you will define one Reconfigurable Partition having two Reconfigurable Modules (addition and multiplication). You will create multiple Configurations and run the Partial Reconfiguration implementation flow to generate full and partial bitstreams. You will use ZedBoard and/or Zybo to verify the design in hardware using a SD card to initially configure the FPGA, and then partially reconfigure the device using the PCAP under user software control.

Objectives

After completing this lab, you will be able to:

- Use a Tcl script to generate a Vivado IPI design, create a wrapper file from it and generate the design checkpoint
- Use Vivado's bottom-up methodology to synthesize the necessary RMs
- Floorplan the design
- Add the desired RMs
- Create multiple configurations
- Implement the design and generate full and partial bitstreams for various configurations
- Download bitstreams to demonstrate a working partial reconfigurable design

Design Description

The purpose of this lab exercise is to implement a design that can be dynamically reconfigurable using PCAP resource and PS sub-system. The system consists of one peripheral (math functions), having two unique capabilities (addition and multiplication). The user verifies the functionality using a user application. The dynamic modules are reconfigured using the PCAP resource available through Device Configuration block.

The design is shown in Figure 1.

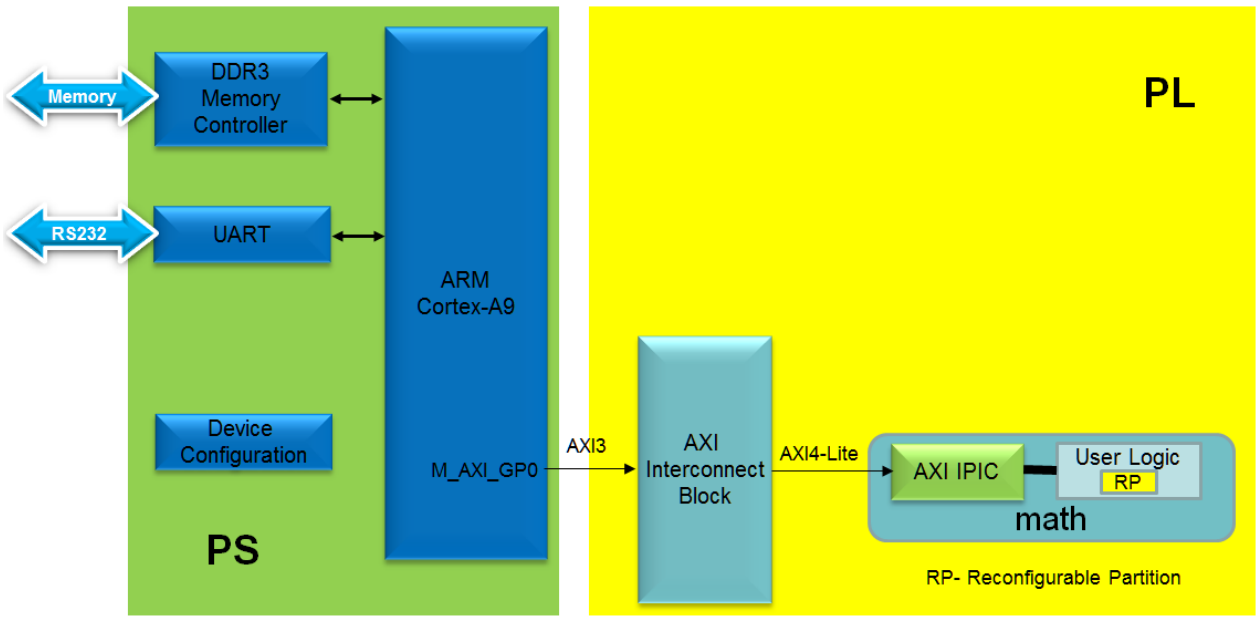
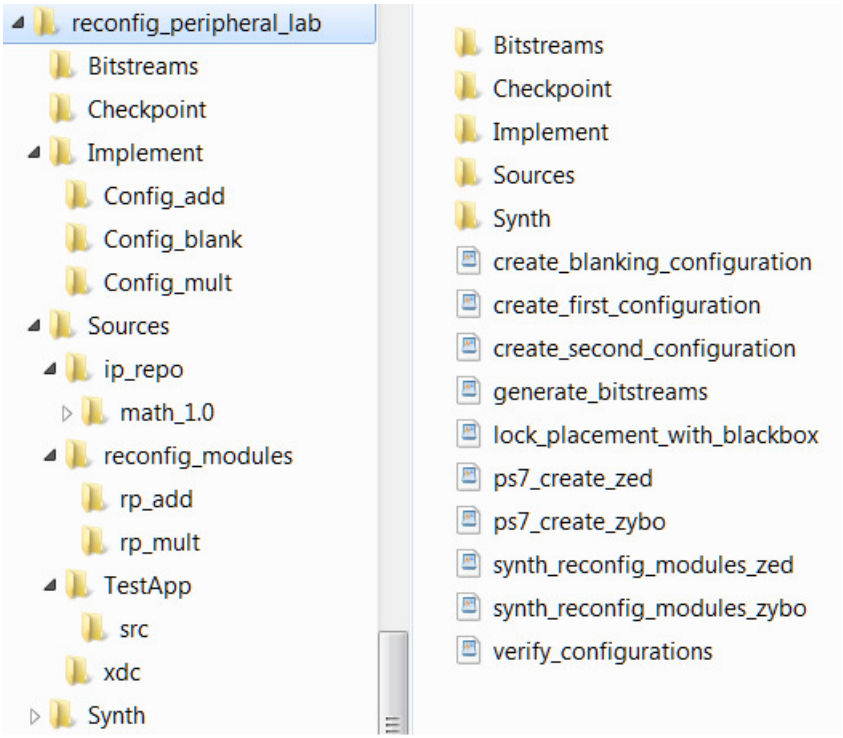


Figure 1. The design

The directory structure is as shown here:

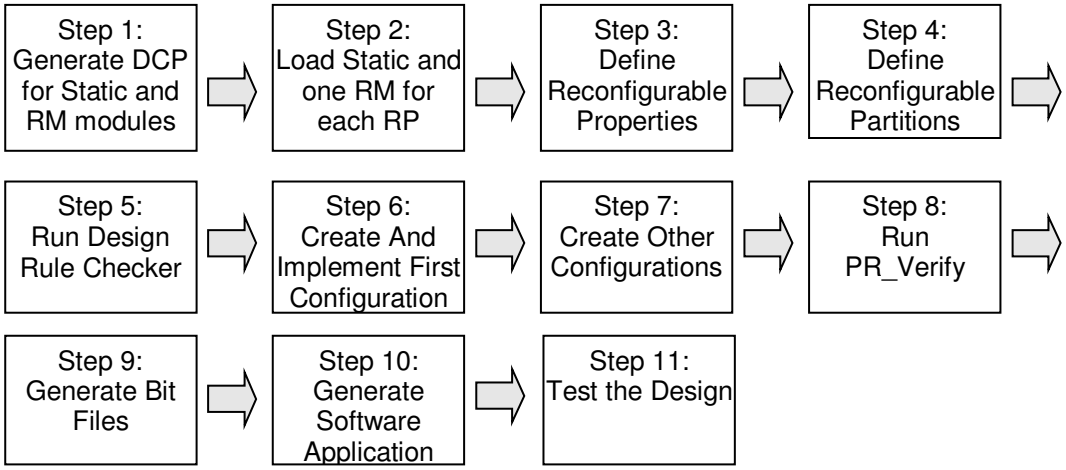


The Sources directory provides the math processor core (in ip_repo directory), source file for add and mult functions (in reconfig_modules > rp_add and reconfig_modules > rp_mult directories), the software application (in TestApp directory), and a place holder for the floorplan constraints (in xdc directory). The Synth and its sub-directories structure will hold the synthesized checkpoints, the Implement and its sub-directories will hold the implemented configurations, the Checkpoint will hold the static, and the two configuration checkpoints, and the Bitstreams directory will hold the generated full and partial bitstreams. In the home directory, there are several Tcl scripts which will perform several tasks including the processor system creation and the bottom-up synthesis of the reconfigurable modules.

Procedure

This lab is separated into steps that consist of general overview statements that provide information on the detailed instructions that follow. Follow these detailed instructions to progress through the lab.

General Flow for this Lab



Generate DCPs for the Static Design and RM Modules Step 1

1-1. Start Vivado and execute the provided Tcl script to create the design check point for the static design having one RP.

1-1-1. Open **Vivado** by selecting **Start > All Programs > Xilinx Design Tools > Vivado 2015.2 > Vivado 2015.2**

1-1-2. In the Tcl Shell window enter the following command to change to the lab directory and hit **Enter**.

```
cd c:/xup/PR/labs/reconfig_peripheral_lab
```

1-1-3. Generate the PS design executing the provided Tcl script.

`source ps7_create_zed.tcl` (for ZedBoard) `or`

`source ps7_create_zybo.tcl` (for Zybo)

This script will create the block design called system, instantiate ZYNQ PS with SD 0 and UART 1 interfaces enabled. It will also enable the GP0 interface along with FCLK0 and RESET0_N ports. The provided math IP will then be instantiated. It will then create a top-level wrapper file called system_wrapper.v which instantiates the system.bd (the block design).

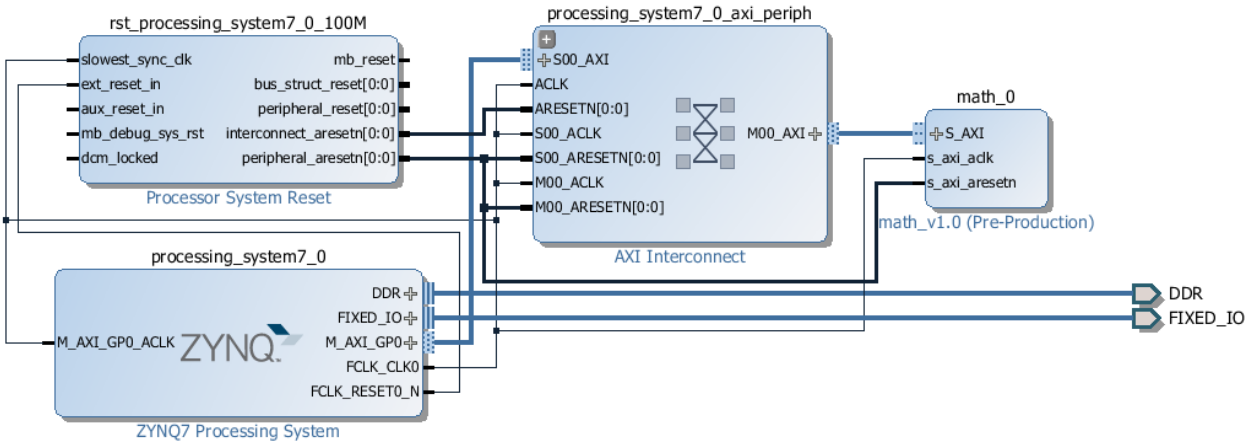


Figure 3. The system block design

1-1-4. Click **Run Synthesis** under the *Synthesis* group in the *Flow Navigator* to run the synthesis process.

Wait for the synthesis to complete. When done click **Cancel**.

1-1-5. Using the windows explorer, copy the **system_wrapper.dcp** file from *reconfig_peripheral_<board>_lab\reconfig_peripheral_<board>_lab.runs\synth_1* into the *Synth\Static* directory under the current lab directory. Use zed or zybo for <board>.

1-1-6. Close the project by typing the `close_project` command in the Tcl console or selecting **File > Close Project**.

1-2. Since we have RMs in HDL format, we need to synthesize them and generate the dcp for each of the RMs. The generated dcps should be stored in appropriate directories so they can be accessed correctly; particularly, the dcp files for RM must be in separate directories as their dcp file names will be same for a given RP.

1-2-1. In the Tcl Shell window enter the following command to change to the lab directory and hit **Enter**.

```
cd c:/xup/PR/labs/reconfig_peripheral_lab
```

1-2-2. Synthesize each of the RMs (two) executing the provided Tcl script.

`source synth_reconfig_modules_zed.tcl` (for ZedBoard) or

`source synth_reconfig_modules_zybo.tcl` (for Zybo)

This script will synthesize the HDL files for each RM in an out of context mode and write the design checkpoint (dcp) in the respective destination folder under the Synth directory. After each RM's dcp is generated, the respective design is closed.

1-2-3. At this point the directory content will look like shown below.

Figure 4. Synth directory hierarchy and content

Load Static and one RM for the RP in Vivado — Step 2

Since all required netlist files (dcp) for the design are now available, you will use Vivado to floorplan the design, define Reconfigurable Partitions, add Reconfigurable Modules, run the implementation tools, and generate the full and partial bitstreams.

2-1. In this step you will load the static and one RM designs for the RP.

2-1-1. In the Tcl Shell window enter the following command to change to the lab directory and hit **Enter**.

```
cd c:/xup/PR/labs/reconfig_peripheral_lab
```

2-1-2. Load the static design using the **open_checkpoint** command.

```
open_checkpoint Synth/Static/system_wrapper.dcp
```

You can see the design structure in the Netlist pane with one black box for the **rp_instance** module.

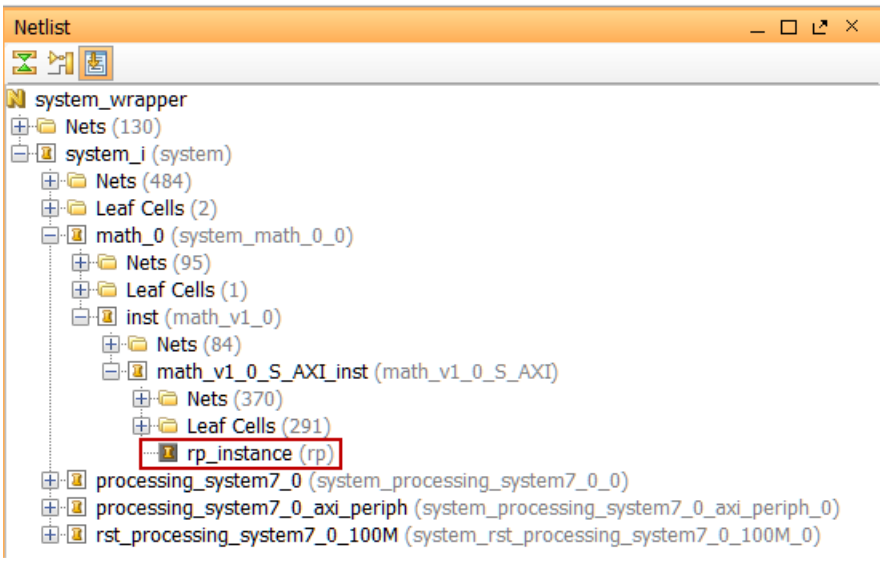


Figure 4. Static design with a black box

2-1-3. Select the *rp_instance* instance and then select the *Properties* tab in the **Cell Properties** window. Note that the *IS_BLACKBOX* checkbox is checked.

2-1-4. Load one RM for the RP by using the **read_checkpoint** command.

```
read_checkpoint -cell
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance
Synth/reconfig_modules/rp_add/add_mult_synth.dcp
```

You can now see the design structure in the Netlist pane with an RM for the *rp_instance* module loaded.

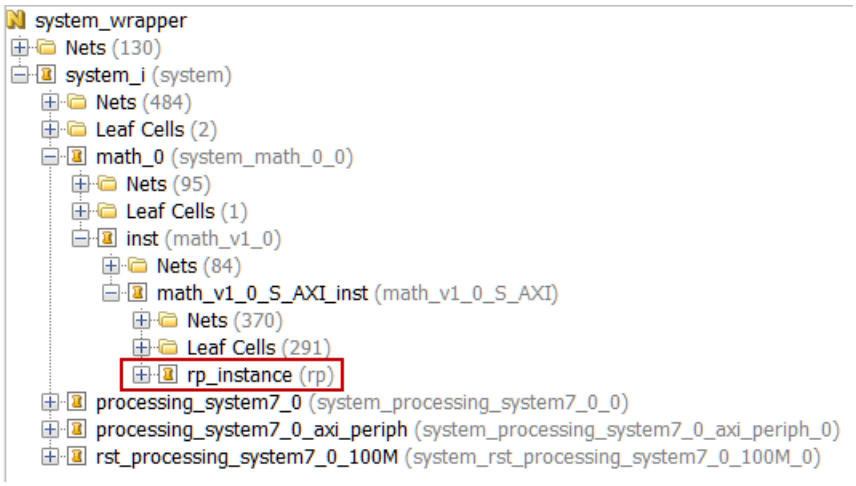


Figure 5. Static design with RM loaded

2-1-5. Select the *rp_instance* instance and then select the *Properties* tab in the **Cell Properties** window. Note that the *IS_BLACKBOX* checkbox is not checked since a RM design is loaded.

2-1-6. Select the *Statistics* tab and note the amount and type of resources the module uses.

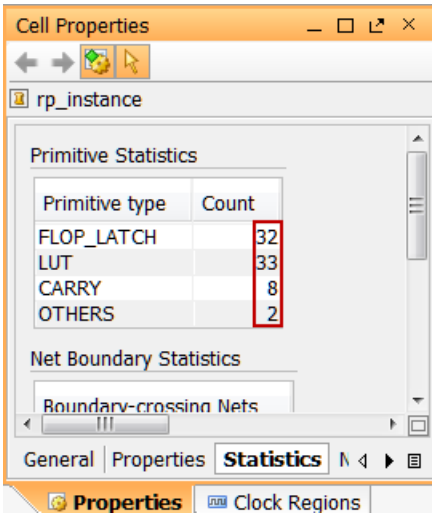


Figure 6. Resources used by the loaded RM

Define Reconfigurable Properties on each RM — Step 3

3-1. In this design you have one Reconfigurable Partition having two RMs. Define the reconfigurable properties to the loaded RM.

3-1-1. Define each of the loaded RMs (submodules) as partially reconfigurable by setting the **HD.RECONFIGURABLE** property using the following commands.

```
set_property HD.RECONFIGURABLE 1 [get_cells
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance]
```

This is the point at which the Partial Reconfiguration license is checked.

3-1-2. Select the `rp_instance` instance and notice that the DONT_TOUCH checkbox is selected in the **Cell Properties** window.

3-1-3. Save the assembled design state for this initial configuration using the following command.

```
write_checkpoint Checkpoint/top_link_add.dcp
```

This will write the *dcp* file in the provided **Checkpoint** directory.

Define the Reconfigurable Partition Region — Step 4

4-1. Next you must floorplan the RP region. Depending on the type and amount of resources used by all the RMs for the given RP, the RP region must be appropriately defined so it can accommodate any RM variant.

4-1-1. You execute the following command to define the region, perform the DRC and go to **Step 6, OR** continue following the step-by-step instructions.

`read_xdc Sources/xdc/fplan_zed.xdc` (for ZedBoard) or

`read_xdc Sources/xdc/fplan_zybo.xdc` (for Zybo) or

4-1-2. Select **Edit > Find**.

For ZedBoard: In the *Find* field, select **Sites** in the *Find* drop-down box, then *Name* and enter ***SLICE_X34Y109**, click on the **+** button, then select **OR** using the drop-down button, *Name* again, ***SLICE_X39Y123**, and finally click **OK**.

For Zybo: In the *Find* field, select **Sites** in the *Find* drop-down box, then *Name* and enter ***SLICE_X8Y50**, click on the **+** button, then select **OR** using the drop-down button, *Name* again, ***SLICE_X13Y64**, and finally click **OK**.

You will see a new tab, called Sites – Find will appear showing two entries.

4-1-3. Select one entry at a time, right-click and select **Mark**.

You will see marked sites in the Device window. You may have to zoom out.

4-1-4. Select the **rp_instance** instance in the *Netlist* window, right-click, and select *Floorplanning* > *Draw Pblock*.

4-1-5. Draw a box that bounds `SLICE_X34Y109:SLICE_X39Y123` (for ZedBoard) or `SLICE_X8Y50:SLICE_X13Y64` (for Zybo) marked in the previous step.

4-1-6. Click **OK** to include SLICE as well as DSP48 slices as the resources to be reconfigured. The DSP48 slices are required for the multiplier RM.

Run Design Rule Checker Step 5

5-1. It is always good idea to run a design rule checker so you can catch errors as soon as possible.

5-1-1. Select **Tools > Report > Report DRC.**

5-1-2. Deselect **All Rules**, select **Partial Reconfiguration**, and then click **OK** to run the PR-specific design rules.

You should not see any error.

5-1-3. Save the Pblocks and associated properties by issuing the following command.

`write_xdc Sources/xdc/fplan_<board>.xdc` Use **zed** (ZedBoard) or **zybo** (Zybo.) for <board>.

Create and Implement First Configuration Step 6

6-1. Create and implement the first Configuration.

6-1-1. Execute the following command.

```
source create_first_configuration.tcl
```

The script will do the following tasks:

- The script will optimize, place and route the design by executing the following commands.

  ```
  opt_design

  place_design

  route_design
  ```

- Once the design is implemented (placed and routed), zoom in into the Device view to see the *pblock_rp_instance*.

  You will see the introduction of Partition Pins. These are the physical interface points between static and reconfigurable logic. They are anchor points within an interconnect tile

through which each IO of the reconfigurable module must route. They appear as white boxes in the placed design view.

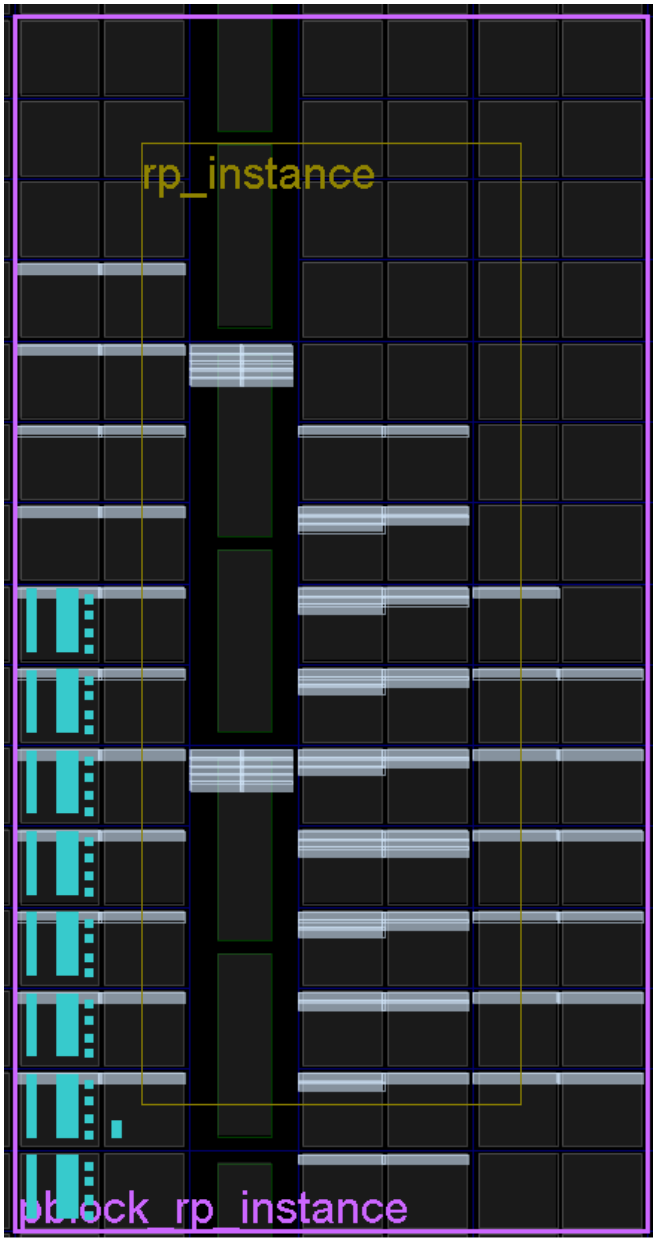


Figure 7. Partition pins in pblock_rp_instance

- Save the full design checkpoint.

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

- Save checkpoints for the reconfigurable module.

```
write_checkpoint -force -cell
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance
Checkpoint/rp_instance_add_route_design.dcp
```

At this point, a fully implemented partial reconfiguration design from which full and partial bitstreams can be generated is ready. The static portion of this configuration **must** be

used for all subsequent configurations, and to isolate the static design, the current reconfigurable module must be removed.

6-2. After the first configuration is created, the static logic implementation will be reused for the rest of the configurations. So it should be saved. But before you save it, the loaded RM should be removed.

6-2-1. Execute the following command to update the design with the blackbox and write the checkpoint.

```
source lock_placement_with_blackbox.tcl
```

The script will do the following tasks:

- Clear out the existing RMs executing the following commands.

  ```
  update_design -cell
  system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance -black_box
  ```

  Issuing this command will result in design changes including, the number of Fully Routed nets (green) decreased, the number of Partially Routed nets (yellow) has increased, and *rp_instance* may appear in the Netlist view as empty.

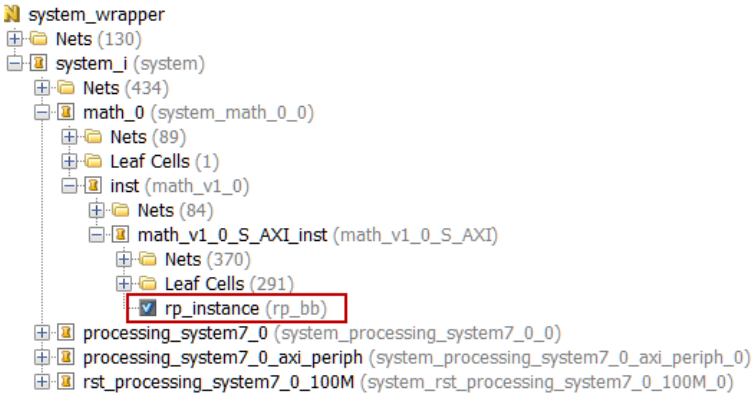


Figure 8. The design with unloaded module

- Lock down all placement and routing by executing the following command.

  ```
  lock_design -level routing
  ```

  Because no cell was identified in the `lock_design` command, the entire design in memory (currently consisting of the static design with black boxes) is affected.

- Write out the remaining static-only checkpoint by executing the following command.

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

  This static-only checkpoint would be used for any future configuration, but here, you simply keep this design open in memory.

Create Other Configurations Step 7

7-1. Read next set of RM dcp, create and implement the second configuration.

7-1-1. Execute the following command to create and implement the second configuration

```
source create_second_configuration.tcl
```

The script will do the following tasks:

- With the locked static design open in memory, read in post-synthesis checkpoint for the second reconfigurable module.

```
read_checkpoint -cell
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance
Synth/reconfig_modules/rp_mult/add_mult_synth.dcp
```

- Optimize, place and route the design by executing the following commands.

```
opt_design

place_design

route_design
```

- Save the full design checkpoint.

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

- Save the checkpoint for the reconfigurable module.

```
write_checkpoint -force -cell
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance
Checkpoint/rp_instance_mult_route_design.dcp
```

- Close the project

```
close_project
```

7-2. Create the blanking configuration.

7-2-1. Execute the following command to create and implement the second configuration

```
source create_blanking_configuration.tcl
```

The script will do the following tasks:

- Open the static route checkpoint.

```
open_checkpoint Checkpoint/static_route_design.dcp
```

- For creating the blanking configuration, use the `update_design -buffer_ports` command to insert LUTs tied to constants to ensure the outputs of the reconfigurable partition are not left floating.

```
update_design -buffer_ports -cell
system_i/math_0/inst/math_v1_0_S_AXI_inst/rp_instance
```

- Now place and route the design. There is no need to optimize the design.

```
place_design

route_design
```

The base (or blanking) configuration bitstream, when we generate in the next section, will have no logic for either reconfigurable partition, simply outputs driven by ground. Outputs can be tied to VCC if desired, using the HD.PARTPIN_TIEOFF property.

- Save the checkpoint in the `Config_blank` directory.

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

- Close the project

```
Close_project
```

Run PR_Verify — Step 8

8-1. You must ensure that the static implementation, including interfaces to reconfigurable regions, is consistent across all Configurations. To verify this, you run the PR_Verify utility

8-1-1. Run the **pr_verify** command from the Tcl Console.

```
source verify_configurations.tcl
```

The script will perform the following tasks:

- execute the pr_verify command and then close the project:

```
pr_verify -initial Implement/Config_add/top_route_design.dcp -
additional {Implement/Config_mult/top_route_design.dcp
Implement/Config_blank/top_route_design.dcp}
```

You should see the message indicating the `Config_add` configuration is compatible with `Config_mult`, and the `Config_add` configuration is compatible with `Config_blank`.

- Execute the following command to close the project.

```
close_project
```

Generate Bit Files Step 9

9-1. After all the Configurations have been validated by PR_Verify, full and partial bit files must be generated for the entire project

9-1-1. Generate the full configurations and partial bitstreams by executing the following tcl script.

```
source generate_bitstreams.tcl
```

9-1-2. The script will do the following tasks:

- Read the first configuration in the memory, using the command:

  ```
  open_checkpoint Implement/Config_add/top_route_design.dcp
  ```

- Generate the full and partial bitstreams for this design.

  ```
  write_bitstream -file Bitstreams/Config_add.bit

  write_cfgmem -format BIN -interface SMAPx32 -disablebitswap
  -loadbit "up 0
  Bitstreams/Config_add_pblock_rp_instance_partial.bit"
  Bitstreams/add.bin

   close_project
  ```

  Notice the three bitstreams will be created.

  Config_add.bit – This is the power-up, full design bitstream.
  Config_add_pblock_rp_instance_partial.bit – This is the partial bit file for the *adder* module.
  add.bin – This is the partial bit file for the *adder* module in the bin format.

- Generate full and partial bitstreams for the second configuration.

  ```
  open_checkpoint Implement/Config_mult/top_route_design.dcp

  write_bitstream -file Bitstreams/Config_mult.bit

  write_cfgmem -format BIN -interface SMAPx32 -disablebitswap
  -loadbit "up 0
  Bitstreams/Config_mult_pblock_rp_instance_partial.bit"
  Bitstreams/mult.bin

  close_project
  ```

  The three files will be created.

- Generate a full bitstream with black boxes, plus blanking bitstreams for the reconfigurable modules. Blanking bitstreams can be used to “erase” an existing configuration to reduce power consumption.

  ```
  open_checkpoint Implement/Config_blank/top_route_design.dcp

  write_bitstream -file Bitstreams/blanking.bit
  ```

```
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap
-loadbit "up 0
Bitstreams/blanking_pblock_rp_instance_partial.bit"
Bitstreams/blank.bin

close_project
```

Generate the Software Application Step 10

10-1. Open the PS design that was created in Step 1. Export the hardware design and launch SDK.

10-1-1. Click on the **Open Project** link, browse to *c:/xup/PR/labs/reconfig_peripheral_lab/reconfig_peripheral_<board>_lab*, select the *reconfig_peripheral<board>_lab.xpr* and click **OK** to open the design created in Step 1.

10-1-2. Select **File > Export > Export Hardware…**

10-1-3. In the *Export Hardware* form, do not check the *Include bitstream* checkbox and click **OK**.

10-1-4. Select **File > Launch SDK**

10-1-5. Click **OK** to launch SDK.

The SDK program will open. Close the Welcome tab if it opens.

10-2. Create a Board Support Package enabling generic FAT file system library.

10-2-1. In **SDK**, select **File** > **New** > **Board Support Package.**

10-2-2. Click **Finish** with the default settings (with standalone operating system).

This will open the Software Platform Settings form showing the OS and libraries selections.

10-2-3. Select **xilffs** as the FAT file support is necessary to read the partial bit files.

	Name	Version	Description
☐	lwip141	1.1	lwIP TCP/IP Stack library: lwIP v1.4.1
☑	xilffs	3.0	Generic Fat File System Library
☐	xilflash	4.0	Xilinx Flash library for Intel/AMD CFI com...
☐	xilisf	5.2	Xilinx In-system and Serial Flash Library
☐	xilmfs	2.0	Xilinx Memory File System
☐	xilrsa	1.1	Xilinx RSA Library
☐	xilskey	2.1	Xilinx Secure Key Library

Figure 10. Selecting the xilffs library support

10-2-4. Click **OK** to accept the settings and create the BSP.

10-3. Create an application.

10-3-1. Select **File** > **New** > **Application Project.**

10-3-2. Enter **TestApp** as the *Project Name*, and for *Board Support Package*, choose **Use Existing** (*standalone_bsp_0* should be the only option).

10-3-3. Click **Next**, and select *Empty Application* and click **Finish.**

10-3-4. Expand the **TestApp** entry in the project view, right-click the *src* folder, and select **Import.**

10-3-5. Expand **General** category and double-click on **File System.**

10-3-6. Browse to *c:\xup\PR\labs\reconfig_peripheral_lab\Sources\TestApp\src* and click **OK.**

10-3-7. Select **TestApp.c** and click **Finish** to add the file to the project.

The program should compile successfully.

Open the source file and verify that the bin file size in the program listed matches the size you made a note earlier (except it is 4x as the program uses the size in words). If different, then change in the program and save it.

10-4. Create a zynq_fsbl application.

10-4-1. Select **File** > **New** > **Application Project.**

10-4-2. Enter **zynq_fsbl** as the *Project Name*, and for *Board Support Package*, choose **Create New**.

10-4-3. Click **Next**, select *Zynq FSBL,* and click **Finish.**

This will create the first stage bootloader application called zynq_fsbl.elf

10-5. Create a Zynq boot image.

10-5-1. Select **Xilinx Tools** > **Create Zynq Boot Image.**

10-5-2. Click the Browse button of the Output BIF file path field, browse to *c:\xup\PR\labs\reconfig_peripheral_lab,* and then click **Save** with the *output* as the default filename.

10-5-3. Click on the **Add** button of the *Boot image partitions,* click the Browse button in the Add Partition form, browse to **c:\xup\PR\labs\reconfig_peripheral_lab\reconfig_peripheral_<board>_lab \reconfig_peripheral_<board>_lab.sdk\zynq_fsbl\Debug** directory, select *zynq_fsbl.elf* and click **Open**. Use **zed** (ZedBoard) or **zybo** (Zybo) for <board>.

Note the partition type is bootloader.

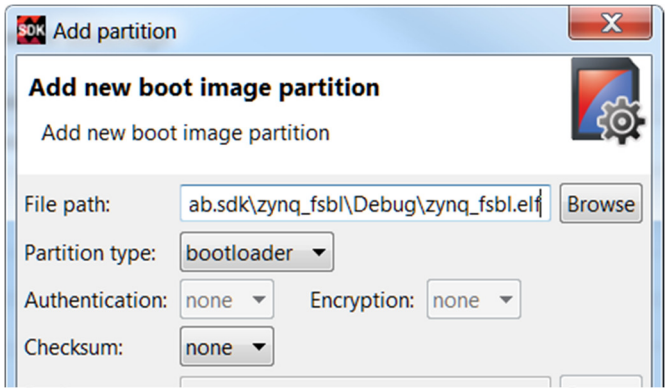


Figure 11. Adding FSBL partition

10-5-4. Click **OK**.

10-5-1. Click again on the **Add** button of the *Boot Image partitions,* click the Browse button in the Add Partition form, browse to **c:\xup\PR\labs\reconfig_peripheral_lab\Bitstreams** directory, select *blanking.bit* and click **Open**.

Note the partition type is datafile.

10-5-2. Click **OK**.

10-5-3. Click again on the **Add** button of the *Boot Image partitions,* click the Browse button in the Add Partition form, browse to **c:\xup\PR\labs\reconfig_peripheral_lab\reconfig_peripheral_<board>_lab\reconfig_peripheral_< board>_lab.sdk\TestApp\Debug** directory, select *TestApp.elf* and click **Open**.

Note the partition type is datafile.

10-5-4. Click **OK**.

10-5-5. Make sure that the output path is **c:\xup\PR\labs\reconfig_peripheral_lab** and the filename is *BOOT.bin,* and click **Create Image**.

10-5-6. Close the SDK program by selecting **File > Exit**.

Test the Design Step 11

11-1. Connect the board with micro-USB cable connected to the UART. Place the board in the SD boot mode. Copy the generated BOOT.bin and the partial bit files on the SD card and place the SD card in the board. Power On the board.

11-1-1. Make sure that a micro-usb cable is connected to the UART port.

11-1-2. Make sure that the board is set to boot in SD card boot mode.

11-1-3. Using the Windows Explorer, copy the **BOOT.bin** from the *c:/xup/PR/reconfig_peripheral_lab/* directory on to a SD Card.

11-1-4. Using the Windows Explorer, rename the three partial bin files in the *Bitstream* directory to *blank.bin, mult.bin,* and *add.bin* and then copy them on to the SD Card

11-1-5. Place the SD Card in the board and power ON the board.

11-2. Start a terminal emulator program such as TeraTerm or HyperTerminal. Select an appropriate COM port (you can find the correct COM number using the Control Panel). Set the COM port for 115200 baud rate communication.

11-2-1. Start a terminal emulator program such as TeraTerm or HyperTerminal.

11-2-2. Select the appropriate COM port (you can find the correct COM number using the Control Panel).

11-2-3. Set the *COM* port for **115200** baud rate communication.

11-2-4. Press **BTN7** to display a menu.

11-2-5. Follow the menu and test various reconfigurations.

Typing 1, 2, or 3 at the menu will let you partially reconfigure the multiplication, addition, blanking functionality respectively. Typing 4 will let you enter the operands and provide you the result.

Try various reconfigurations and enter operands after each reconfiguration to verify that the design indeed works.

When blanking bitstream is loaded, the result is 0.

11-2-6. Close Vivado by selecting **File > Exit**.

11-2-7. Power OFF the board.

Conclusion

This lab showed you steps involved in creating a processor system using Vivado IPI. Full bitstream as well as partial reconfiguration bitstreams were generated by going through the PR flow. You also learned how to generate the boot image as well as how to convert the partial bit files to bin format. You verified the functionality either using ZedBoard and/or Zybo.