Dynamic Partial Reconfiguration – PS+PL

OBJECTIVES

- Implement an embedded project (PS + PL) that can be dynamically reconfigured using the ZYBO Board. Vivado 2019.1.
- Learn the Partial Reconfiguration (PR) TCL-based non-project flow for PS+PL.
- Learn to develop software routines for Partial Reconfiguration via software using the PCAP.

VIVADO PARTIAL RECONFIGURATION - DOCUMENTATION

- UG909: Vivado Design Suite User Guide – Partial Reconfiguration.
- UG947: Vivado Design Suite Tutorial – Partial Reconfiguration. You can follow this for the Xilinx-provided `ug947-vivado-partial-reconfiguration-tutorial.zip` file (this is a Verilog design for the KC705 demonstration board)

TEST PROJECT – PIXEL PROCESSOR: 1 RECONFIGURABLE PARTITION (RP)

- **Pixel Processor:** The circuit, written in VHDL, processes NC NI-bit pixels in parallel and outputs NC NO-bit pixels. The Pixel Processor IP file structure of was modified (with respect to the one in Unit 4) so that it is suitable for this procedure.

- **RECONFIGURABLE PARTITION (RP)**: This is the dynamic region, i.e., the region that we can modify at run-time. This partition only has one parameter: `F(`function selector`: 1 to 5)`. We fix NC=4, NI=NO=8. We can create a set of 5 variants, also known as **Reconfigurable Modules (RM)**.
  `pixfull_rp.vhd`: Wrapper file where we can modify the parameters of the RP in order to create different variants (RMs).
- We will modify (at run-time) the Reconfigurable Partition by utilizing two variants
  - ✓ `F=1`: Function is Gamma Correction (0.5)
  - ✓ `F=2`: Function is Gamma Correction (2)
- **Two Configurations**: RM 1 (`pixfull_rp` has `F=1`), RM 2 (`pixfull_rp` has `F=2`).

PROCEDURE

- This procedure is adapted from the UG947: Vivado Design Suite Tutorial – Partial Reconfiguration. Changes were made to some .tcl files to allow us to use VHDL files and the Zynq-7000 device inside the ZYBO board.

- Extract the **`axipixfull_dprsys.zip`** file. It includes three folders and `.c` files:
  - ✓ `/axipixfull_dr`: Files for implementing the AXI4-Full Pixel Processor IP.
  - ✓ `/axipixfull_dr_static`: Files for implementing the AXI4-Full Pixel Processor IP (only the static portion).
  - ✓ `/my_dynpix`: File structure for implementing the self-reconfigurable system.

- Create an embedded system (`pixfull_drsys`) for the Pixel Processor. The AXI4-Full IP is called `mypixfull` (use the files in the folder `/axipixfull_dr`). The top VHDL file of the RP is `pixfull_rp.vhd`. Create an SDK project (use the `test_pixi_noSD.c` file available). Test to verify that this hardware works.

- Create an embedded system (`pixfull_drsys_static`) for the Pixel Processor but without the Reconfigurable Partition (RP). The AXI4-Full IP is called `mypixfull_static` (use the files in the folder `/axipixfull_dr_static`); this is identical to `mypixfull`, but without `pixfull_rp.vhd` (which becomes a black box). Synthesize this project: we cannot implement it due to the black box component.
  2019.1: Before creating the HDL wrapper, in Block Design → Sources, right-click on the top block diagram (golden tree) and select Generate Output Products. In Synthesis Options, the option Out of context per IP is selected by default; this will generate a netlist for each component, this is not what we want. So, you must select Global (then Apply) so that we only have one netlist for the static design. Then, create the HDL wrapper and synthesize.
  Once synthesized, this project is only useful in order to:
  - ✓ Extract the required XDC files. We know these are the required files as they are used by Vivado when synthesizing (or implementing `pixfull_drsys`). We can see this in the Vivado Implementation Log.
    `/pixfull_drsys_static/.srcs/sources_1/bd/design_1/ip/design_1_processing_system7_0_0/design_1_processing_system7_0_0.xdc`.
    `/pixfull_drsys_static/.srcs/sources_1/bd/design_1/ip/design_1_rst_ps7_0_100M_0/design_1_rst_ps7_0_100M_0.xdc`. (if using ZYBO, where the PL clock is 100 MHz by default; it is 50 MHz in ZYBO Z7-10)
  - ✓ Extract the synthesized file for the embedded system. This is a very important file.
    `/pixfull_drsys_static/.runs/synth_1/design_1_wrapper.dcp`
  - ✓ Find the 'cell' corresponding to `pixfull_rp`, named `ji` here; we might not find that cell in the project `pixfull_drsys`.

- Though we are working with the project `pixfull_drsys_static`, which has an IP called `mypixfull_static` with different driver functions (e.g.: `MYPIXFULL_STATIC_mWriteMemory`), we can safely use the `pixfull_drsys` project to create the SDK project and use its driver functions (e.g.: `MYPIXFULL_mWriteMemory`). This has been successfully tested.

- Go to the **/my_dynpix** folder. This the file structure. Add the .dcp and .xdc files as indicated.
  - ✓ design.tcl: Master script where the design sources, parameters, and structure are defined. We modified it so that the top portion (static here) is not synthesized. The supporting TCL scripts are located in /Tcl.
  - ✓ /Sources/hdl/top: Usually we place here the static region, i.e., the circuit that does not consider the Reconfigurable Partition (RP). Note that the RP is left as a black box. However, since we use the PS (and the .dcp for the static portion), we will leave this blank as we do not have VHDL files for the static portion.
  - ✓ /Sources/hdl/pixfull_1: pixfull_rp.vhd, pixfull_fifointf.vhd, LUT_group.vhd, LUT_NItoNO.vhd, LUT_NIto1.vhd, pack_xtras.vhd, LUT_values8to8.txt: These files constitute a Reconfigurable Module (where F is set to '1' in pixfull_rp.vhd), i.e., a variant of the RP.
  - ✓ /Sources/hdl/pixfull_2: pixfull_rp.vhd, pixfull_fifointf.vhd, LUT_group.vhd, LUT_NItoNO.vhd, LUT_NIto1.vhd, pack_xtras.vhd, LUT_values8to8.txt: These files constitute a Reconfigurable Module (where F is set to '2' in pixfull_rp.vhd), i.e., a variant of the RP.
  - ✓ /Sources/xdc: Constraint files specifying the I/O connections as well as the timing constraints for the clock input pin. These are extracted from the Vivado project pixfull_drsys_static (place the two .xdc files here).
  - ✓ /Synth/Static: Here we place the file design_1_wrapper.dcp which is the synthesized static portion (from the Vivado project pixfull_drsys_static). In the previous unit (LED pattern control), this folder was populated when the .tcl file was run and the top portion was synthesized. Now, we do not do that, just paste it from the Vivado project.

- From here, the procedure is similar to the Tutorial 6 example (LED pattern control). **Important differences**: i) we have to load the .xdc files (2 files), ii) instead of the cell name 'ji', we use the entire path that refers to 'ji': \....<path>\ji (when loading the static portion, you can get this from the critical warning or from Properties of the 'ji' blackbox).
- There might be many critical warnings when reading the xdc files in the process, however it seems to work fine!
- We are using the TCL-based flow (not the Vivado GUI-based flow). So, you have to execute the design.tcl script.

SYNTHESIS
- Open the Vivado TCL Shell (Program → Vivado 2019 TCL Shell). Navigate to the /my_dynpix directory.
- Run the design.tcl script by entering: source design.tcl -notrace. This will Synthesize the design and create output files in the /Synth folder.

ASSEMBLE THE DESIGN
- Open the Vivado IDE (start_gui). Go to the TCL console.
- Load the design: open_checkpoint Synth/Static/design_1_wrapper.dcp
  You can see the design structure in the Netlist pane, but a black box exists for the pixfull_rp partition. The instantiation name in the VHDL code is <path>/ji. * The path name might slightly change depending on the software version.
  For example: <path>/ji = design_1_i/mypixfull_static_0/U0/mypixfull_static_v1_0_S00_AXI_inst/gIP/th/ji
- Load the synthesized checkpoints for first Reconfigurable Module (RM) for each Reconfigurable Partition (RP). In our case, we will use the pixfull_1 as our first RM (tip: always use the one that takes the largest space). We only have one RP (instantiation name: <path>\ji).
  `read_checkpoint -cell <path>/ji Synth/pixfull_1/pixfull_rp_synth.dcp`
  Note that the pixfull_rp module has been filled in with logical resources.
- Define each RP as partially reconfigurable:
  `set_property HD.RECONFIGURABLE 1 [get_cells <path>/ji]`
- Save the assembled design state for this initial configuration (where RP is pixfull_rp with F=1, i.e., pixfull_1):
  `write_checkpoint ./Checkpoint/top_pixfull_1.dcp`

BUILD THE DESIGN FLOORPLAN
Here, you create a floorplan to define the regions that will be partially reconfigured.
- Select the <path>/ji instance in the Netlist pane. Right click and select Floorplanning → Draw Pblock and draw a rectangular box that fits the resources occupied by the largest RM in that particular RP (instance name <path>/ji). The Statistics Tab of the Pblock Properties pane provides a resource estimate and the available resources in the box just drawn. This is useful to optimize the resource count of your RPs.
- Run PR Design Rule Checks by selecting Report → Report DRC. Check for Partial Reconfiguration warnings. Refer to the Xilinx UG947 guide for explanation of warnings and what to do with them.
  Since our RP does not include flip flops, warning suggesting the use of the RESET_AFTER_RECONFIGURATION will not exist.
- Save these Pblock definitions and its associated properties on a .xdc file:
  `write_xdc ./Sources/xdc/fplan.xdc`

IMPLEMENT THE FIRST CONFIGURATION (RP: pixfull_rp F=1)
- Load the constraint files (to set device I/Os and top-level constraints) generated by the project pixfull_drsys_static:
  read_xdc Sources/xdc/design_1_processing_system7_0_0.xdc.
  read_xdc Sources/xdc/design_1_rst_ps7_0_100M_0.xdc. (This is for ZYBO. For ZYBO Z7-10 it will be ...50M_0.xdc)
- Optimize, place, and route the design. Notice the Partition Pins (interface points between static and dynamic regions)

```
opt_design
place_design
route_design
```

- Save the full design checkpoint and create report files:
```
write_checkpoint -force Implement/Config_pixfull_1/top_route_design.dcp
report_utilization -file Implement/Config_pixfull_1/top_utilization.rpt
report_timing_summary -file Implement/Config_pixfull_1/top_timing_summary.rpt
```

At this point, you can use the static portion of this configuration for all subsequent configurations (variants of the circuit with different RMs for each RP). We need to isolate the static design by removing the Reconfigurable Modules:

- Clear out Reconfigurable Module logic:
```
update_design -cell <path>/ji -black_box
```
- Lock down all placement and routing. This is an important step to guarantee consistency for different RMs for each RP.
```
lock_design -level routing
```
- Write out the remaining static-only checkpoint (this checkpoint will be used for any future configurations).
```
write_checkpoint -force Checkpoint/static_route_design.dcp
```

IMPLEMENT THE SECOND CONFIGURATION (RP: pixfull_rp F=2)

- With the locked static design open in memory, read in post-synthesis checkpoints for the other Reconfigurable Module:
```
read_checkpoint -cell <path>/ji Synth/pixfull_2/pixfull_rp_synth.dcp
```
- Optimize, place, and route the new RM.
```
opt_design
place_design
route_design
```
- Save the full design checkpoint and report files:
```
write_checkpoint -force Implement/Config_pixfull_2/top_route_design.dcp
report_utilization -file Implement/Config_pixfull_2/top_utilization.rpt
report_timing_summary -file Implement/Config_pixfull_2/top_timing_summary.rpt
```

- At this point, you have implemented the static design and all Reconfigurable Module variants. This process would be repeated for designs that have more than two Reconfigurable Modules per RP, or more RPs. Close the current design:
```
close_project
```

GENERATE BITSTREAMS

- Run the `pr_verify` command from the TCL console. This is to verify compatibility of all configurations.
```
pr_verify Implement/Config_pixfull_1/top_route_design.dcp
Implement/Config_pixfull_2/top_route_design.dcp
```

- Read the first configuration into memory:
```
open_checkpoint Implement/Config_pixfull_1/top_route_design.dcp
```
- Generate full and partial bitstreams for the first configuration
```
write_bitstream -file Bitstreams/Config_pixfull_1.bit
```

  Two bitstreams are created:
  `Config_pixfull_1.bit`: Power-up, full design bitstream
  `Config_pixfull_1_pblock_ji_partial.bit`: Partial bit file for the `pixfull_rp` module (first RM – F = 1)

  PCAP programming: We need bin files (.bit files without header). For this, we use (do not indicate .bin extension):
```
write_bitstream -bin_file -no_binary_bitfile Bitstreams/Config_pixfull_1
```

  The following files are created (they can also be used for JTAG programming):
  `Config_pixfull_1.bin`: Power-up, full design bitstream
  `Config_pixfull_1_pblock_ji_partial.bin`: Partial bit file for the `pixfull_rp` module (first RM – F = 1)
  We can also create all the .bin and .bit files if we use: `write_bitstream -bin_file Bitstreams/Config_pixfull_1`

```
close_project
```

  The partial .bin bitstream created by the previous command cannot be used to program the PL via PCAP (unless byte order is changed manually via software). A better way is to use the following command that uses the .bit file:
```
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit "up 0x0
Bitstreams/Config_pixfull_1_pblock_ji_partial.bit" -file
Bitstreams/Config_pixfull_1_pblock_ji_partialu
```
  The following bitstream (with bytes swapped) is created (useful for PCAP writing, not useful for JTAG programming):
  `Config_pixfull_1_pblock_ji_partialu.bin`: Partial bit file for the `pixfull_rp` module (first RM – F = 1)

- Read the second configuration into memory:
```
open_checkpoint Implement/Config_pixfull_2/top_route_design.dcp
```
- Generate full and partial bitstreams for the second configuration
```
write_bitstream -file Bitstreams/Config_pixfull_2.bit
```

Two bitstreams are created:
`Config_pixfull_2.bit`: Power-up, full design bitstream
`Config_pixfull_2_pblock_ji_partial.bit`: Partial bit file for the `pixfull_rp` module (second RM – F = 2)

PCAP programming: We need bin files (.bit files without header). For this, we use (do not indicate .bin extension):

```
write_bitstream -bin_file -no_binary_bitfile Bitstreams/Config_pixfull_2
```

The following files are created (they can also be used for JTAG programming):
`Config_pixfull_2.bin`: Power-up, full design bitstream
`Config_pixfull_2_pblock_ji_partial.bin`: Partial bit file for the `pixfull_rp` module (first RM – F = 2)
We can also create all the .bin and .bit files if we use: `write_bitstream -bin_file Bitstreams/Config_pixfull_2`

```
close_project
```

The partial .bin bitstream created by the previous command cannot be used to program the PL via PCAP (unless byte order is changed manually via software). A better way is to use the following command that uses the .bit file:

```
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit "up 0x0
Bitstreams/Config_pixfull_2_pblock_ji_partial.bit" -file
Bitstreams/Config_pixfull_2_pblock_ji_partialu
```

The following bitstream (with bytes swapped) is created (useful for PCAP writing, not useful for JTAG programming):
`Config_pixfull_2_pblock_ji_partialu.bin`: Partial bit file for the `pixfull_rp` module (first RM – F = 2)

- Generate a full bitstream with a blackbox for the RP, plus blanking bitstreams for the RMs, these can be used to erase an existing configuration to reduce power consumption:

```
open_checkpoint Checkpoint/static_route_design.dcp
update_design -cell <path>/ji -buffer_ports
place_design
route_design
write_checkpoint Checkpoint/Config_black_box.dcp
write_bitstream -file Bitstreams/Config_black_box.bit
```

The base configuration bitstream will have no logic in the RP. The `update_design` command inserts constant drivers (GND) for all outputs so that they don't float.

Two bitstreams are created:
`Config_black_box.bit`: Power-up, full design bitstream
`Config_black_box_pblock_ji_partial.bit`: Partial bit file for the `pixfull_rp` module (RM – black box)

PCAP programming: We need bin files (.bit files without header). For this, we use (do not indicate .bin extension):

```
write_bitstream -bin_file -no_binary_bitfile Bitstreams/Config_black_box
```

The following files are created (they can also be used for JTAG programming):
`Config_black_box.bin`: Power-up, full design bitstream
`Config_black_box_pblock_ji_partial.bin`: Partial bit file for the `pixfull_rp` module (first RM – F = 2)
We can also create all the .bin and .bit files if we use: `write_bitstream -bin_file Bitstreams/Config_pixfull_2`

```
close_project
```

The partial .bin bitstream created by the previous command cannot be used to program the PL via PCAP (unless byte order is changed manually via software). A better way is to use the following command that uses the .bit file:

```
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit "up 0x0
Bitstreams/Config_black_box_pblock_ji_partial.bit" -file
Bitstreams/Config_black_box_pblock_ji_partialu
```

The following bitstream is created (useful for PCAP writing, not useful for JTAG programming):
`Config_black_box_pblock_ji_partialu.bin`: Partial bit file for the `pixfull_rp` module (first RM – F = 2)

```
close_project
```

FPGA CONFIGURATION (FULL AND PARTIAL) – PIXEL PROCESSOR

- Open the SDK Project you created at the beginning for the embedded system `pixfull_drsys`.
- From the main Vivado IDE, select `Flow` → `Open Hardware Manager`.
- Then `Open a New hardware Target`.
- Select `Program Device` and pick the XC7Z010 Device. Navigate to the `/Bitstreams` folder to select `Config_pixfull_1.bit` (full bitstream with the First Configuration). Program the device. Use the SDK project to test the pixel processor. The expected results are:

Input	Output
0xDEADBEEF	0xEED2DDF7
0xBEBEDEAD	0xDDDDEED2
0xFADEBEAD	0xFDEEDDD2
0xCAFEBEDF	0xE3FFDDEF

PARTIAL RECONFIGURATION USING JTAG

- Select `Program Device`. Navigate to the Bitstreams Folder to select `Config_pixfull_2_pblock_ji_partial.bit` (partial bitstream for the Second Configuration). Program the Device. Run the SDK project again to test the pixel processor. The expected results are:

Input	Output
0xDEADBEEF	0xC1758DDF
0xBEBEDEAD	0x8D8DC175
0xFADEBEAD	0xF4C18D75
0xCAFEBEDF	0x9FFC8DC2

- Select `Program Device`. Navigate to the Bitstreams Folder to select `Config_black_box_pblock_ji_partial.bit` (partial bitstream for the Black Box Configuration). Program the Device. Do not run the software routine as it might freeze since there is no IP.
- You can also program the `Config_pixfull_1_pblock_ji_partial.bit` file (partial bitstream for the First Configuration) in order to restore the First Configuration.

You can repeat this experiment over and over with new partial bitstreams.

PARTIAL RECONFIGURATION USING PCAP

- Create a new SDK application. On Project Name, you can use: `pixtst_rp`.
- Copy the following files in the `/src` folder: `test_pixi_rp.c`, `xtra_func.h`. These files require the use of the '`xilffs`' library and the enabling of the string manipulation functions in the `xilffs` library. See Tutorial - Unit 5 for details.
- You might want to deactivate the (default) option Build All. This way, you compile (Build Project) only when needed.
- Right-click on `pixtst_rp` application. Click on Generate Linker Script. You MUST assign enough space in the heap/stack for the input, intermediate, and output data. As we dynamically allocate memory for partial bitstreams, enough space must be assigned in the heap; the compiler might not tell you that there is no enough memory. This is usually slightly more than the size of the partial bitstreams. Also, place the code/heap/stack section in DDR memory (the largest one).
- Copy the following files on the SD card:
  - ✓ `Config_pixfull_1_pblock_ji_partialu.bin`: Partial bit file for the `pixfull_rp` module (first RM – F = 2). Modify the filename to **`pix_1p.bin`** (because the SD card driver can only deal with files of the format 8.3).
  - ✓ `Config_pixfull_2_pblock_ji_partialu.bin`: Partial bit file for the `pixfull_rp` module (second RM – F = 1). Modify the filename to **`pix_2p.bin`** (because the SD card driver can only deal with files of the format 8.3).

- In Vivado, go to `Open a New hardware Target`.
  - ✓ Select `Program Device` and pick the XC7Z010 Device. Navigate to the `/Bitstreams` folder to select `Config_pixfull_2.bit` (full bitstream with the Second Configuration). Program the device.
- Use the SDK project `pixtst_rp` to test the pixel processor. The software routine tests the initial configuration before applying partial reconfiguration. Then, it loads the partial bitstream 1 and tests it. Finally, it loads partial bitstream 2 and tests it. The expected results from applying the two partial bitstreams are:

	Input	Output
RP: pixfull F=1	0xDEADBEEF	0xEED2DDF7
	0xBEBEDEAD	0xDDDDEED2
	0xFADEBEAD	0xFDEEDDD2
	0xCAFEBEDF	0xE3FFDDEF

	Input	Output
RP: pixfull F=2	0xDEADBEEF	0xC1758DDF
	0xBEBEDEAD	0x8D8DC175
	0xFADEBEAD	0xF4C18D75
	0xCAFEBEDF	0x9FFC8DC2

TEST PROJECT – DCT 2D: 1 RECONFIGURABLE PARTITION (RP)

- **DCT 2D:** The circuit, written in VHDL, processes $N \times N$ B-bit pixels and outputs $N \times N$ NO-bit pixels

- **RECONFIGURABLE PARTITION (RP)**: This RP has several parameters. We only vary one: `N(DCT Transform Size: 4,8,16)` and fix B=8, NO=16, NH=16. We can create a set of variants, also known as **Reconfigurable Modules (RM)**. `dctfull_rp.vhd`: Wrapper file where we can modify the parameters of the RP in order to create different variants (RMs).
- We will modify (at run-time) the Reconfigurable Partition by utilizing two variants:
  ✓ `N=4`: DCT 4x4 ✓ `N=8`: DCT 8x8
- **Two Configurations**: RM 1 (`dctfull_rp` has `N=4`), RM 2 (`dctfull_rp` has `N=8`).

PROCEDURE

- Extract the **`axidctfull_dr.zip`** file. It includes three folders and `.c` files:
  ✓ `/axidctfull_dr`: Files for implementing the AXI4-Full 2D DCT IP.
  ✓ `/axidctfull_dr_static`: Files for implementing the AXI4-Full 2D DCT IP (only the static portion).
  ✓ `/mydct_dyn`: File structure for implementing the self-reconfigurable system.

- Create an embedded system (`mydct_sys`) for the 2D DCT. The AXI4-Full IP is called `mydctfull` (use the files in the folder `/axidctfull_dr`, note that the IP file structure was modified so that it is suitable for this procedure). The top VHDL file of the RP is `dctfull_rp.vhd`. Create an SDK project (use the `dct_tst.c` file available). Test to verify that this hardware works.

- Create an embedded system (`mydct_sys_static`) for the 2D DCT minus the Reconfigurable Partition (RP). The AXI4-Full IP is called `mydctfull_static` (use the files in the folder `/axidctfull_dr_static`). This is like `mydctfull`, but without `dctfull_rp.vhd` (which becomes a <u>black box</u>). Synthesize this project (do not implement as the RP is a black box).
  <u>2019.1</u>: Before creating the HDL wrapper, in Block Design → Sources, right-click on the top block diagram (golden tree) and select Generate Output Products. In Synthesis Options, the option Out of context per IP is selected by default; this will generate a netlist for each component, this is not what we want. So, you must select Global so that we only have one netlist for the static design. Then, create the HDL wrapper and synthesize.
  Once synthesized, this project is only useful in order to:
  ✓ Extract the required XDC files. We know these are the required files as they are used by Vivado when synthesizing (or implementing `mydct_sys`). We can see this in the Vivado Implementation Log.
  `/mydct_sys_static/.srcs/sources_1/bd/design_1/ip/design_1_processing_system7_0_0/design_1_processing_system7_0_0.xdc`.
  `/mydct_sys_static/.srcs/sources_1/bd/design_1/ip/design_1_rst_processing_system7_0_50M_0/design_1_rst_processing_system7_0_50M_0.xdc`. (in ZYBO Z7-10 where PL clock = 50MHz by default; it is 100MHz in ZYBO)
  ✓ Extract the synthesized file for the embedded system. This is a very important file.
  `/mydct_sys_static/.runs/synth_1/design_1_wrapper.dcp`
  ✓ Find the 'cell' corresponding to `dctfull_rp`, named `ji` here; we might not find that cell in the project `mydct_sys`.

- Go to the **`/mydct_dyn.zip`** file. This is the file structure. Add the `.dcp` and `.xdc` to the folders as indicated.
  ✓ `design.tcl`: Master script where the design sources, parameters, and structure are defined. We modified it so that the top portion (static here) is not synthesized. The supporting TCL scripts are located in `/Tcl`.
  ✓ `/Sources/hdl/top`: Usually we place here the static region, i.e., the circuit that does not consider the Reconfigurable Partition (RP). Note that the RP is left as a <u>black box</u>. However, since we use the PS (and the .dcp for the static portion), we will leave this blank as we do not have VHDL files for the static portion
  ✓ `/Sources/hdl/dctfull_4one`: `dctfull_rp.vhd`, `dct_fifointf.vhd`, `dct_2d.vhd` (and all the derivative files including the `.txt` files): These files constitute a Reconfigurable Module (where `N` is set to '4' in `dctfull_rp.vhd`), i.e., a variant of the RP.
  ✓ `/Sources/hdl/dctfull_8one`: `dctfull_rp.vhd`, `dct_fifointf.vhd`, `dct_2d.vhd` (and all the derivative files including the `.txt` files): These files constitute a Reconfigurable Module (where `N` is set to '8' in `dctfull_rp.vhd`), i.e., a variant of the RP.
  ✓ `/Sources/xdc`: Constraint files specifying the I/O connections as well as the timing constraints for the clock input pin. These are extracted from the Vivado project `mydct_sys_static` (place the two .xdc files here).
  ✓ `/Synth/Static`: Place the file `design_1_wrapper.dcp` here which is the synthesized static portion (from the Vivado project `mydct_sys_static`). In the previous unit (LED pattern control), this folder was populated when the `.tcl` file was run and the top portion was synthesized. Now, we do not do that, just paste it from the Vivado project.

- From here, the procedure is similar to the Tutorial 6 example (LED pattern control). **Important differences**: i) we have to load the .xdc files (2 files), ii) instead of the cell name '`ji`', we use the entire path that refers to '`ji`': `\....<path>\ji` (when loading the static portion, you can get this from the critical warning or from Properties of the '`ji`' blackbox).
- <u>We saw many critical warnings when reading the xdc files in the process</u>, however it seems to work fine!
- We are using the TCL-based flow (not the Vivado GUI-based flow). So, you have to execute the `design.tcl` script.

SYNTHESIS

- Open the Vivado TCL Shell. Navigate to the `/mydct_dyn` directory.
- Run the `design.tcl` script: `source design.tcl -notrace ↵`. This will Synthesize the design and create output files in the `/Synth` folder.

ASSEMBLE THE DESIGN

- Open the Vivado IDE (`start_gui`). Go to the TCL console.
- Load the design: `open_checkpoint Synth/Static/design_1_wrapper.dcp`
  You can see the design structure in the Netlist pane, but a black box exists for the `dctfull_rp` partition. The instantiation name in the VHDL code is `<path>/ji`.
  For example: `<path>/ji` = `design_1_i/mydctfull_static_0/U0/mydctfull_v1_0_S00_AXI_inst/th/th/ji`
- Load the synthesized checkpoints for first Reconfigurable Module (RM) for each Reconfigurable Partition (RP). In our case, we will use the `dctfull_8one` as our first RM (tip: always use the one that takes the largest space). We only have one RP (instantiation name: `<path>/ji`).
  ```
  read_checkpoint -cell <path>/ji Synth/dctfull_8one/dctfull_rp_synth.dcp
  ```
  Note that the `dctfull_rp` module has been filled in with logical resources.
- Define each RP as partially reconfigurable:
  ```
  set_property HD.RECONFIGURABLE 1 [get_cells <path>/ji]
  ```
- Save the assembled design state for this initial configuration (where RP is `dctfull_rp` with `N=8`, i.e., `dctfull_8one`):
  ```
  write_checkpoint ./Checkpoint/top_dctfull_8one.dcp
  ```

BUILD THE DESIGN FLOORPLAN

Here, you create a floorplan to define the regions that will be partially reconfigured.

- Select the `<path>/ji` instance in the Netlist pane. Right click and select Floorplanning → Draw Pblock and draw a rectangular box that fits the resources occupied by the largest RM in that particular RP (instance name `<path>/ji`). The Statistics Tab of the `Pblock` Properties pane provides a resource estimate and the available resources in the box just drawn. This is useful to optimize the resource count of your RPs.
- Run PR Design Rule Checks by selecting Report → Report DRC. Check for Partial Reconfiguration warnings. Refer to the Xilinx UG947 guide for explanation of warnings and what to do with them.
  - ✓ RESET_AFTER_RECONFIGURATION: Recall that: i) during DPR: the RP outputs toggle, thus the FIFOs might have incorrect values, so we need to reset the FIFOs, and ii) after DPR: the FFs inside the RP are not cleared. Our RP includes flip flops and the warning suggest the use of the RESET_AFTER_RECONFIGURATION property. We prefer not to use it as it enforces more constraints on the RP shape and it does not reset the FIFOs (as they are not part of the RP). Instead, the AXI4-Full Interface allows for the reset of the RP and FIFOs via software.
  - ✓ SNAPPING_MODE: For small RPs, we can usually avoid the warnings regarding the left/right edges of the RP by changing the RP shape. However, for large RPs (as in this example), this is not possible. Instead, we must set the SNAPPING_MODE property as ON (as suggested by a warning). This will cause certain areas of the box to be avoided, thereby having less resources available (so re-check resource estimation).
- Save these Pblock definitions and its associated properties on a `.xdc` file:
  ```
  write_xdc ./Sources/xdc/fplan.xdc
  ```

IMPLEMENT THE FIRST CONFIGURATION (RP: dctfull_rp N=8)

- Load the constraint files (to set device I/Os and top-level constraints) generated by the project `mydct_sys_static`:
  ```
  read_xdc -cells design_1_i/processing_system7_0/inst Sources/xdc/design_1_processing_system7_0_0.xdc
  read_xdc Sources/xdc/design_1_rst_processing_system7_0_50M_0.xdc
  ```
- Optimize, place, and route the design. Notice the Partition Pins (interface points between static and dynamic regions)
  ```
  opt_design
  place_design
  route_design
  ```
- Save the full design checkpoint and create report files:
  ```
  write_checkpoint -force Implement/Config_dctfull_8one/top_route_design.dcp
  report_utilization -file Implement/Config_dctfull_8one/top_utilization.rpt
  report_timing_summary -file Implement/Config_dctfull_8one/top_timing_summary.rpt
  ```

At this point, you can use the static portion of this configuration for all subsequent configurations (variants of the circuit with different RMs for each RP). We need to isolate the static design by removing the Reconfigurable Modules:

- Clear out Reconfigurable Module logic:
  ```
  update_design -cell <path>/ji -black_box
  ```
- Lock down all placement and routing. This is an important step to guarantee consistency for different RMs for each RP.
  ```
  lock_design -level routing
  ```
- Write out the remaining static-only checkpoint (this checkpoint will be used for any future configurations).
  ```
  write_checkpoint -force Checkpoint/static_route_design.dcp
  ```

IMPLEMENT THE SECOND CONFIGURATION (RP: dctfull_rp N=4)

- With the locked static design open in memory, read in post-synthesis checkpoints for the other Reconfigurable Module:
  ```
  read_checkpoint -cell <path>/ji Synth/dctfull_4one/dctfull_rp_synth.dcp
  ```

- Optimize, place, and route the new RM.
```
opt_design
place_design
route_design
```
- Save the full design checkpoint and report files:
```
write_checkpoint -force Implement/Config_dctfull_4one/top_route_design.dcp
report_utilization -file Implement/Config_dctfull_4one/top_utilization.rpt
report_timing_summary -file Implement/Config_dctfull_4one/top_timing_summary.rpt
```

- At this point, you have implemented the static design and all Reconfigurable Module variants. This process would be repeated for designs that have more than two Reconfigurable Modules per RP, or more RPs. Close the current design:
```
close_project
```

GENERATE BITSTREAMS

- Run the `pr_verify` command from the TCL console. This is to verify compatibility of all configurations.
```
pr_verify Implement/Config_dctfull_8one/top_route_design.dcp
Implement/Config_dctfull_4one/top_route_design.dcp
```

- Read the configuration for 4x4 DCT into memory:
```
open_checkpoint Implement/Config_dctfull_4one/top_route_design.dcp
```
  - ✓ Generate full and partial bitstreams for this configuration
```
write_bitstream -file Bitstreams/Config_dctfull_4one.bit
```
    - Two bitstreams are created:
      `Config_dctfull_4one.bit`: Power-up, full design bitstream
      `Config_dctfull_4one_pblock_ji_partial.bit`: Partial bit file for the `dctfull_rp` module (RM with N = 4)
  - ✓ **PCAP programming:** Create the .bin file (.bit file without header) with bytes swapped (ready for PCAP programming):
```
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit "up 0x0
Bitstreams/Config_dctfull_4one_pblock_ji_partial.bit" -file
Bitstreams/Config_dctfull_4one_pblock_ji_partial
```
    - The following bitstream is created (useful for PCAP writing, not useful for JTAG programming):
      `Config_dctfull_4one_pblock_ji_partial.bin`: Partial bit file for the `dctfull_rp` module (RM with N = 4)
```
close_project
```

- Read the configuration for 8x8 DCT into memory:
```
open_checkpoint Implement/Config_dctfull_8one/top_route_design.dcp
```
  - ✓ Generate full and partial bitstreams for this configuration
```
write_bitstream -file Bitstreams/Config_dctfull_8one.bit
```
    - Two bitstreams are created:
      `Config_dctfull_8one.bit`: Power-up, full design bitstream
      `Config_dctfull_8one_pblock_ji_partial.bit`: Partial bit file for the `dctfull_rp` module (RM with N = 8)
  - ✓ **PCAP programming:** Create the .bin file (.bit file without header) with bytes swapped (ready for PCAP programming):
```
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit "up 0x0
Bitstreams/Config_dctfull_8one_pblock_ji_partial.bit" -file
Bitstreams/Config_dctfull_8one_pblock_ji_partial
```
    - The following bitstream is created (useful for PCAP writing, not useful for JTAG programming):
      `Config_dctfull_8one_pblock_ji_partial.bin`: Partial bit file for the `dctfull_rp` module (first RM with N = 8)
```
close_project
```

IMPORTANT NOTE:

- We can compile the entire design (from RTL to bitstreams) by running the `design_complete.tcl` script. However, you still need to run `design.tcl` to perform Synthesis and define the PR constraints. Then, the following steps need to be completed:
  - ✓ Change the name of the file `/Synth/Static/design_1_wrapper.dcp` to `top_synth.dcp`.
  - ✓ Create the `top.xdc` file: merge `design_1_processing_system7_0_0.xdc` and `fplan.xdc` (RP constraints). Once you Build the Design Floorplan, RP constraints are available (in this tutorial, they are saved in `/Sources/xdc/fplan.xdc`). If your design has external PL ports, you need to include the I/O and clocking constraints in `top.xdc`.
  - ✓ Verify the RP inst name in the script: `design_1_i/mydctfull_static_0/U0/mydctfull_v1_0_S00_AXI_inst/th/th/ji` This name can change due to software version, board, name of your peripheral, etc. Always double-check.
- To execute the script, go to the Vivado Tcl Shell and type: `source design_complete_tcl -notrace` ↵
  - ✓ Note: Synthesis will be executed again. This is not a problem (if you prefer, you can set `run.rmSynth` to 0 in the script).
- The script generates more intermediate checkpoints and reports. Note that the `.bin` bitstreams ready for PCAP programming are not generated. To do this, open Vivado, navigate to the `/mydct_dyn` folder and execute:

    -
      ```
      write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit "up 0x0
      Bitstreams/Config_dctfull_4one_pblock_ji_partial.bit" -file
      Bitstreams/Config_dctfull_4one_pblock_ji_partial
      ```

    -
      ```
      write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -loadbit "up 0x0
      Bitstreams/Config_dctfull_8one_pblock_ji_partial.bit" -file
      Bitstreams/Config_dctfull_8one_pblock_ji_partial
      ```

FPGA CONFIGURATION (FULL AND PARTIAL) – 2D DCT

- Open the SDK Project you created at the beginning for the embedded system `mydct_sys`.
- From the main Vivado IDE, select `Flow` → `Open Hardware Manager`.
- Then `Open a New hardware Target`.
- Select `Program Device` and pick the XC7Z010 Device. Navigate to the `/Bitstreams` folder to select `Config_dctfull_4one.bit` (full bitstream for 4x4 DCT Configuration). Program the device. Use the SDK project to test the 4x4 2D DCT (2 blocks are used). Make sure to use `dctsize = 4` (in `dct_test.c`). The expected results are:

Input (columns)	Output (rows)	
0xDEADBEEF	0x8000E92E	0x14C00D82
0xBEBEDEAD	0x18A6E418	0xDB3E1FB2
0xFADEBEAD	0x0A401E19	0x1D40236D
0xCAFEBEDF	0xF8382A32	0xDEC9FDE7

Input (columns)	Output (rows)	
0xCFC7C9C7	0x80000CF4	0xFF0003D5
0xCAC4C6C3	0x0471045F	0xFF89FF65
0xC6C3C7C3	0x010003CE	0x0000000B
0xBEBDC2BD	0x06D0FFE5	0x00310020

PARTIAL RECONFIGURATION USING JTAG

- Select `Program Device`. Navigate to the Bitstreams Folder to select `Config_dctfull_8one_pblock_ji_partial.bit` (partial bitstream for 8x8 DCT Configuration). Program the Device. Run the SDK project again to test the 8x8 2D DCT (1 block is used). Make sure to use `dctsize = 8` (in `dct_test.c`) The expected results are:

Input (columns)		Output (rows)			
0x7d807e79	0x7c7e7d77	0x7FFF0000	0x00000000	0x00000000	0x00000000
0x7c7a7a82	0x7d787c81	0x8000202C	0x29C602C9	0x8000CDDE	0xCEE9DA0A
0x7f7c7b81	0x7c797d7f	0xB5682122	0x276BC396	0x80000B24	0x5155311F
0x827e7b7f	0x7b7a7d7c	0x0C3F4BCC	0x7FFFD3F4	0x828AC81F	0x17CAF927
0x807e7b7e	0x7a7b7d7a	0xB4DDAEC2	0x7FFF3228	0x4B6035A5	0x5713FE62
0x7c7c7b7e	0x7a7b7c79	0xF7C2D535	0x7FFF37DE	0x56DB0688	0x2A25DD48
0x7b7d7c7e	0x7c797b7b	0x3D550DCB	0x7FFFE8A6	0x205904F2	0x5754143E
0x7f807d7d	0x7d78797d	0x789F58C8	0x5BD0B605	0xC7AECDC9	0x233B1094

- You can also program the `Config_dctfull_4one_pblock_ji_partial.bit` file (partial bitstream for the First Configuration) in order to restore the First Configuration.

You can repeat this experiment over and over with new partial bitstreams.

PARTIAL RECONFIGURATION USING PCAP

- Create a new SDK application. On Project Name, you can use: `dcttst_rp`.
- Copy the following files in the `/src` folder: `test_dct_rp.c`, `xtra_func.h`. These files require the use of the '`xilffs`' library and the enabling of the string manipulation functions in the `xilffs` library. See Tutorial – Unit 5 for details.
- Right-click on `dcttst_rp` application. Click on Generate Linker Script. You MUST assign enough space in the heap/stack for the input, intermediate, and output data. As we dynamically allocate memory for partial bitstreams, enough space must be assigned in the heap; the compiler might not tell you that there is no enough memory. This is usually slightly more than the size of the partial bitstreams. Also, place the code/heap/stack section in DDR memory (the largest one).

- Copy the following files on the SD card:
  - ✓ `Config_dctfull_4one_pblock_ji_partial.bin`: Partial bit file for the `dctfull_rp` module (first RM – N = 4). Modify the filename to **`dct_4o.bin`** (because the SD card driver can only deal with files of the format 8.3).
  - ✓ `Config_dctfull_8one_pblock_ji_partial.bin`: Partial bit file for the `dctfull_rp` module (second RM – N = 8). Modify the filename to **`dct_8o.bin`** (because the SD card driver can only deal with files of the format 8.3).

- In Vivado, go to `Open a New hardware Target`.
  - ✓ Select `Program Device` and pick the XC7Z010 Device. Navigate to the `/Bitstreams` folder to select `Config_dctfull_4one.bit` (full bitstream for the 4x4 DCT Configuration). Program the device.
- Use the SDK project `dcttst_rp` to test the 2D DCT. The software routine:
  - ✓ Tests the initial configuration (4x4) before applying partial reconfiguration.
  - ✓ Loads the partial bitstream for 8x8, asserts PR_reset (to reset the RP and the FIFOs), and tests the configuration.
  - ✓ Loads the partial bitstream for 4x4, asserts PR_reset (to rest the RP and the FIFOs), and tests the configuration.

- Verify that the results are as expected (see tables at the beginning of this section).