

Replicating the auto-switching reconfigurable architecture

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 four Reconfigurable Partitions having three Reconfigurable Modules (KL_UCB, UCB and Comparator). You will create multiple Configurations and run the Partial Reconfiguration implementation flow to generate full and partial bitstreams. You will use ZC706 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.

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 four peripheral (arms), having two unique function calculation capabilities (KL_UCB, UCB) and a reconfigurable Comparator design. The architecture is designed such that it switches automatically to a much resource-efficient UCB algorithm after learning the arm statistics using KL-UCB. The output (Quality-factor) of these machines goes into a Comparator IP which selects the machine with the largest Q-factor value. 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.

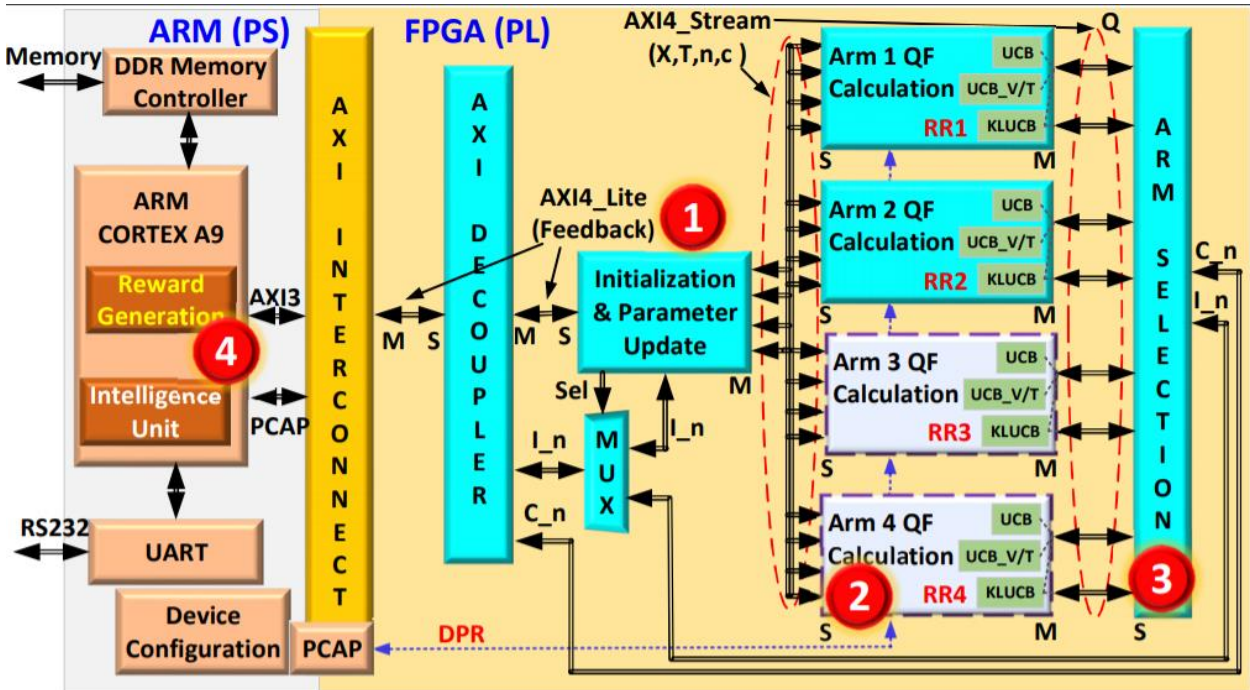


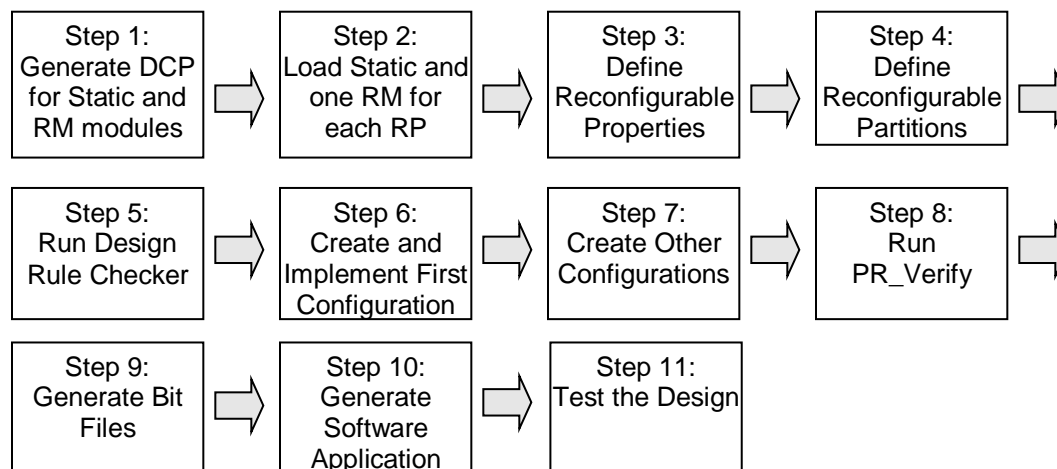
Figure 1. The design

The Sources directory provides the machine cores, source file for KL-UCB, UCB, the software application (C-code TestApp.c), and a place holder for the floorplan constraints (floorPlan.xdc). 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 2018.2 > Vivado 2018.2**

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

```
cd c:/Summer/Tutorial
```

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

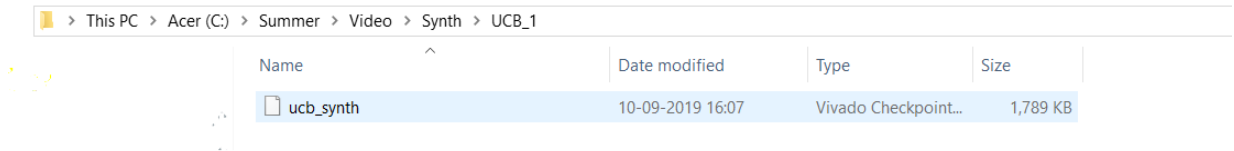
```
source blockDesign.tcl
```

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 machines IPs, PR decoupler and 2 Comparator IPs will also be instantiated. It will then create a top-level wrapper file called system_wrapper.v which instantiates the system.bd (the block design).

Figure 2. The system block design

- 1-1-4. Select the **Address Editor** tab. Expand **Data**. Expand **Unmapped Slaves**, if any, and right-click and select **Assign Address**.
- 1-1-5. Select **Tools > Validate Design**.
- 1-1-6. Select **File > Save Block Design**.
- 1-2. **Synthesize the design to generate the dcp for the static logic of the design.**
 - 1-2-1. 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**.
 - 1-2-2. Using the windows explorer, copy the **system_wrapper.dcp** file from *tutorial\tutorial.runs\synth_1* into the *Synth\Static* directory under the current lab directory.
 - 1-2-3. Copy design checkpoints for the *auto_pc*, *machine_arms*, *comparator_0*, *comparator_1*, *pr_decoupler_0*, *xbar_0*, *rst_ps7_0_20M*, and *processing_system7_0* instances to *Synth\Static* to sit alongside *system_wrapper.dcp*
 - 1-2-4. Close the project by typing the `close_project` command in the Tcl console or selecting **File > Close Project**.
- 1-3. **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.**
 - 1-3-1. The HDL files for all the algorithms corresponding to each have been provided. Synthesize each each RM by creating a separate Vivado project. For the algorithm, synthesize the *KL_UCB* IP as *kl_ucb*, and synthesize the *UCB* IP as *Q_function*. You can also see these IPs instantiated in the given HDL files.
 - 1-3-2. Synthesize each of the RMs and write the design checkpoint (dcp) in the respective destination folder under the *Synth* directory. After each RM's dcp is generated, close the design.
 - 1-3-3. At this point the directory content will look like shown below. Here, *UCB_1* denotes that this dcp corresponds to the *UCB* algorithm for machine 1. You just need to synthesize each algorithm separately and add the corresponding .dcp files to the respective folders.



Name	Date modified	Type	Size
ucb_synth	10-09-2019 16:07	Vivado Checkpoint...	1,789 KB

Figure 7. Synth directory hierarchy and content

- 1-3-4.** Also, synthesize the dcp for both the comparator modules. One of the modules will be switched to blanking configuration once the architecture switches to the resource-efficient UCB algorithm.

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:/Summer/Tutorial
```

- 2-1-2.** Execute the following Tcl script to load the static design checkpoint.

```
source load_design_checkpoints.tcl
```

The script will do the following:

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

```
open_checkpoint Synth/Static/system_wrapper.dcp
```
- Load the IP checkpoint for the Processing System by using the **read_checkpoint** command.

```
read_checkpoint -cell system_i/processing_system7_0  
Synth/Static/system_processing_system7_0_0.dcp
```
- Load the IP checkpoint for the machines by using the **read_checkpoint** command.

```
read_checkpoint -cell system_i/machine  
Synth/Static/system_machine_arms_0_0.dcp
```
- Load the IP checkpoint for the decoupler by using the **read_checkpoint** command.

```
read_checkpoint -cell system_i/pr_decoupler_0  
Synth/Static/system_pr_decoupler_0_0.dcp
```
- Load the IP checkpoint for the Processing Reset by using the **read_checkpoint** command.

```
read_checkpoint -cell system_i/rst_ps7_0_100M  
Synth/Static/system_rst_ps7_0_20M_0.dcp
```

- Load the IP checkpoint for the auto pc by using the **read_checkpoint** command.

```
read_checkpoint -cell
system_i/ps7_0_axi_periph/s00_couplers/auto_pc
Synth/Static/system_auto_pc_0.dcp
```

- Load the IP checkpoint for the Comparator 1 by using the **read_checkpoint** command.

```
read_checkpoint -cell system_i/ComparePR
Synth/Static/system_comparatorPR_0_0.dcp
```

- Load the IP checkpoint for the Comparator 2 by using the **read_checkpoint** command.

```
read_checkpoint -cell system_i/Compare
Synth/Static/system_comparator_0_0.dcp
```

- Load the IP checkpoint for the system bus xbar by using the **read_checkpoint** command.

```
read_checkpoint -cell system_i/ps7_0_axi_periph/xbar
Synth/Static/system_xbar_0.dcp
```

2-1-3. Load one RM (KL-UCB) for the RP by using the **read_checkpoint** command.

```
read_checkpoint -cell
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u1
Synth/KL_1/ucb_synth.dcp
```

```
read_checkpoint -cell
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u2 Synth/
KL_2/ucb_synth.dcp
```

```
read_checkpoint -cell
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u3 Synth/
KL_3/ucb_synth.dcp
```

```
read_checkpoint -cell
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u4 Synth/
KL_4/ucb_synth.dcp
```

```
read_checkpoint -cell
system_i/ComparePR/inst/comparatorPR_v1_0_S00_AXI_inst/u1 Synth/
Compare/compare.dcp
```

You can now see the design structure in the Netlist pane with an RM for the *u1*, *u2*, *u3* and *u4* module loaded.

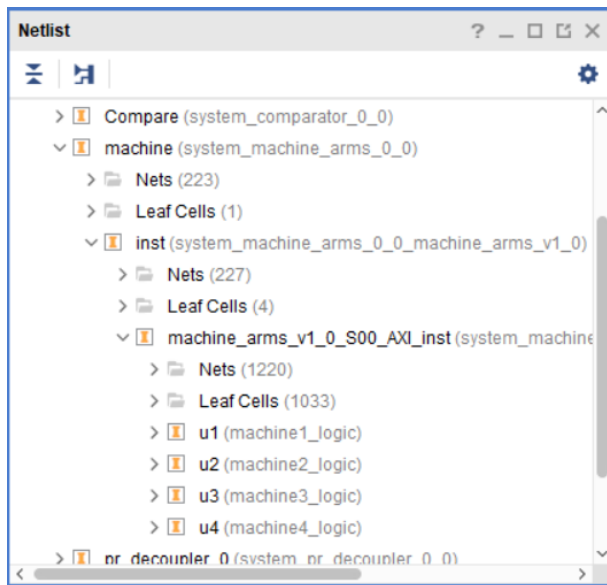


Figure 9. Static design with RM loaded

- 2-1-4.** Select the *u1*, *u2*, *u3*, *u4* and *u1*(Comparator) instances one-by-one 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.

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/machine/inst/machine_arms_v1_0_S00_AXI_inst/u1]

set_property HD.RECONFIGURABLE 1 [get_cells
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u2]

set_property HD.RECONFIGURABLE 1 [get_cells
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u3]

set_property HD.RECONFIGURABLE 1 [get_cells
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u4]

set_property HD.RECONFIGURABLE 1 [get_cells
system_i/ComparePR/inst/comparatorPR_v1_0_S00_AXI_inst/u1]
```

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

- 3-1-2.** Select the *u1*, *u2*, *u3*, *u4* and *u1*(Comparator) instance and notice that the *DONT_TOUCH* checkbox is selected in the **Cell Properties** window.

Define the Reconfigurable Partition Region

3-2. 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.

3-2-1. You execute the following command to define the region for each RP, perform the DRC.

```
read_xdc floorplan.xdc
```

Create and Implement First Configuration

4-1. Create and implement the first Configuration.

4-1-1. Execute the following command to implement the first configuration, the KL-UCB variant.

```
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
```

- Save the full design checkpoint.

```
write_checkpoint -force Implement/KL/top_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.

4-2. 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.

4-2-1. 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/machine/inst/machine_arms_v1_0_S00_AXI_inst/u1 -
```

```
black_box
```

```
update_design -cell
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u2 -
black_box

update_design -cell
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u3 -
black_box

update_design -cell
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u4 -
black_box

update_design -cell
system_i/ComparePR/inst/comparatorPR_v1_0_S00_AXI_inst/u1 -
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.

- 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

5-1. Read next set of RM dcp, create and implement the second configuration.

- 5-1-1.** Execute the following command to create and implement the second configuration, the UCB_T variant.

```
source create_second_configuration.tcl
```

The script will do the following tasks:

- First, it will open the blanking configuration using the tcl command:

```
open_checkpoint Checkpoint/static_route_design
```

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


```
read_checkpoint -cell
system_i/machine1/inst/machine1_blank_v1_0_S00_AXI_inst/u1
Synth/UCB_1/ucb_synth.dcp
```

```
read_checkpoint -cell
system_i/machine2/inst/machine2_blank_v1_0_S00_AXI_inst/u1
Synth/UCB_2/ucb_synth.dcp
```

```
read_checkpoint -cell
system_i/machine3/inst/machine3_blank_v1_0_S00_AXI_inst/u1
Synth/UCB_3/ucb_synth.dcp
```

```
read_checkpoint -cell
system_i/machine4/inst/machine4_blank_v1_0_S00_AXI_inst/u1
Synth/UCB_4/ucb_synth.dcp
```

```
read_checkpoint -cell
system_i/ComparePR/inst/comparatorPR_v1_0_S00_AXI_inst/u1
Synth/Compare/compare.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/UCB/top_route_design.dcp
```

- Close the project

```
close_project
```

5-2. Create the blanking configuration.

5-2-1. 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/machine/inst/machine_arms_v1_0_S00_AXI_inst/u1
```

```
update_design -buffer_ports -cell
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u2
```

```
update_design -buffer_ports -cell
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u3
```

```
update_design -buffer_ports -cell
system_i/machine/inst/machine_arms_v1_0_S00_AXI_inst/u4
```

```
update_design -buffer_ports -cell
system_i/ComparePR/inst/comparatorPR_v1_0_S00_AXI_inst/u1
```

- 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 `BLANK` directory .

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

- Close the project

```
Close_project
```

Run PR_Verify

- 6-1. 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**

- 6-1-1.** 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/KL/top_route_design.dcp -additional
{Implement/BLANK/top_route_design.dcp
Implement/UCB/top_route_design.dcp}
```

You should see the message indicating the KL configuration is compatible with BLANK, and the KL configuration is compatible with UCB. Execute the following command to close the project.

```
close_project
```

Generate Bit Files

7-1. After all the Configurations have been validated by PR_Verify, full and partial bit files must be generated for the entire project

7-1-1. Generate the full configurations and partial bitstreams by executing the following tcl script.

```
source generate_bitstreams.tcl
```

7-1-2. The script will do the following tasks:

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

```
open_checkpoint Implement/UCB/top_route_design.dcp
```

- Generate the full and partial bitstreams for this design.

```
write_bitstream -file Bitstreams/UCB.bit
```

```
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -
loadbit "up 0 Bitstreams/UCB_pblock_machine1_partial.bit"
Bitstreams/UCB_1.bin
```

```
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -
loadbit "up 0 Bitstreams/UCB_pblock_machine2_partial.bit"
Bitstreams/UCB_2.bin
```

```
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -
loadbit "up 0 Bitstreams/UCB_pblock_machine3_partial.bit"
Bitstreams/UCB_3.bin
```

```
write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -
loadbit "up 0 Bitstreams/UCB_pblock_machine4_partial.bit"
Bitstreams/UCB_4.bin
```

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

```
close_project
```

Notice the five bitstreams will be created.

7-1-3. Generate full and partial bitstreams for the second configuration.

- Open the checkpoint for KL-UCB using

```

open_checkpoint Implement/KL/top_route_design.dcp

write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -
loadbit "up 0 Bitstreams/ KL_pblock_machine1_partial.bit"
Bitstreams/UCBV_1.bin

write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -
loadbit "up 0 Bitstreams/ KL_pblock_machine2_partial.bit"
Bitstreams/UCBV_2.bin

write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -
loadbit "up 0 Bitstreams/ KL_pblock_machine3_partial.bit"
Bitstreams/UCBV_3.bin

write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -
loadbit "up 0 Bitstreams/ KL_pblock_machine4_partial.bit"
Bitstreams/UCBV_4.bin

close_project

```

The five 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.

```

write_bitstream -file Bitstreams/BLANK.bit

write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -
loadbit "up 0 Bitstreams/BLANK_pblock_machine1_partial.bit"
Bitstreams/BLANK_1.bin

write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -
loadbit "up 0 Bitstreams/BLANK_pblock_machine2_partial.bit"
Bitstreams/BLANK_2.bin

write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -
loadbit "up 0 Bitstreams/BLANK_pblock_machine3_partial.bit"
Bitstreams/BLANK_3.bin

write_cfgmem -format BIN -interface SMAPx32 -disablebitswap -
loadbit "up 0 Bitstreams/BLANK_pblock_machine4_partial.bit"
Bitstreams/BLANK_4.bin

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

close_project

```

- Generate full and partial bitstreams for the third configuration.

About the C code

- This function is used for moving the partial binaries from the SD Card to the DDR memory.

```
int SD_TransferPartial(char *FileName, u32 DestinationAddress, u32 ByteLength)
{
    FIL fil;
    FRESULT rc;
    UINT br;

    rc = f_open(&fil, FileName, FA_READ);
    if (rc) {
        xil_printf(" ERROR : f_open returned %d\r\n", rc);
        return XST_FAILURE;
    }

    rc = f_lseek(&fil, 0);
    if (rc) {
        xil_printf(" ERROR : f_lseek returned %d\r\n", rc);
        return XST_FAILURE;
    }

    rc = f_read(&fil, (void*) DestinationAddress, ByteLength, &br);
    if (rc) {
        xil_printf(" ERROR : f_read returned %d\r\n", rc);
        return XST_FAILURE;
    }

    rc = f_close(&fil);
    if (rc) {
        xil_printf(" ERROR : f_close returned %d\r\n", rc);
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}
```

- This function is used to initialize the SD controller.

```
int SD_Init()
{
    FRESULT rc;

    rc = f_mount(&fatfs, "", 0);
    if (rc) {
        xil_printf(" ERROR : f_mount returned %d\r\n", rc);
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}
```

- Below, we have defined the function to write the data to our arms.

```
#define mWriteReg(BaseAddress, RegOffset, Data) \
    Xil_Out32((BaseAddress) + (RegOffset), (u32)(Data))
```

- Below, we have defined the function to read the data from the comparator.

```
#define mReadReg(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (RegOffset))
```

- Below is the pseudocode for the Multi-Armed Bandit algorithm on FPGA.

```
algo()
{
    for n times {
        • Generate a random data value.
        • Read the output from the Comparator
        • If Comparator says Machine i is better-
            1. CHECK THE REWARD.
            2. INCREMENT X(i) if REWARD=1, else don't.
            3. INCREMENT T(i) as the Machine i was selected.
            4. WRITE THE CORRESPONDINT DATA TO THE FPGA CONVEYING THE
               MACHINE SELECTED AND ITS REWARD.
    }
}
```

Generate the Software Application

Step 10

8-1. Open the PS design that was created in Step 1. Export the hardware design and launch SDK.

8-1-1. Click on the **Open Project** link, browse to *c:/Summer/Tutorial/tutorial*, select the *tutorial.xpr* and click **OK** to open the design created in Step 1.

8-1-2. Select **File > Export > Export Hardware...**

8-1-3. In the *Export Hardware* form, do not check the *Include bitstream* checkbox and click **OK**.

8-1-4. Select **File > Launch SDK**

8-1-5. Click **OK** to launch SDK.

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

8-2. Create a Board Support Package enabling generic FAT file system library.

8-2-1. In **SDK**, select **File > New > Board Support Package**.

8-2-2. Click **Finish** with the default settings (with standalone operating system).

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

8-2-3. Select **xilffs** as the FAT file support is necessary to read the partial bit files.

Name	Version	Description	
<input type="checkbox"/> libmetal	1.0	Libmetal Library	
<input type="checkbox"/> lwip141	1.6	lwIP TCP/IP Stack library: lwIP v1.4.1	
<input type="checkbox"/> openamp	1.1	OpenAmp Library	
<input checked="" type="checkbox"/> xilffs	3.4	Generic Fat File System Library	
<input type="checkbox"/> xilflash	4.2	Xilinx Flash library for Intel/AMD CFI com...	
<input type="checkbox"/> xilisf	5.7	Xilinx In-system and Serial Flash Library	
<input type="checkbox"/> xilmfs	2.1	Xilinx Memory File System	
<input type="checkbox"/> xilpm	2.0	Power Management API Library for Zynq...	
<input type="checkbox"/> xilrsa	1.2	Xilinx RSA Library	
<input type="checkbox"/> xilskey	6.0	Xilinx Secure Key Library	

Figure 13. Selecting the xilffs library support

8-2-4. Click **OK** to accept the settings and create the BSP.

8-3. Create an application.

8-3-1. Select **File > New > Application Project**.

8-3-2. Enter **TestApp** as the *Project Name*, and for *Board Support Package*, choose **Use Existing** (*standalone_bsp_0* should be the only option).

8-3-3. Click **Next**, and select *Empty Application* and click **Finish**.

8-3-4. Expand the **TestApp** entry in the project view, right-click the *src* folder, and select **Import**.

8-3-5. Expand **General** category and double-click on **File System**.

8-3-6. Browse to *c:\Summer\Tutorial\Sources* and click **OK**.

8-3-7. Select **TestApp.c** and click **Finish** to add the file to the project.

8-3-8. Right-click on **TestApp** and select **C/C++ Building Settings**.

8-4. Create a zynq_fsbl application.

8-4-1. Select **File > New > Application Project**.

8-4-2. Enter **zynq_fsbl** as the *Project Name*, and for *Board Support Package*, choose **Create New**.

8-4-3. Click **Next**, select *Zynq FSBL*, and click **Finish**.

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

8-5. Create a Zynq boot image.

8-5-1. Select **Xilinx Tools > Create Boot Image**.

8-5-2. Click the Browse button of the Output BIF file path field, browse to **c:\Summer\Tutorial**, and then click **Save** with the *output* as the default filename.

8-5-3. Click on the **Add** button of the *Boot image partitions*, click the Browse button in the Add Partition form, browse to **c:\Summer\Tutorial\tutorial\tutorial.sdk\zynq_fsbl\Debug** directory, select *zynq_fsbl.elf* and click **Open**.

8-5-4. Click **OK**. Click again on the **Add** button of the *Boot Image partitions*, click the Browse button in the Add Partition form, browse to **c:\Summer\Tutorial\Bitstreams** directory, select *BLANK.bit* and click **Open**.

8-5-5. Click **OK**.

8-5-6. Click again on the **Add** button of the *Boot Image partitions*, click the Browse button in the Add Partition form, browse to **c:\Summer\Tutorial\tutorial\tutorial.sdk\TestApp\Debug** directory, select *TestApp.elf* and click **Open**.

8-5-7. Click **OK**.

8-5-8. Make sure that the output path is **c:\Summer\Tutorial** and the filename is *BOOT.bin*, and click **Create Image**.

8-5-9. Close the SDK program by selecting **File > Exit**.

Test the Design

Step 11

9-1. 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.

- 9-1-1. Make sure that a micro-usb cable is connected to the UART port.
- 9-1-2. Make sure that the board is set to boot in SD card boot mode.
- 9-1-3. Using the Windows Explorer, copy the **BOOT.bin** from the **c:\Summer\Tutorial** directory on to a SD Card. Also copy all the partial bitstreams into the SD card.
- 9-1-4. Place the SD Card in the board and power ON the board.
- 9-2. **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.**
- 9-2-1. Start a terminal emulator program such as TeraTerm or HyperTerminal.
- 9-2-2. Select the appropriate COM port (you can find the correct COM number using the Control Panel).
- 9-2-3. Set the *COM* port for **115200** baud rate communication.
- 9-2-4. Reset the board and the menu will appear. Use it as you want.

```
No. of enabled machines: 3
1. Enable a new arm.
2. Disable an arm.
3. An experiment for 10000 slots.
Enter the choice: 3

Starting new experiment with mean arm probabilities of 0.51, 0.52, 0.53
Architecture is switching from KL-UCB to UCB at 1526th slot.
Final result = Arms 1, 2, 3 were selected 2320, 4017, 3663 times.
*****
1. Enable a new arm.
2. Disable an arm.
3. An experiment for 10000 slots.
Enter the choice: 1

Enabling arm.
No. of enabled arms: 4
*****
1. Enable a new arm.
2. Disable an arm.
3. An experiment for 10000 slots.
Enter the choice: 3

Starting new experiment with mean arm probabilities of 0.51, 0.52, 0.53, 0.54
Architecture is switching from KL-UCB to UCB at 1809th slot.
Final result = Arms 1, 2, 3, 4 were selected 1523, 2810, 2421, 3246 times.
*****
1. Enable a new arm.
2. Disable an arm.
3. An experiment for 10000 slots.
Enter the choice: 2

Disabling arm.
No. of enabled arms: 3
*****
1. Enable a new arm.
2. Disable an arm.
3. An experiment for 10000 slots.
```

The snapshot of the terminal with example user experiments

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 using ZC706.