ZyCAP

A high-speed Reconfiguration Controller for Xilinx Zynq SoC

User Guide

Shreejith Shanker Vipin Kizheppatt

7 December 2015

Version 1.0

Contents

List of Abbreviations

AXI	Advanced eXtensible Interface
ICAP	Internal Configuration Access Port
DMA	Direct Memory Access
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GP port	General Purpose Port
HP port	High Performance Port
FSM	Finite State Machine
PL	Programmable Logic
PR	Partial Reconfiguration
PRR	Partially Reconfigurable Region
PS	Processing System
RM	Reconfigurable Module

One

Introduction

ZyCAP is a custom ICAP controller for high speed and efficient *partial reconfiguration* (PR) on Xilinx's Zynq SoCs. It has a hardware IP core, which can be used within the Xilinx's XPS environment or Vivado tool-flow to integrate with the Zynq processing system (PS). The ZyCAP driver takes care of low level reconfiguration operations, bitstream caching and bitstream memory management. The user is provided with API functions that can be used for software-driven reconfiguration from the PS. Currently, the design has been verified on the Digilent Zed Board and the Zynq ZC-702 platforms, with the Standalone OS and the Xillinux kernel. These working examples can be directly downloaded from the repository hosted at https://github.com/archntu/zycap.git.

1.1 Integration Flow: Hardware

ZyCAP is available as a packaged IP core: as a *pcore* for XPS environment and as an *IP package* for Vivado environment. User designs can instantiate the IP core into their hardware design like any other standard IP cores, after importing the ZyCAP IP repository into their project.

A block-level interface of the ZyCAP IP is as shown in Fig. 1.1. ZyCAP uses two AXI interfaces to communicate with the Zynq PS. A low-speed AXI4 lite interface (S_AXI_LITE) is used to configure and initiate a reconfiguration operation from the PS. A second high-speed AXI4 streaming interface (M_AXI_MM2S) is used to read the partial bitstream from the PS memory (SD Card or DDR) and configure the PL. The interrupt pin (mm2s_introut) of ZyCAP is wired to the PL Interrupt of the Zynq PS for synchronising the software flow to the hardware reconfiguration.

The example design (in folder *zycap/exdesign/ise* and *zycap/exdesign/vivado*) provides a framework for integrating ZyCAP in a user design. Instantiating ZyCAP into XPS or Vivado design flows will automatically

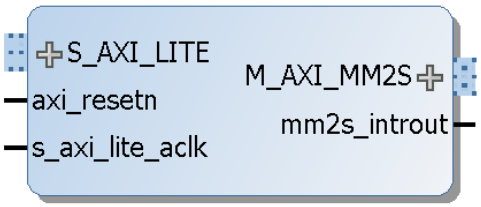


Figure 1.1: Hardware Interfaces of ZyCAP

connect the interfaces using the connection automation scheme built into the respective tools. ZyCAP clock has to be configured at 100 MHz, which is the recommended configuration speed for ICAP interface. The example design instantiates a register interface with an up/down counter (depending on the configuration) which can be replaced by the user design.

1.2 Integration Flow: Software

The software driver allows user applications to utilise hardware (partial) reconfigurability at high-speeds without being aware of the low-level operations associated with PR. The driver handles high-level tasks such as management of bitstreams and DRAM memory as well as low-level operations like configuration of the hardware reconfiguration manager (ICAP and PCAP) as well as the bitstream movement. Simple APIs are provided to the user application to trigger reconfiguration and to synchronise software tasks with the hardware adaptation.

The list of APIs and their functionality are shown in table 1.1. The *Configure_* API takes a *mode* name as its input, loads the corresponding configuration bitstream and triggers the operation. For advanced cases where the software flow need not wait for the completion of reconfiguration tasks, the synchronisation may be disabled. The operations can be synchronised later using the *Sync_* API in such cases. An explicit *Prefetch_* API allows the configuration bitstreams to be pre-loaded into the DRAM for consistent high-speed reconfiguration. In case of Linux OS, the *Configure_* API will always execute the *Sync_* API for stable reconfiguration.

The drivers for the standalone operating system from Xilinx is associated with the IP core itself (zycap/core/ise/driver for ISE/XPS flow

Table 1.1: ZyCAP APIs and their operation in Standalone (SA) and Linux environments

API	Arguments	Description	OS
Init_Zycap()	*intr_cntrlr	Initialise the Zycap controller and allocate memory for PR bitstreams. A pointer to the interrupt controller has to be passed as an argument to the API.	SA
	void	Initialise the Zycap controller and allocate memory for PR bitstreams. Takes no arguements	Linux
Config_PR_Bitstream()	bitstream name, intr_sync	Reprogram using a bitstream. If the intr_sync option is set, the API returns immediately after initialising the reconfiguration. The Sync_Zycap() API should be used in this case before accessing the reconfigured peripheral.	SA
	bitstream name	Reprogram using a bitstream. The interface synchronises internally and hence does not use intr_sync argument.	Linux
Prefetch_PR_Bitstream()	bitstream name	Prefetch the PR bitstream from SD card to DRAM	SA & Linux
Sync_Zycap()	(void)	Synchronise ZyCAP reconfiguration interrupt	SA only

and zycap/core/vivado/driver for vivado flow). The example application (core/ise/app/pr_app.c or core/vivado/app/pr_app.c) shows the detailed use of ZyCAP core and the different APIs supported. Note that the *'zycap.h'* header file has to be included in user applications to access the APIs.

For Linux operating system, a pre-built image that includes ZyCAP driver is provided in (zycap/linux/sdimg/), which can be copied to the SD Card (see Chapter 4 for detailed instructions). The hardware build for linux usage is based on the Vivado flow, though the ISE/XPS build could also be used (see Chapter 4 → "Porting the example to other systems"

for instructions and notes).

For better understanding about ZyCAP system and our case studies in real-time embedded systems and software defined radio applications, users may refer to our papers in IEEE Embedded Systems Letters [1], Design Automation Conference [2] and Cognitive Radio Oriented Wireless Networks Conference [3].

For more information about PR and the tool flows, see the user guides from Xilinx Inc. ([4, 5])

1.3 Folder Organisation

The folder structure for the git repository is shown in Fig. 1.2. The *doc* folder hosts this user guide, while the *zycap* folder hosts the ip core (directory *core*) example designs (directory *exdesign*) and the linux image (directory *linux*) that uses ZyCAP for reconfiguration management. The *core* folder holds the ZyCAP ipcore for *ise* and *vivado* tool flows along with the corresponding user *app* and software *driver*. Similarly, *exdesign* folder hosts the projects for building the example design using *ise* and *vivado* flows. More details about the contents of these folders are discussed in the subsequent sections.

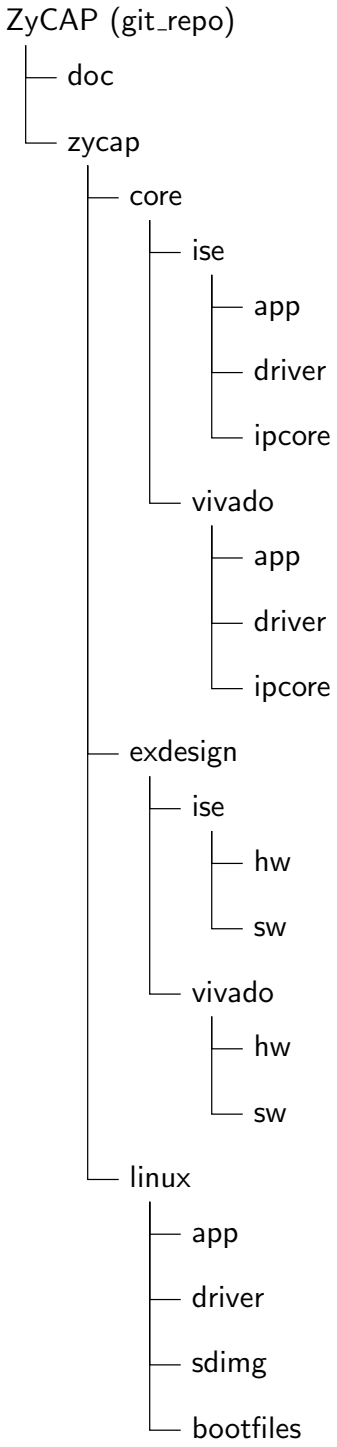


Figure 1.2: Folder structure of the ZyCAP repository

Two

ZyCAP in XPS Environment

XPS is part of the older Embedded Design Kit (EDK) tool-flow from Xilinx Inc. XPS allows designers to build embedded computing logic using simple drag-and-drop operation. XPS can be used for Zynq and older FPGA devices from Xilinx. XPS can also act as a stand-alone implementation tool for designs that can be completely described within this environment (i.e., Soft-processors + pre-defined IPs).

For partial reconfiguration, we make use of another tool from Xilinx called PlanAhead. Though similar in operation to Xilinx ISE, PlanAhead allows more extensive control and advanced capabilities over ISE. Note that PR could also be performed using ISE by manually defining a *PXML* file, though this is not covered here.

The ZyCAP core path (path_to_zycap/core/ise/hw) should be added to the project settings (**Project Options** → **Advanced** → **Project Peripheral Repository**) for including ZyCAP in user design. The core files are actually contained within path_to_zycap/core/ise/ipcore/pcores/zycap_v1_00_a.

The project has been implemented with the ISE Version 14.7. This can also be easily ported over to older versions of ISE/XPS and PlanAhead.

2.1 Building the Example Design

PR requires the designs to be compiled with a bottom-up flow. This means that the reconfigurable modules (RMs) have to be synthesized separately from the main project and later integrated into the design. Reconfigurable modules have to be synthesized without having *iobufs* for them to be treated as valid sub-designs. ISE/PlanAhead uses the switch *-noiobuf* during synthesis to achieve the same. The example design is

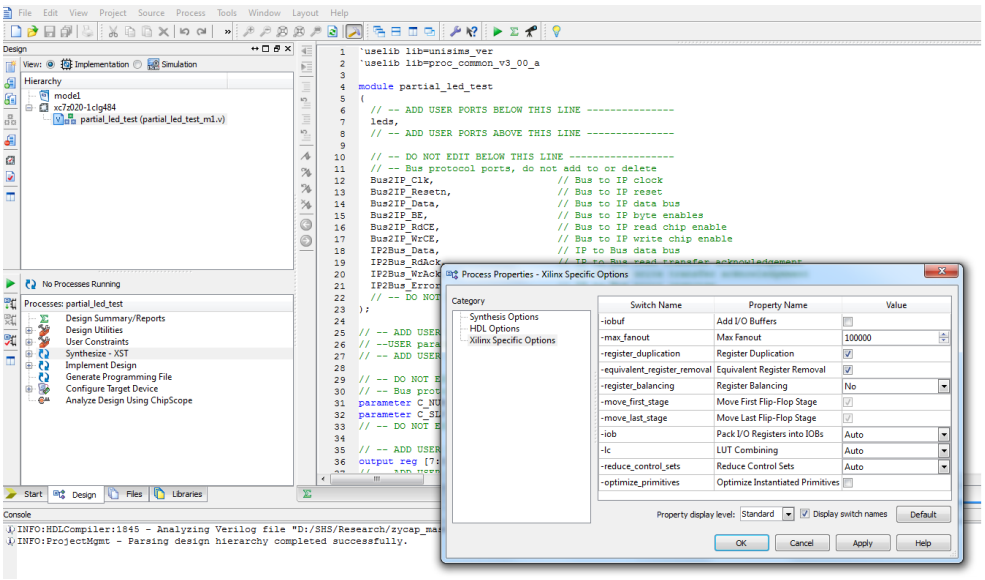


Figure 2.1: The project window corresponding to Mode 1 of example design

contained in the zycap/exdesign/ise folder.

In the following subsections, the different steps for regenerating the example design from the start. Pre-synthesized RMs and the different configuration bitstreams are present in the corresponding folders if the user wants to test the design directly. The same steps can be used to enable ZyCAP in a user design.

Synthesize RMs

The codes for the different reconfigurable modules are contained in the zycap/exdesign/ise/hw/modes. To start from scratch, open the ISE project mode1.xise, contained inside the folder mode1. The project window will highlight the top source (*partial_led_test_0*) and the hierarchy of the RM, as shown in Fig. 2.1

In *mode 1*, we have a set of 4 registers that increments the content written by the PS (by different amounts). Also, to easily identify the mode loaded on the FPGA, we use a counter whose count is displayed using the on-board LEDs. To synthesize the RM without iobuf, right-click the **Synthesize-XST** process and select **Process Properties** → **Xilinx Specific Options**. Disable (un-check) the *-iobuf* switch and click *Apply*. Next double click **Synthesize-XST** to run the synthesis, which generates the netlist

of the first RM.

It is also possible to use a Xilinx command-line to synthesize the RMs, using the command

cd *path_to_zycap/exdesign/ise/hw/modes/mode1*
xst -instyle ise -ifn "partial_led_test_0.xst" -ofn "partial_led_test_0.syr"

This generates the synthesized netlist for the current RM configuration. The synthesized netlist is saved as partial_led_test_0.ngc in the same folder.

Alternatively, the *mode1.tcl* file within the project folder may be sourced via a Xilinx tcl shell to run the command.

cd *path_to_zycap/exdesign/ise/hw/modes/mode1*
source mode1.tcl rebuilt_project

Similarly, open the second project (mode2.xise) in the folder *mode2*. This mode performs a decrement while writing and also decrements the led counter. Now re-run the same steps as above to generate the netlist for the second mode.

Build top-level XPS Project

The top-level design is an XPS project that integrates the ZyCAP core and other peripherals to the Zynq PS. The project (system.xmp) is contained within the folder zycap/exdesign/ise/hw/xps. Opening the project in XPS shows the instantiated IPs and their connectivity. User IPs could be added into the design using the **Hardware** → **Create or Import Peripheral** option.

In the project, it can be observed that IPs *zycap* and *partial_led_test* are instantiated as AXI peripherals in the design (with the name zycap_0 and partial_led_test_0 respectively), as shown in Fig. 2.2. These cores are listed under *Project Local PCores* as *User* defined cores. This is because the project uses a local copy of the ZyCAP and Partial_led_test IPs. To make them globally available, make sure that the path_to_zycap/core/ise/hw is listed as a repository in (**Project Options** → **Advanced** → **Project Peripheral Repository**).

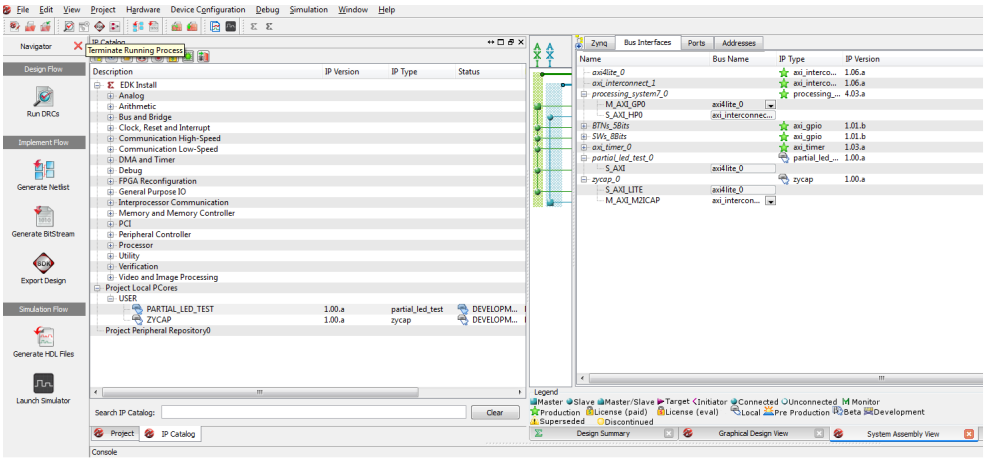


Figure 2.2: XPS design instantiating PS, ZyCAP, RM and support peripherals.

XPS uses absolute references for paths and hence it is required to remap the project repository to reflect local directory structure at the user end. The IP's are imported locally in the project to ensure seamless inference, and is not a mandatory requirement.

The local copies of the IP's are contained within zycap/exdesign/ise/hw/xps/pcores. Observing the sources for the IP partial_led_test reveals that it is an empty module and will be treated as a blackbox during synthesis, as required by the bottom-up synthesis flow.

Once the system is completely described, use the **Generate Netlist** button (or **Hardware** → **Generate Netlist** option) to synthesize the top-level design.

Partial Reconfiguration flow with PlanAhead

PlanAhead allows systematic implementation of the different configurations in a PR design. For using PR flow, users need to have

- A valid tool license that supports Partial Reconfiguration.
- Synthesized netlists of the top design and the individual modes, by following the steps above.

A PlanAhead project for the example design is provided in the folder zycap/exdesign/ise/hw/zycap_test. If a new project is started, make

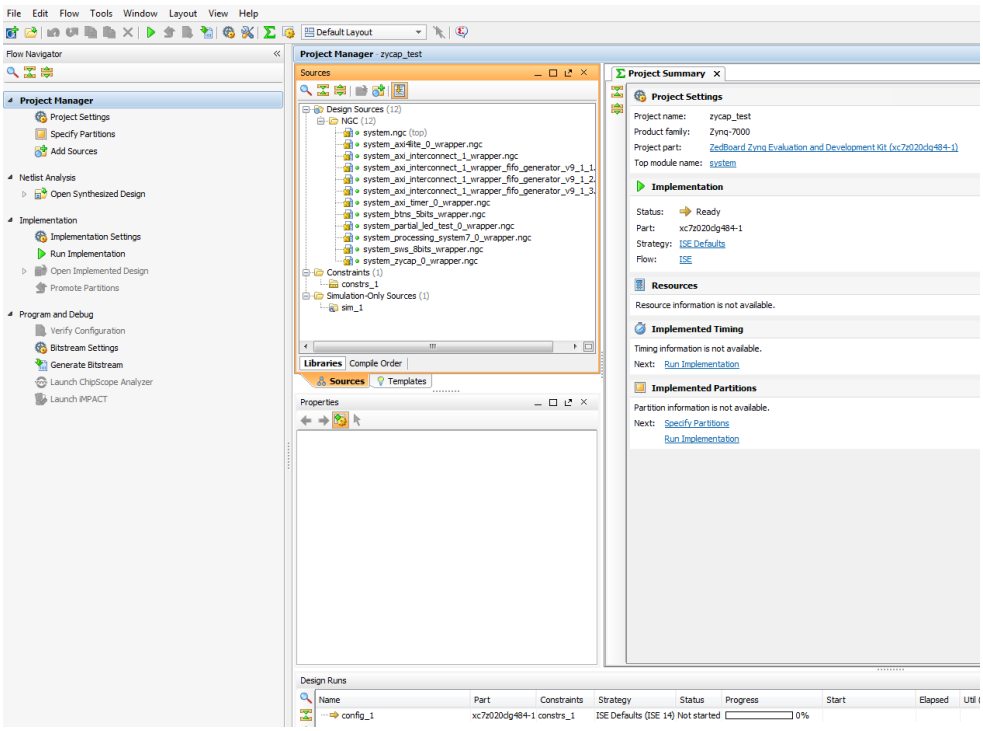


Figure 2.3: Sources listed in the Post-Synthesis PlanAhead project.

sure while creating the project that the *Post-Synthesis* project mode is selected and that the *Enable Partial Reconfiguration* option is checked. The provided project uses the same settings as the XPS project and targets the Zedboard. If the provided project is opened, then the source hierarchy is listed under *Design Sources* → *NGC* with **system.ngc** marked as the top file, as shown in Fig. 2.3. If a new project is created, add all the ngc sources from the zycap/exdesign/ise/hw/xps/implementation folder into the project using the **Add Sources** option.

Once the project is created, click the *Open Synthesized Design* option to see the hierarchy and to set the RMs as partially reconfigurable. While opening, warnings will appear showing that partial_led_test_0 cannot be resolved and is treated as a blackbox, as required by our design. Once these are dismissed, the hierarchy of modules will also show that partial_led_test_0 is marked differently to other modules, as shown in Fig 2.4

Next, the RM has to be set as a reconfigurable partition to preserve the PR flow during implementation. For this, right-click the blackbox module (partial_led_test_0) and select *Set Partition*. In the configuration window (see Fig. 2.1(a)), select the module as a reconfigurable partition

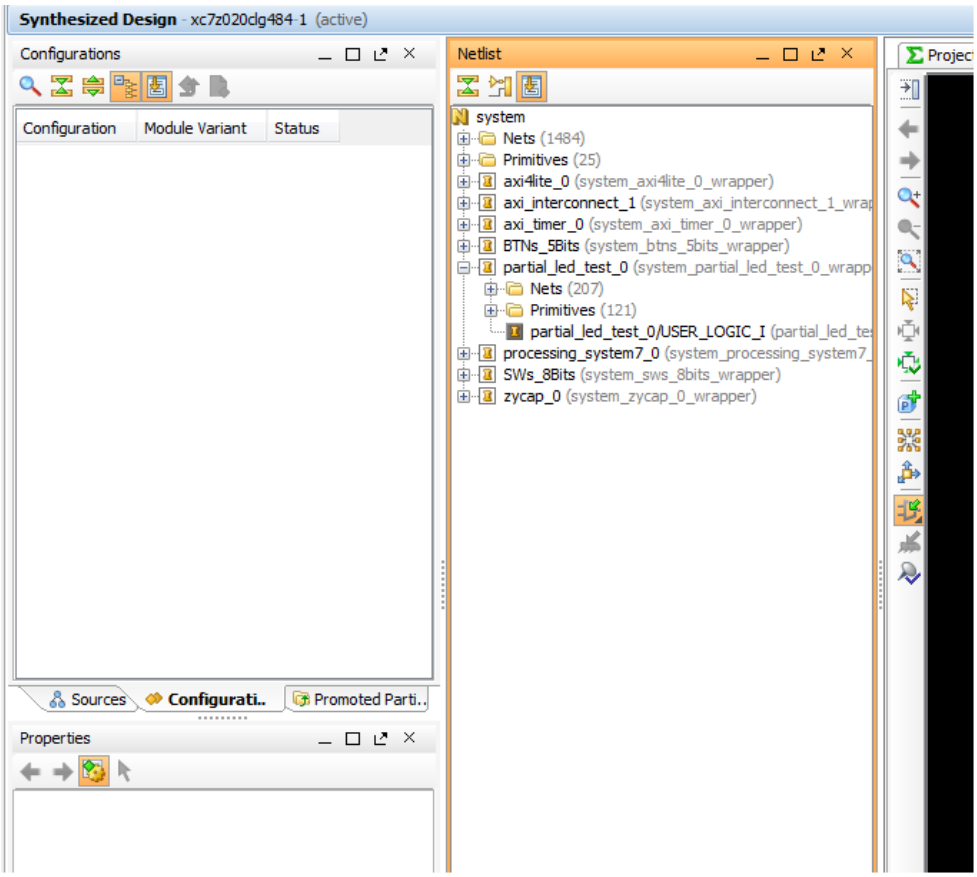
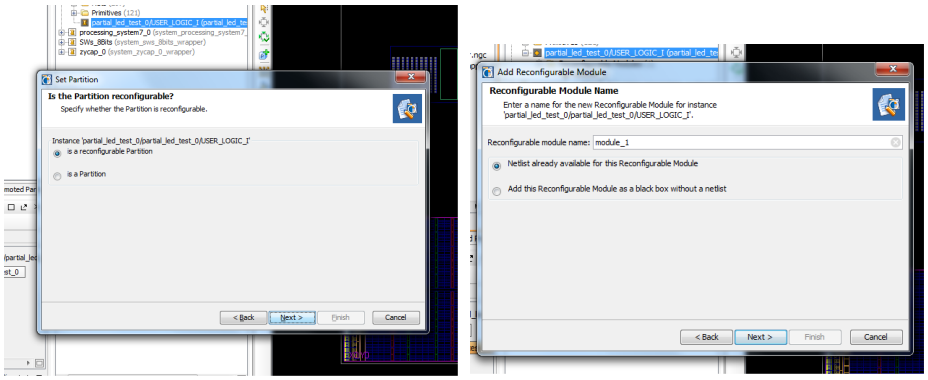


Figure 2.4: Hierarchy of modules with Partial_led_test_0 unresolved.

(in the second page) and click Next. In the next option, select *Netlist already ... Module* (see Fig. 2.1(b)) and select next to add the netlist for this configuration. In the next page, add the mode1.ngc netlist from path_to_zycap/exdesign/ise/hw/modes/mode1. Click next twice and then the finish button to complete the module creation.

Note: You may also create a blackbox module by choosing Add this RM as a blacbox without Netlist option, if you need to create a startup configuration that doesn't load any of the RMs. The example project does create a blackbox configuration as module_1, and this may be ignored if not required in the design.



(a) Select RM as reconfigurable partition.

(b) Add Sources for RM later.

Figure 2.5: Set the blackbox as a Reconfigurable Partition

To add our precompiled netlist *mode2.ngc* as another module of the RM, right click the module partial_led_test_0 and click **Add Reconfigurable Module**. The module names are given incrementally by the tool and in the **Specify top-level sources** section, add mode2.ngc (from from path_to_zycap/exdesign/ise/hw/modes/mode2) as the module netlist.

Next a Pblock has to be assigned to the top-level of the RM to ensure that the components within the RM are constrained to a specific location on the FPGA. For this, right-click the partial_led_test_0 netlist and click **Set Pblock Size**. The floorplan of the device would open up. Now select the size and location of the RM on the floorplan by selecting a rectangular region. There are certain imposed constraints for selecting a Pblock size, region and elements that can be included (see Hierarchical design UG748 and UG905 for PlanAhead and Vivado respectively).

Note: Xilinx places some constraints on the start and end edges of a Pblock and these have to be respected while choosing the Pblock location and size. Choosing the Pblock wrongly will result in DRC errors in the later stage. Note that the Pblock constraints are different in PlanAhead (constraints respecting older device requirements) and Vivado (constraints required for 7-series and beyond devices).

An example floorplan for the Pblock that respects the rules is shown in Fig. 2.1. Once completed, run the design rules check for PR using **Tools** → **Report DRC** and Select Partial Reconfig (only) to check for violations. Note that you may get a few warnings about uninitialised registers, and can be ignored.

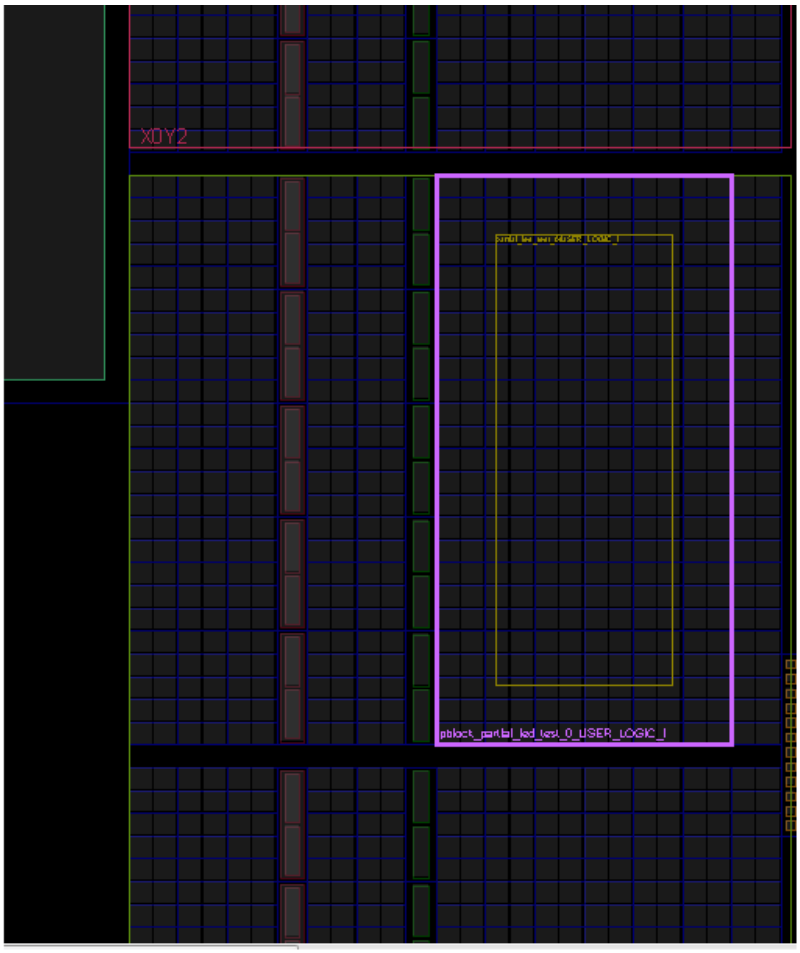


Figure 2.6: A valid floorplan for Zynq respecting PlanAhead requirements.

Implementing first configuration

In the design runs tab, click on the available runs to see the properties of the run. In the partitions tab, ensure that the required module variant is chosen. By default, this corresponds to module_1. However, if you have do not plan to implement the blackbox configuration, but have chosen it as one of the modules (i.e., added a module without adding its sources), then chose the appropriate module that you intend to implement (see Fig. 2.1). Once set up, run the configuration by right-clicking the config_1 runs and selecting **Launch Runs**.

This will execute the PlanAhead implementation scripts to optimise, place and route the logic. Once the implementation is complete, a pop-up

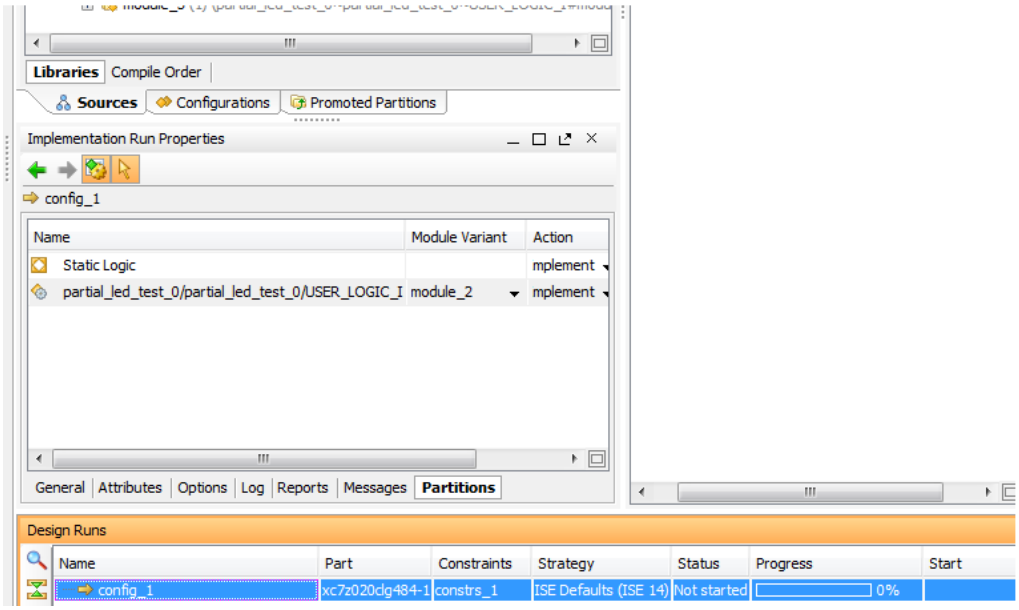


Figure 2.7: Configure Run property by using the partitions tab to choose the active RM.

window will open and wait for user input (see Fig. 2.1). Choose **Promote Partitions** and click *OK*.

Another window now opens up to display the partitions that will be promoted. Note that both the top-design (system) and the RM (module_1 or module_2, as per what you had configured in the run configuration) will be shown in the window. Select *OK* to proceed. Now the placement and routing of the current blocks will be saved and made available for re-use while implementing other configurations of the RM.

Implementing subsequent configurations

In the design runs tab, right-click and select **Add Runs**. In the run configuration, click the enlarge options button next to the text window (the button with ...) to expand the state of partitions being used in the run. For the static region, choose import as the option. For the RM (partial_led_test_0), choose the module variant that needs to be implemented next (module_2 or module_3), which will toggle the **Action** from *Import* to *Implement* (see Fig. 2.1). Choose Ok and click the next button to launch the newly created run.

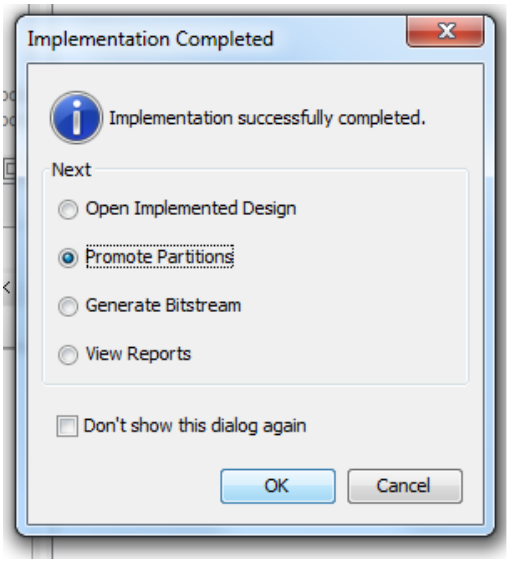


Figure 2.8: Pop-up window after successfully completing the implementation. Choose Promote from the options.

This will execute the PlanAhead implementation scripts to optimise, place and route the logic, with the new RM. Once the implementation is complete, a pop-up window similar to the one earlier (see Fig. 2.1) will open. Choose **Promote Partitions** and click *OK*. Repeat the steps if you have more configurations.

Verify Implemented Configurations

From the **Program and Debug** options in **Flow Navigator** window, select **Verify Configurations**. In the verify configuration window, select all the configurations that were implemented to check if these can co-exist. The check will now run across the selected configurations and will confirm that the implemented configurations are valid for the design.

Alternately, you may run

verify_config -runs { *config_1 config_2* } **-file** *./logfile.log*

where, *config_1* and *config_2* are the names of the implementation runs.

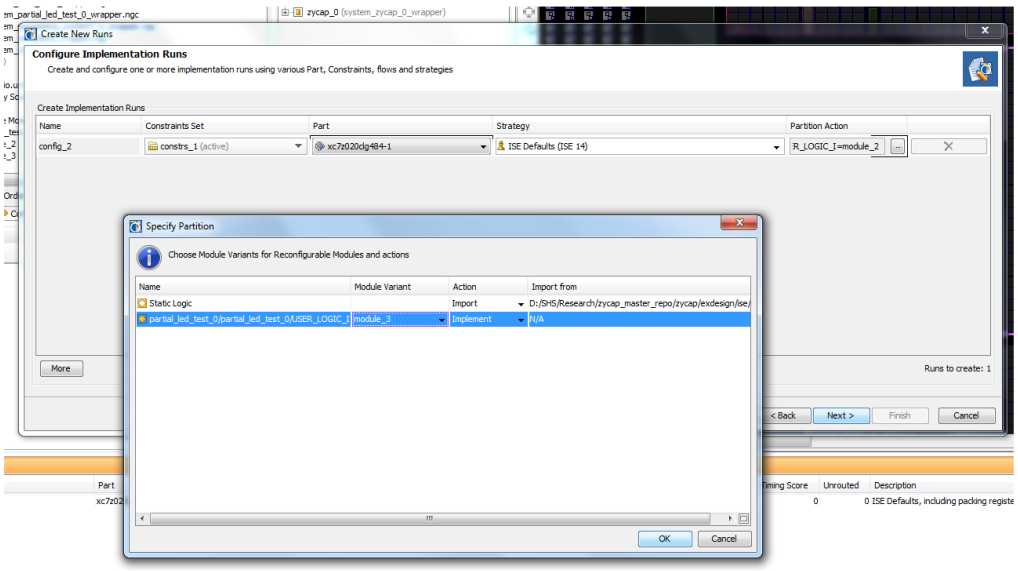


Figure 2.9: Choosing the configuration options for the newly created run.

Generate bitstreams

Once the configurations are verified, the bitstreams (full and partial) can be generated for the different configurations. In the *Design Runs* window, select all configurations (*Shift* + *Click*), right-click and select generate bitstreams. Use default options in the properties window that pops up to generate the '.bit' file. The partial bitstream will be of the form *config*_m_*HierearchicalModuleName_module*_n_*partial.bit*, where 'm' and 'n' are the numbers corresponding to the run configuration and module configuration. Rename the long file names to *config_1_partial.bit* and *config_2_partial.bit* from the file explorer.

> Note: The bitgen command does provide option to generate the binary file for partial bitstreams. However, these are not in the format that is required by ZyCAP. Hence we make use of the *Promgen* command from EDK environment to generate the byte-swapped binary for use with ZyCAP.

Once the full bitstreams are generated, open a Xilinx Shell from **Start → All Programs → Xilinx Design Tools → ISE Design Suite 14.7 → Accessories → ISE Design Suite Command Prompt**. Switch to the project

directory (*cd path_to_zycap/core/ise/hw/zycap_test*) and use the command below to generate the binary partial bitstream(s).

promgen -b -w -p bin -data_width 32 -u 0 *zycap_test.runs-/config_1/config_1_partial.bit* **-o** *./bitstreams/config1.bin*
promgen -b -w -p bin -data_width 32 -u 0 *zycap_test.runs-/config_2/config_2_partial.bit* **-o** *./bitstreams/config2.bin*

NOTE: Users can also make use of the pa_implementation.tcl script provided in the scripts folder to run the entire implementation through command line (use *source scripts/pa_implementation.tcl* in the PlanAhead tcl window from the hw project folder). Note that warnings and errors will have to be decoded from the textual report files which are generated at each step. The *promgen* command to generate the '.bin' file has to be executed separately.

2.2 Building the Software Application

To build the software infrastructure, information about the PS configuration and its set of peripherals have to be exported to the SDK. The software folder is located at *zycap/exdesign/ise/hw/xps/SDK* For this, open the XPS project (system.xmp) and select **Project** → **Export Hardware Design to SDK**. In the configuration window, un-check (deselect) **include bitstream and BMM file**, since this is not required in our current flow. Choose **Export Only** to export the hardware description file for use in Xilinx SDK. You may directly click the **Export and Open SDK** button to export the hardware design and open the SDK directly from the linked workspace. In this case, SDK environment is now invoked with the hardware definitions of the generated system. Alternatively, if SDK is launched separately, then point the workspace location to *path_to_zycap-/exdesign/ise/hw/xps/SDK*. In the SDK, the hardware details and their address ranges would be described, which can be cross checked with the ones in Vivado block design.

Next, add a board support package (bsp) for running the application on the Zynq platform, using **File** → **New** → **Board Support Package**. Choose the default settings (bsp name as 'standalone', target hardware as

'system_wrapper_hardware_platform_0', target CPU as 'ps7_cortexa9_0' and os type as 'standalone') and click finish.

Finally, import ZyCAP drivers and the standalone PR application file into the project. Create an application project using the **File** → **New** → **Application Project** menu. Name the project as 'pr_app' and choose 'standalone_bsp_0' as the bsp. In the next menu, choose 'Empty Application' to create an empty application project. For importing the application file and the driver, use the **File** → **Import** menu. Choose **General** → **File System** option, and import the contents of ZyCAP driver folder (zycap/core/ise/driver) into the 'pr_app/src' folder. Similarly import the pr_app file from zycap/core/ise/app/pr_app.c into the 'pr_app/src' folder.

Copy the partial '.bin' files generated earlier to the root directory of the SD card and attach it to the Zedboard. Next, configure the FPGA using **Xilinx Tools** → **Program FPGA** and choose *path_to_project/zycap_test/zycap_test.runs/config_1/config_1.bit* as the configuration file and click **Program FPGA**. Then load the application 'elf' file to the PS to run the example design. A terminal program like 'TeraTerm' can be used to observe the UART outputs. Alternatively, the UART output can be redirected to the SDK console by configuring the 'Run/Debug Configuration'.

Three

ZyCAP in Vivado Environment

Vivado is the new integrated tool-flow from Xilinx Inc. for compiling user designs for 7 series devices and beyond. Vivado integrates block design generation (which was earlier done through the XPS flow) and the design compilation flow into a single workspace environment. Further, Vivado also supports partial reconfiguration in the same environment, although only through *tcl* scripts or command line interface. ZyCAP has also been ported into the Vivado tool-flow, which can be imported into the design as a *packaged IP*. Note that the ZyCAP core path (path_to_zycap/core/vivado/ipcore) should be added to the project settings (**Project Settings** $\rightarrow$ **IP** $\rightarrow$ **IP Repositories**) to drag and drop ZyCAP into a user design.

NOTE:

- ZyCAP for Vivado has been tested on Version 2014.2 of Vivado. Lower versions of the tool produces errors (routing antennas) during partial reconfiguration enabled flow (See AR# 62231,63168).
- Assigning pblock to a hierarchically lower RM results in the tool applying the pblock assignment to the top level instance of the RM. This would result in an error during the *implementation* phase (*Error: each RM must have a pblock assigned to it or lower cells.*). This could probably be solved by hierarchically assigning pblocks and have not been tested.

3.1 Building the Example Design

Vivado supports PR flow using the bottom-up synthesis approach. Reconfigurable modules have to be synthesized without having *iobufs* for

them to be treated as valid sub-designs. Vivado uses the *Out-of-Context* (OOC) synthesis switch to achieve the same. The example design is contained in the zycap/exdesign/vivado folder.

In the following subsections, the different steps for regenerating the example design from the start. Pre-synthesized RMs and the different configuration bitstreams are present in the corresponding folders if the user wants to test the design directly. The same steps can be used to enable ZyCAP in a user design.

Synthesize RMs

The codes for the different reconfigurable modules are contained in the zycap/exdesign/vivado/hw/modes. To start from scratch, open a new Vivado project and add the module files into the design (alterantively, use the mode1.xpr project inside the mode1 directory) The project window will highlight the top source (*system_partial_led_test_0_0*) and the hierarchy of the RM, as shown in Fig. 3.1 As in the previous case, *mode 1* uses a set of 4 registers which increments the content during a write from the PS (file partial_led_test_v1_0_S00_AXI, lines 229,236,243,250), and an incremental counter (line 404 in the same file) which is displayed using on-board LEDs.

To synthesize the RM in OOC mode, use the following commands in the *tcl console*:

cd *path_to_zycap/exdesign/vivado/hw/modes*
open_project *modes.xpr***synth_design -top** *top_file* **-mode out_of_context -part** *part_number*

For the example design,
top_file = system_partial_led_test_0_0
part = XC7z020clg484-1

This generates the synthesized netlist for the current RM configuration. To save the synthesized netlist as a design checkpoint (dcp) for importing into the top design,

write_checkpoint -force *./mode1.dcp*

Similarly, generate the RM for the subsequent modes by altering the design files and generating the dcps in the OOC configuration. For the

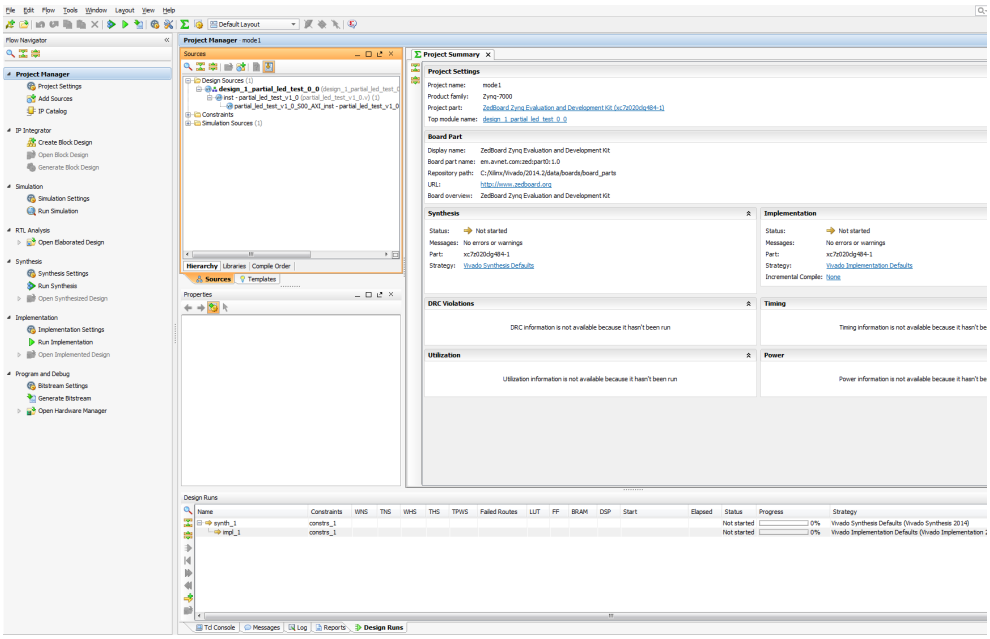


Figure 3.1: The project window corresponding to Mode 1 of example design

example design, alter the lines 229, 236, 243, 250 and 404 in the file file partial_led_test_v1_0_S00_AXI by replacing the '+' sign by the '-' sign. Now re-run the above commands and save the synthesized netlist as *mode2.dcp*

Synthesize Top-level design

For implementing the circuit, a top-level design needs to be synthesized, which can then be linked to the different RMs for implementing different functionalities. ZyCAP is easily instantiated using the block design generator under the IP integrator flow, which is similar to the XPS design flow that was discussed earlier.

To synthesize the top-level of the example design, open the system_top-.xpr project inside the zycap/exdesign/vivado/hw/zycap_test directory. A project window opens containing a block design (*system*) and its verilog wrapper file. Open the block design by double clicking the name in the *sources* window to reveal the block design.. The block design instantiates a Zynq PS, the ZyCAP IP core and a *partial_leds_test* module (which is the RM), along with the axi interconnects and the processor reset module, as shown in Fig. 3.2 In the address editor tab, observe that

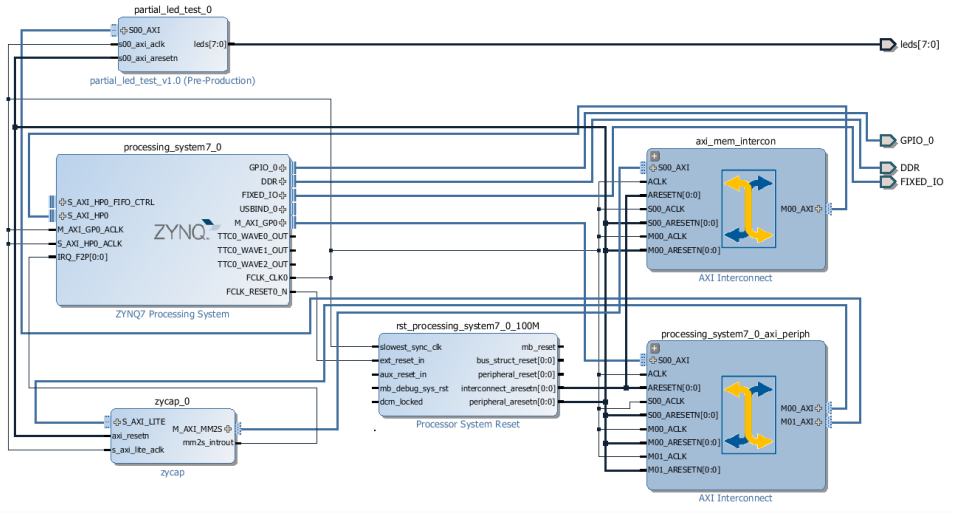


Figure 3.2: Block design instantiating PS, ZyCAP, RM and support peripherals.

ZyCAP has an offset address of 0x40400000 (64K). You can validate the block design to ensure that there are no errors at this stage by choosing **Tools** → **Validate Design** (or directly the F6 key). If there are no further errors, the save and close the block design.

In the project window, right click the block design (system.bd) to generate/update the top-level wrapper file, with the default option (*Let Vivado manage* ...). For synthesizing and implementing the block design, Vivado needs to compile the block design and generate the HDL instantiation files for each of the IPs. This is achieved by clicking the **Generate Block Design** button in the *Flow Navigator* window.

Note: After generating the files for the block design, expand the partial_leds_test IP to reveal its top file (system_partial_led_test_0_0) and note its location (should be path_to_project/system_top.srcs/sources_1/bd/system/ip/system_partial_led_test_0_0/synth). Locate the file using file explorer, open the file using an editor and comment out the instantiation of partial_led_test_v1_0 (i.e., convert it to a blackbox for PR flow by commenting out lines 125 – 151).

Once completed, synthesize the top design using the command (in *tcl console*)

cd *path_to_zycap/exdesign/vivado/hw/zycap_test*
synth_design -top *top_file* **-part** *part_number*
write_checkpoint -force *./checkpoints/top_synth.dcp*

Where,
top_file = system_wrapper
part = XC7z020clg484-1

Alternately, you can use the scripts provided in the zycap/exdesign/-vivado/hw/scripts folder to perform these steps. In the tcl console, you can use

cd *path_to_zycap/exdesign/vivado/hw*
source *scripts/system_create.tcl*

to create a Vivado project (system_top) within the directory zycap_test, generate the block design and its instantiation files. For correct working, make sure that you run the script by switching to the directory zycap/ex-design/vivado/hw in the *tcl console*, since the paths to the ZyCAP IP is fixed in relation to this directory. As in the above case, locate the gener-ated HDL top file of our RM instance (system_*RM name*) and comment the instantiation of the RM in the file to convert it to a blackbox. Then, move into the project folder (zycap_test), synthesize the top-level design using the same **synth_design** command and save the check point using the **write_checkpoint** command.

Create the first configuration

Vivado expects PR blocks to be marked as RECONFIGURABLE cells and assigned with a location constraint before implementing them. To achieve that, we open the synthesized top level netlist (*top_synth*) and use the user interface to assign the constraints to the RM(s). Also, the lo-cation constraints for the system IO's (LEDs, and other PL/PS pins) must be imported into the project before implementation.

Note: The following commands are to be executed from within the project directory (zycap_test).

First, open the synthesized top module by

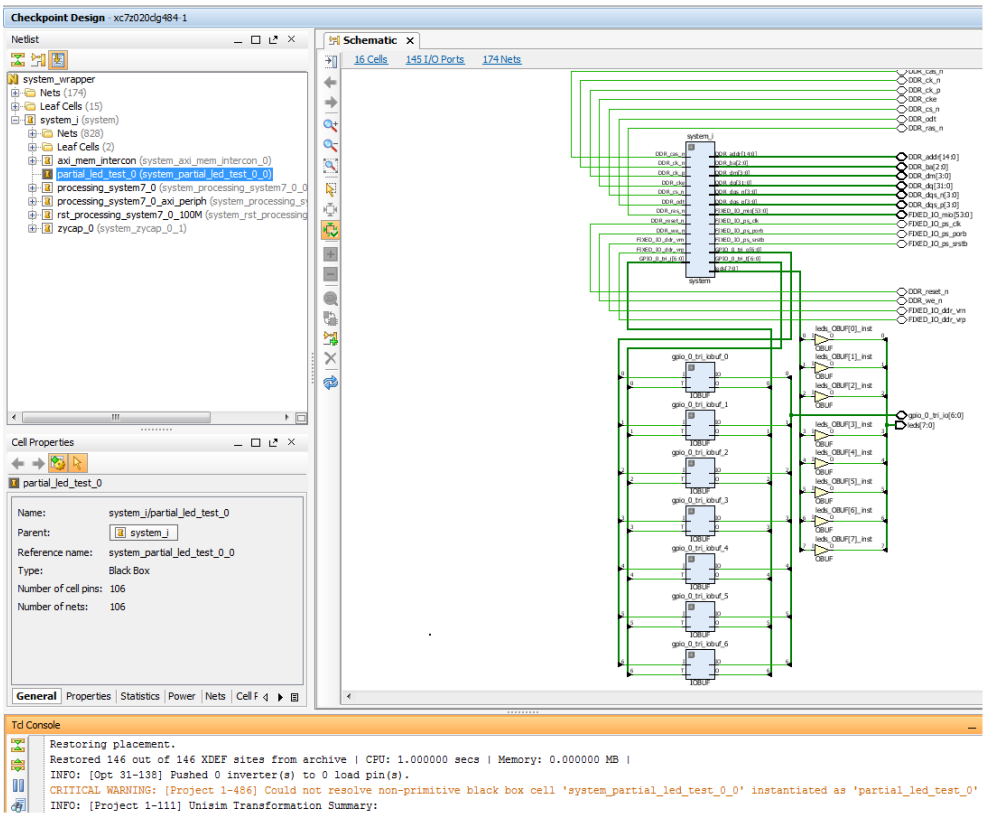


Figure 3.3: The hierarchy of synthesized top-file. Note that the cell partial_led_test_0 is greyed out, showing that its internal structure could not be resolved.

open_checkpoint *./checkpoints/top_synth.dcp*

This opens up a project window with a critical warning that the non-primitive cell partial_led_test_0 is treated as a blackbox, which can be ignored. The system hierarchy would resemble the one shown in Fig. 3.3.

Next, mark the cell partial_led_test_0 as a RM using

set_property HD.RECONFIGURABLE *true* **[get_cells** *system_i/partial_led_test_0*]

Vivado will now check if your installation has the license for compiling PR projects and will show that the feature is available if it can checkout

the license. In the next step, the system level IO constraints has to be read into the project using

read_xdc *./constraints/top_io.xdc*

Next, link the first RM to the top level project by reading the generated checkpoint.

read_checkpoint *../modes/mode1.dcp* **-cell** *system_i/partial_led_test_0*

The design will now be updated and the netlist will show that the cell *partial_led_test_0* has been resolved (has child cells and logic corresponding to *mode1*), as shown in Fig. 3.4.

Next, the placement of the RM has to be locked in the design. This is required for each RM in a PR enabled design. To do this, right-click the cell *partial_led_test_0* in the netlist window select **Floorplanning** → **Draw Pblock**. This opens up the floor-plan of the device. Select the region where you want the RM to be placed by drawing on the floor-plan.

7-series devices places some constraints on the start and end edges of a Pblock and these have to be respected while choosing the Pblock location and size. Generally, the Pblock left edge must start on an even row and must include even number of CLB rows (right edge constraint). Choosing the Pblock wrongly will result in DRC errors in the later stage. Also, to choose *reset after reconfiguration* property, the Pblock has to be aligned to the boundaries of the clock region.

An example floor-plan that respects all the above rules and allows *reset after reconfiguration* to be applied to the RM property is shown in Fig. 3.5. Note that the Pblock must include all the resource types that may be needed by your RM (i.e., Slices, DSPs and BRAMs). This can be seen in the *statistics* tab of the *Pblock properties* window. Once the Pblock location is chosen, its validity can be checked by the running the design rules check (DRC) command from **Tools** → **Reports** → **Report DRC**. From the window, choose only *Partial Reconfiguration* and run the commands. Any violation would now be reported in a DRC tab. If there are violations, these needs to be corrected before proceeding further.

Once the floor-planning is done satisfying all the requirements, the current constraints can be saved using

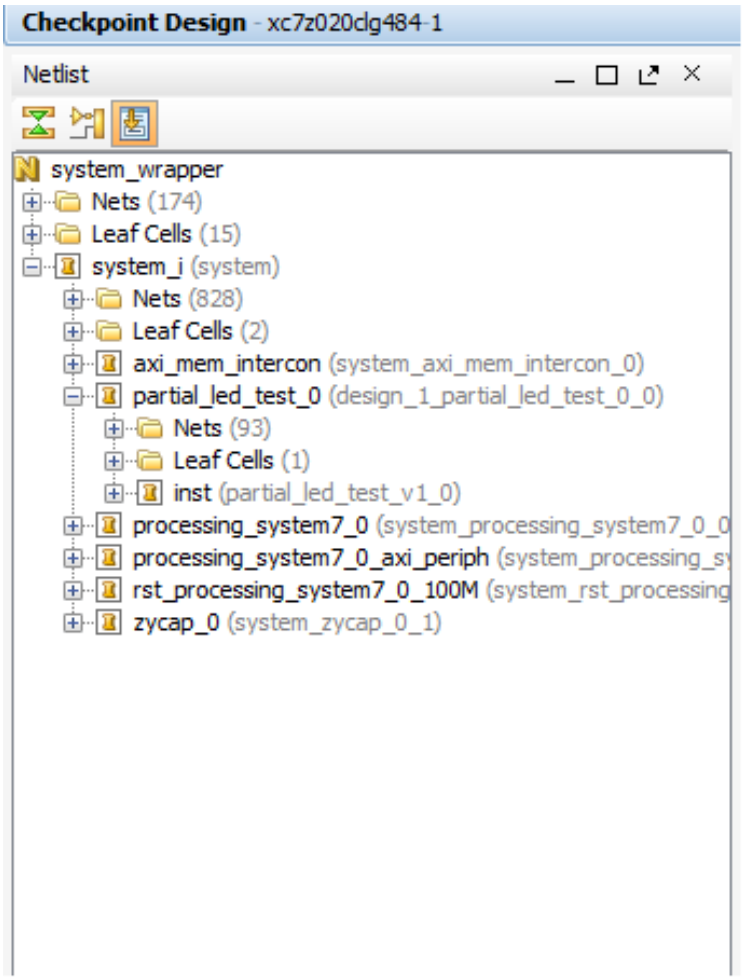


Figure 3.4: The netlist hierarchy shows that the blackbox has been resolved after linking *mode1.dcp* with top design.

write_xdc -force *./constraints/top_fplan.xdc*

It is also recommended to save the current design state by writing the checkpoint as

write_checkpoint -force *./checkpoints/top_link_mode1.dcp*

Next, we can run the implementation steps using the commands

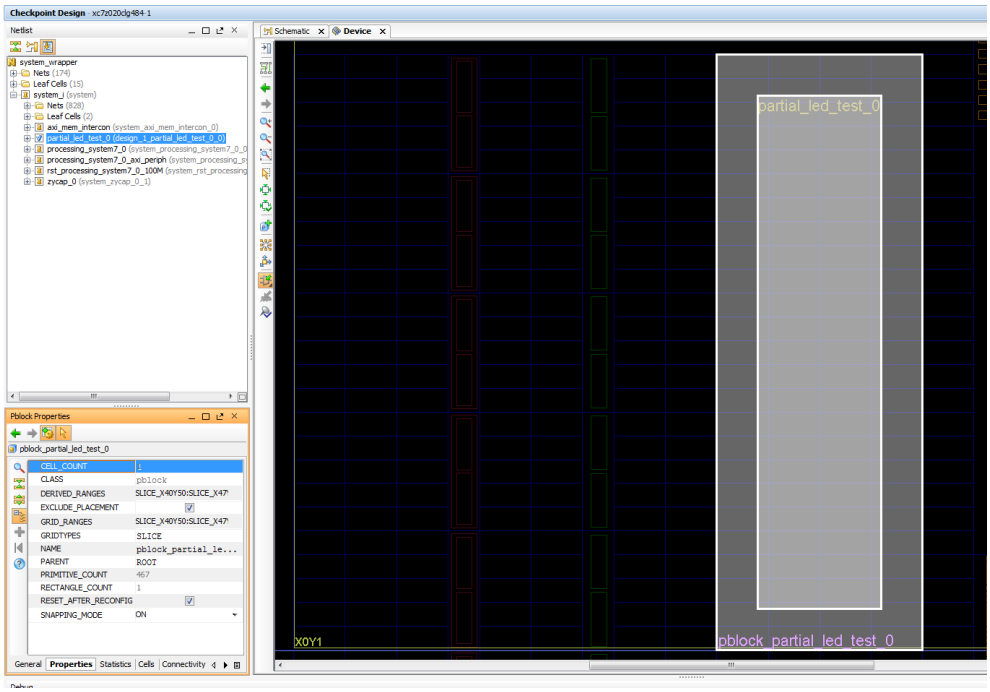


Figure 3.5: An example floorplan for the Pblock that allows *reset after reconfiguration* property to be directly applied and respects the floorplanning rules of 7-series.

opt design
place design
route design

The commands are executed sequentially to optimise the linked design, followed by placement and routing. Once completed, save the design checkpoint corresponding to the implementation.

write_checkpoint -force *./checkpoints/top_mode1_routed.dcp*

The steps can also be done by using the script *create_mode1.tcl* in the scripts directory

source *../scripts/create_mode1.tcl*

Preserve the configuration of Static logic

The configuration of static logic needs to be preserved to ensure that the second configuration is also implemented with the exact same constraints and routing (of the static blocks). First, the design is updated to unlink components corresponding to *mode1* alone and convert it to black box.

update_design -cell *system_i/partial_led_test_0* **-black_box**

Next, the placement and routing of the remaining components (static logic) has to be locked. This is done via

lock_design -level routing

Finally, preserve the state by writing it as a checkpoint

write_checkpoint -force *./checkpoints/static_routed.dcp*

Now the design can be closed by

close_design

The steps can also be done by using the script *preserve_static.tcl* in the scripts directory

source *../scripts/preserve_static.tcl*

Implement second configurations

To implement further configurations corresponding to the RM(s), start with the preserved design

```
open_checkpoint ./checkpoints/static_routed.dcp
```

Link the second RM to the opened design

```
read_checkpoint -cell system_i/partial_led_test_0 ../modes/mode2.dcp
```

Implement the second design and write the checkpoint

```
opt design
place design
route design
```

```
write_checkpoint -force ./checkpoints/top_mode2_routed.dcp
close_design
```

The steps can also be done by using the script *create_mode2.tcl* in the scripts directory

```
source ../scripts/create_mode2.tcl
```

Checking the configurations

Before generating the bitstreams for the different configurations, the tool provides a method to check if the the two configurations are valid for the same static design. To check this,

```
pr_verify -full_check ./checkpoints/top_mode1_routed.dcp ./checkpoints/top_mode2_routed.dcp
```

This will extensively check for any violations and if they are compatible, then the configurations are valid for use.

Generating bitstreams

To generate the bitstream, open the implemented checkpoint and use the **write_bitstream** command.

```
open_checkpoint ./checkpoints/top_mode1_routed.dcp
write_bitstream -force ./bitstreams/mode1.bit
close_design
open_checkpoint ./checkpoints/top_mode2_routed.dcp
write_bitstream -force ./bitstreams/mode2.bit
close_design
```

ZyCAP requires the bitstreams to be in byte-reversed '*.bin*' format. However, using the **-bin** switch with the write_bitstream command does not generate the binary file in the format required by ZyCAP. Hence we make use of the **bootgen** command that is part of the Xilinx SDK environment. The **bootgen** command requires a boot information file (.bif) file as its input, which specifies the input files to be processed by the command. Copy the content below and save it as a '.bif' file in the 'bitstreams' directory.

```
the_ROM_image:
{
mode1_pblock_partial_led_test_0_partial.bit
mode2_pblock_partial_led_test_0_partial.bit
}
```

Now open a Xilinx command prompt. In a windows environment, open it from **Start** → **All Programs** → **Xilinx Design Tools** → **Vivado 2014.2** → **Xilinx SDK 2014.2 Command Prompt**. Alternatively, you can start a standard windows command shell and execute the 'settings32.bat' (or 'settings64.bat') command from ***Vivado Installation Folder*/2014.2/** to get access to Xilinx commands. Now 'cd' to the *bitstreams* on the command line and execute

bootgen -image *bif_file.bif* **-w -process_bitstream bin**

This will now generate the correct binary files for use with ZyCAP. Rename these files to *mode1.bin* and *mode2.bin* respectively for easy reference. Copy them to the root directory of the SD card.

3.2 Building the Software Application

To build the software infrastructure, information about the PS configuration and its set of peripherals have to be exported to the SDK. The software folder is located at *zycap/exdesign/vivado/sw* For this, open the top level project from Vivado tcl interface using

open_project *path_to_project/system_top.xpr*

Once opened, export the hardware design from tcl console using the command

write_hwdef -file *path_to_sw_folder/system_wrapper.hdf*

This will export the hardware description file for use in Xilinx SDK. Now launch SDK through the tcl console using the command

launch_sdk -workspace *path_to_sw_folder* **-hwspec** *path_to_sw_folder/system_wrapper.hdf*

SDK environment is now invoked with the hardware definitions of the generated system. In the SDK, the hardware details and their address ranges would be described, which can be cross checked with the ones in Vivado block design.

Next, add a board support package (bsp) for running the application on the Zynq platform, using **File** → **New** → **Board Support Package**. Choose the default settings (bsp name as 'standalone', target hardware as 'system_wrapper_hardware_platform_0', target CPU as 'ps7_cortexa9_0' and os type as 'standalone') and click finish. In the configuration window that pops up, enable **xilffs** (Generic FAT filesystem library).

Finally, import ZyCAP drivers and the standalone PR application file into the project. Create an application project using the **File** → **New** → **Application Project** menu. Name the project as 'pr_app' and choose

'standalone_bsp_0' as the bsp. In the next menu, choose 'Empty Application' to create an empty application project. For importing the application file and the driver, use the **File** → **Import** menu. Choose **General** → **File System** option, and import the contents of ZyCAP driver folder (zycap/core/vivado/driver) into the 'pr_app/src' folder. Similarly import the pr_app file from zycap/core/vivado/app/pr_app.c into the 'pr_app/src' folder.

Once the project has been successfully compiled, configure the FPGA using **Xilinx Tools** → **Program FPGA** and choose *path_to_project/zycap_test/bitstreams/mode1.bit* as the configuration file and click **Program FPGA**. Then load the application 'elf' file to the PS to run the example design. A terminal program like 'TeraTerm' can be used to observe the UART outputs. Alternatively, the UART output can be redirected to the SDK console by configuring the 'Run/Debug Configuration'.

Four

Using ZyCAP with Linux OS

Linux has been a popular OS choice for Zynq boards, given their capable ARM cores. Different flavours of Linux have been ported to Zynq, including official ones from Xilinx as well as community developed ones like Xillinux (from Xillybus). For this setup, we have modified the Xillinux kernel to add support for ZyCAP reconfiguration management. The implementation provides terminal access to the Zynq board, which can be further extended by incorporating the VGA interfaces available in the Xillinux distribution. We make use of the hardware logic generated with the Vivado environment (see chap. 3) for the setup.

NOTE:

- The same setup can be achieved using the hardware generated from ISE/PlanAhead tools. However, the device-tree has to be updated to reflect the changes in hardware configuration (base-address of peripherals, peripheral <compatible> names). The device-tree corresponding to the hardware project can be exported from the corresponding SDK project.
- Alternatively, you may edit the working device-tree blob (dts file) to reflect the changes in your hardware configuration and generate the device-tree binary file. **Do not change the PS peripheral details in the original device tree as this may need the kernel to be recompiled**.

4.1 Preparing the Boot Image

The procedure followed is identical to the one used in Xillinux setup. An 8GB or higher SD card is needed for the setup. **Note that the SD card that came with ZedBoard is not suitable for this setup**. The setup

requires partitioning the SD card into two partitions, uncompressing the boot image and copying them to the first partition, and finally adding the bootloader, PL bitstream and the kernel Image to the second partition. The steps below details the methods to achieve this on Windows and Linux platforms.

Windows host

On a Windows platform, it is recommended to use the "USB Imaging Tool [6]" to copy the image to the SD card. Windows 7 or later platforms which have a built-in SD card reader may also need "Win32 Disk Imager [7]". The *USB Imaging Tool* can be used in both GUI mode and command line.

For GUI mode, start the program by double-clicking the 'USB Imaging Tool.exe' file from the extracted directory. In the tool, choose the *Device Mode* on the left-hand side drop-down menu (see Fig. 4.1). Choose *Restore*, select Compressed (zip) image files as the type in the pop-up window. Now select the zylinux.imz from the zycap/linux/sdimg folder, and the tool will start restoring the image. Once done, safely remove the SD card. The compressed image contains the partition table for preparing the SD Card, which creates a FAT file system for adding the boot files and the second partition for the Linux root file system. Hence the card needs to be removed from the host machine so that the host can recognise the change in the partition type, next time the SD card is plugged in.

> **NOTE:** The GUI may fail to run on some systems, since it requires "Microsoft .Net Component [8]" to be installed.

Alternatively, you can use windows command line to copy the image file. For this, first identify the enumeration number of the SD Card once it is plugged into the host machine using

```
D:/usbit>usbitcmd l
```

Note the number under the *Device* heading (*dev_num*). Now, initiate copy using

```
D:/usbit>usbitcmd r dev_num path-to-zycap/linux/sdimg/zylinux.imz /d
/c
```

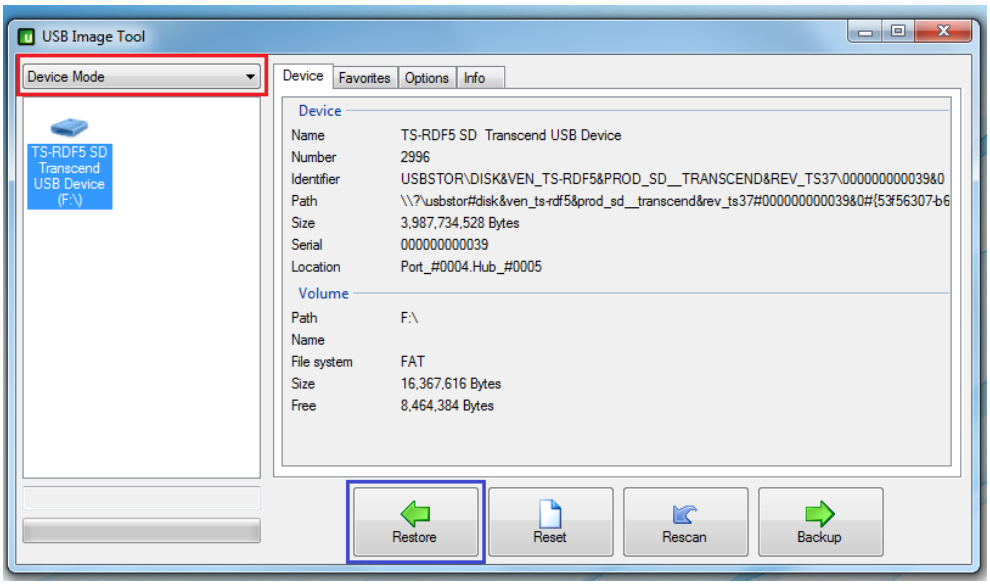


Figure 4.1: GUI window of USB Imaging Tool. Note that you need to select *Device Mode* in the drop-down menu.

Once completed, remove the SD card from the host so that it can register the change in file-system on the SD Card.

Finally, to copy the boot files, reinsert the SD card to the host machine. A FAT partition will now be visible in the SD card with a *uImage* file, to which you can copy the *BOOT.bin* and *devicetree.dtb* files from the zycap/linux/bootfiles folder. Also, copy the PL bitstream into the same folder and rename it to *config_1.bit*. Eject the card when done.

Linux host

First, unzip the compressed image using

```
# unzip path-to-zycap/linux/sdimg/zylinux.imz
```

This creates the uncompressed image file (zylinux.ima) which can be directly written into the SD card. To copy, use

```
# dd if=zylinux.ima of=path-to-sdc bs=512
Where, path-to-sdc is generally /dev/sdc
```

> **NOTE:** Check the location of the SD card by checking the /var/log/messages or /var/log/syslog log file to see the name corresponding to the SD Card. *sdc* shown here is the case with our environment and it may be different on a different host machine.

Once the copy is completed, eject the SD card to allow the host to recognise the change in file system. Re-insert the SD Card and copy the boot files (*BOOT.bin* and *devicetree.dtb* from *bootfiles* folder) and the PL bitstream (to be renamed as config_1.bit) to the partition. If manual mounting is needed, use

```
# mkdir /home/username/sd
# mount /dev/sdb1 /home/username/sd
```

Now, copy the boot files to the folder /home/*username*/sd and unmount the SD card.

> **NOTE:** Mount and Umount may require root/super-user privileges.

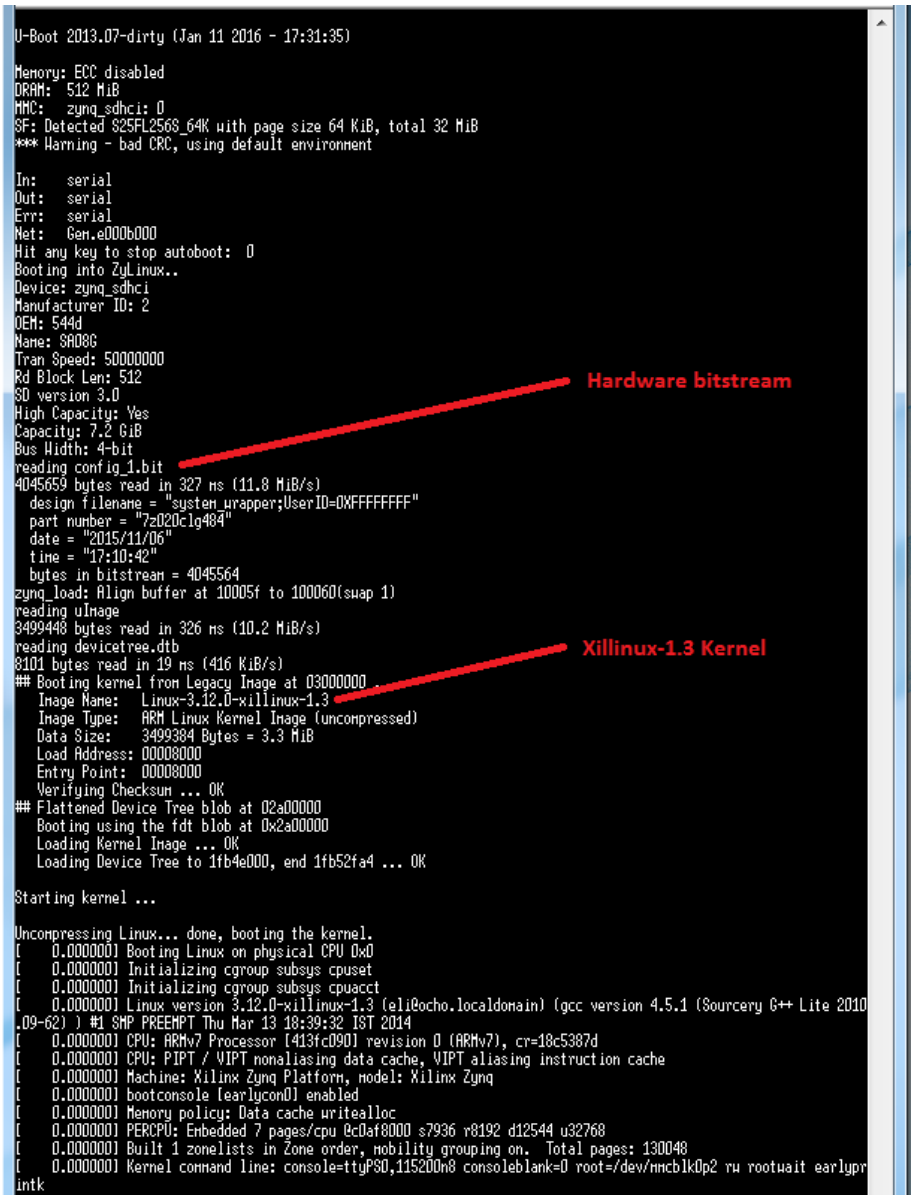
4.2 Booting the ZedBoard

The ZedBoard provides jumpers J7 – J11 for supporting different boot sources (like SD Card, JTAG, NV–RAM) for booting the Zynq PS and for configuring the PL. For this setup, we need to configure these jumpers to boot the Zynq from the SD card. This is achieved by setting J9 and J10 to high (Vcc), while the others are set to low (GND) (see ZedBoard Hardware Guide [9] for more details). Now insert the SD card into the slot and power on the board.

You can view the boot log by using a terminal program (like TeraTerm) and connecting to the UART–to–USB bridge port on the ZedBoard. A typical boot screen is shown in Fig. 4.2.

Once the boot is completed, you can check if ZyCAP has been recognised in the system by using

```
# dmesg | grep zycap
```




```
U-Boot 2013.07-dirty (Jan 11 2016 - 17:31:35)

Memory: ECC disabled
DRAM:  512 MiB
MMC:   zynq_sdhci: 0
SF: Detected S25FL256S_64K with page size 64 KiB, total 32 MiB
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   Gem.e000b000
Hit any key to stop autoboot:  0
Booting into ZyLinux..
Device: zynq_sdhci
Manufacturer ID: 2
OEM: 544d
Name: SA08G
Tran Speed: 50000000
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 7.2 GiB
Bus Width: 4-bit
reading config_1.bit
4045659 bytes read in 327 ms (11.8 MiB/s)
  design filename = "system_wrapper;UserID=0XFFFFFFFF"
  part number = "7z020clg484"
  date = "2015/11/06"
  time = "17:10:42"
  bytes in bitstream = 4045564
zynq_load: Align buffer at 10005f to 100060(swap 1)
reading uImage
3499448 bytes read in 326 ms (10.2 MiB/s)
reading devicetree.dtb
8101 bytes read in 19 ms (416 KiB/s)
## Booting kernel from Legacy Image at 03000000 ...
   Image Name:   Linux-3.12.0-xillinux-1.3
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3499384 Bytes = 3.3 MiB
   Load Address: 00008000
   Entry Point:  00008000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 02a00000
   Booting using the fdt blob at 0x2a00000
   Loading Kernel Image ... OK
   Loading Device Tree to 1fb4e000, end 1fb52fa4 ... OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.12.0-xillinux-1.3 (eli@ocho.localdomain) (gcc version 4.5.1 (Sourcery G++ Lite 2010
.09-62) ) #1 SMP PREEMPT Thu Mar 13 18:39:32 IST 2014
[    0.000000] CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[    0.000000] Machine: Xilinx Zynq Platform, model: Xilinx Zynq
[    0.000000] bootconsole [earlycon0] enabled
[    0.000000] Memory policy: Data cache writealloc
[    0.000000] PERCPU: Embedded 7 pages/cpu @c0af8000 s7936 r8192 d12544 u32768
[    0.000000] Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 130048
[    0.000000] Kernel command line: console=ttyPS0,115200n8 consoleblank=0 root=/dev/mmcblk0p2 ru rootwait earlypr
intk
```

Figure 4.2: Boot log captured from Serial Terminal.

This should present a confirmation as shown in Fig. 4.3. Note that the timestamp may be different, depending on the time taken to boot into the linux kernel.

You may also **ssh** into the Zedboard if you have connected the board to an Ethernet port.

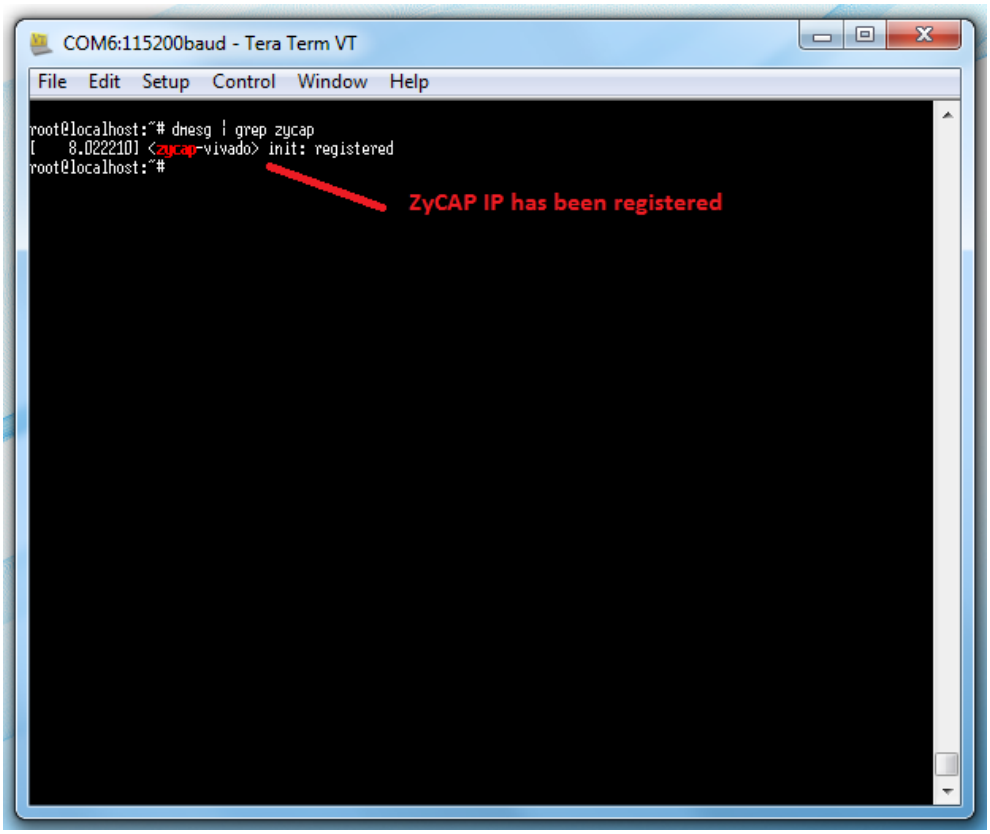


Figure 4.3: Check if ZyCAP has been recognised as a peripheral.

4.3 Executing the example application

From the home directory, switch to *zycap_test* on the terminal application (Teraterm or through ssh). In this folder, you can see the example application C file (*prexapp.c*), another C file that interfaces with the registers in the PR region (*checkdat.c*), the partial bitfiles corresponding to the different modes (*mode1.bin*, *mode2.bin*) and an *oled* folder that contains the binary bit-maps that illustrate which mode is currently active on the OLED display of the Zedboard. Create the executables for the example applications (prexapp.c) using

```
# gcc -o execname prexapp.c -lzycap
```

This will compile the application program by invoking the ZyCAP libraries and create the executable. To run the tests, use

```
# ./execname mode1
```

This command will run the reconfiguration 4-times from mode1 → mode2 → mode1 → mode2: the first-two times with un-buffered bitstreams since these have not been read before, and the next two using buffered bitstreams. The runs will show the bandwidth achieved by ZyCAP when the partial bitstreams are un-buffered (first two cases) and when it is buffered (last two cases).

The example application uses the *init_zycap*, *Config_PR_Bitstream* and the *close_zycap* APIs. Users may also use the *Prefetch_PR_Bitstream* API to buffer the bitstream prior to executing the reconfiguration for better reconfiguration speed.

4.4 Porting the example to other systems

For using ZyCAP on other Zynq boards with the same SD Card image, it is required to alter the following.

- The *BOOT.bin* file has to be rebuilt using the new hardware definition. The required BOOT.bin combines the FSBL corresponding to the new hardware project and the u-boot binary file, which has to be rebuilt from Xilinx U-boot sources (https://github.com/Xilinx/u-boot-xlnx.git). The version in the example design uses the u-boot release tagged xilinx-v14.7 (use *git checkout tag* to use this version). Also see the patches applied to the xilinx-v14.7 version in the *path-to-zycap/linux/bootfiles/u-boot-readme.text*.

- Replace the BOOT.bin, the devicetree binary (devicetree.dtb) and the default PL bitstream (which has to be named config_1.bit) in the boot partition of the SD Card.

- Replace the mode1.bin, mode2.bin in the example directory with the corresponding binary files from the hardware project.

Note: Please ensure that the device-tree has PS elements addressed at the same location as the existing devicetree in the design as this may lead to conflicts. You may want to rebuild the Linux kernel from https://github.com/Digilent/linux-Digilent-Dev.git (tagged xilinx-v2013.4). Also, see the patches in /usr/src/xillinux/kernel-patches (of Linux, after booting in using the image) which have been applied for this version. See http://www.wiki.xilinx.com/Zynq+Linux for step by step instructions.

4.5 Building ZyCAP drivers for a Linux OS

The device drivers and user-level drivers for ZyCAP are available in *path-to-zycap/linux/drivers* folder (within device and userdriver folders). These can be compiled on the device using standard make routine as below.

```
# cd path-to-driver-folder
# make
# make install
# cd path-to-userdriver-folder
# make
# make install
```

It is needed to insert the ZyCAP IP into the devicetree for the drivers to work. Please see the devicetree.dts file at *path-to-zycap/linux/bootfiles* for the format.

Bibliography

[1] K. Vipin, S. Fahmy *et al.*, "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq," *Embedded Systems Letters, IEEE*, vol. 6, no. 3, pp. 41–44, 2014.

[2] S. Shreejith and S. A. Fahmy, "Security Aware Network Controllers for Next Generation Automotive Embedded System," in *Proceedings of the Design Automation Conference (DAC)*, 2015.

[3] S. Shreejith, B. Banarjee, K. Vipin, and S. A. Fahmy, "Dynamic Cognitive Radios on the Xilinx Zynq Hybrid FPGA," in *Proceedings of the International Conference on Cognitive Radio Oriented Wireless Networks (CROWNCOM)*, 2015.

[4] Xilinx Inc, *UG702: Partial Reconfiguration User Guide*, Xilinx Inc., 2012.

[5] ——, *UG909: Vivado Design Suite User Guide : Partial Reconfiguration*, Xilinx Inc., 2014.

[6] "USB Image Tool," http://download.cnet.com/USB-Image-Tool/3000-2242_4-75449768.html.

[7] "Win32 Disk Imager," http://www.softpedia.com/get/CD-DVD-Tools/Data-CD-DVD-Burning/Win32-Disk-Imager.shtml.

[8] "Microsoft .Net Component," http://www.microsoft.com/downloads/details.aspx?familyid=0856eacb-4362-4b0d-8edd-aab15c5e04f5.

[9] Avnet Inc., *HW_UG: ZedBoard Hardware User's Guide*, January 2014.