Getting started with Partial Reconfiguration on UltraZed

Introduction

This document provides a short tutorial of how to create partially reconfigurable (PR) modules and how to launch them at runtime using Xilinx Vivado 18.2 and PetaLinux 18.2. The board we target in this tutorial is the UltraZed (xczu3eg-sfva625-1-i). Furthermore, we assume that the Vivado SDx 18.2 suite and PetaLinux 18.2 are already installed.

The necessary examples, drivers and data files can be found at:

https://www.dropbox.com/sh/qsg5m7jp1sn4saj/AABAzSGOa91K0Kvtlz_0LuRta?dl=0

PetaLinux 18.2 can be downloaded from:

https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html

Vivado can be downloaded from:

https://www.xilinx.com/support/download.html

Generating PR Accelerators

Synthesize the module out-of-context:

We start the process by synthesizing the module RTL code which were generated by the Vivado HLS tool as *an out-of-context module*. In our current flow, the use of VHDL code is essential for compatibility!

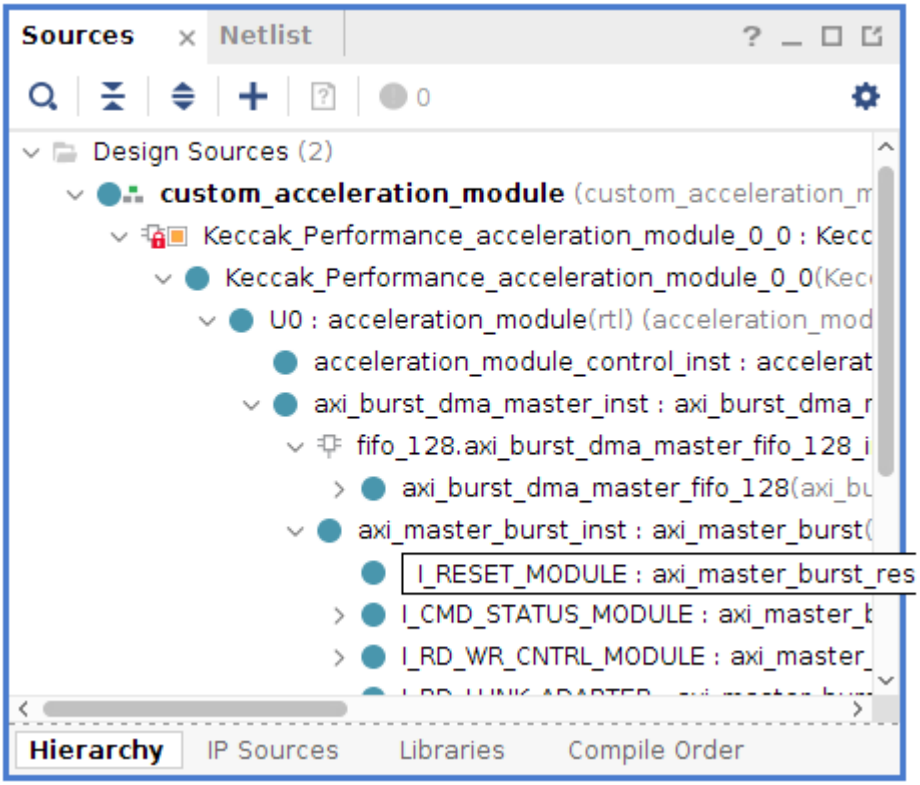
1. Copy the vhdl folder from your Vivado HLS project to the ./Sources folder
2. In Vivado, use the following TCL comments:
    - *read_vhdl* **{list of all files in the vhdl folder}**
    - *synth_design -mode out_of_context -flatten_hierarchy rebuilt -part xczu3eg-sfva625-1-i -top* **{module's top name}**
    - *write_checkpoint -force ./Synth/reconfig_modules/***{module's top name}**
    - *close_design*

Note that in the case of the Keccak module, we need to customize the aforementioned flow further to get that specific module synthesized and compatible to our framework. The following steps describe the customization:

1. AXI interface to each PR region has 1 clock signal signed. Hence, we need to wrap the acceleration_module in the standard AXI interface using a wrapper

"custom_acceleration_module.v" demonstrates this for Keccak module. This would set the AXI and acceleration module to same frequency.

2. Adding sources from IP folder from the original HTV demo project does not import all the sources correctly for *axi_burst_master_inst*. A workaround is to add *"custom_acceleration_module.v"* and then importing the Keccak IP from the block design of HTV project file i.e. *"Keccak_Performance.srcs/sources_1/bd/Keccak_Performance/ip/Keccak_Performance_acceleration_module_0_0"* This would lead to the source hierarchy shown below:



Implement the PR module using blocker templates:

In this step, we are going to physically implement the module based on provided blocker templates. We have templates for modules occupying 1 slot, 2 slots, and 3 slots. The AXI interfaces between the module and the static system is pre-placed and pre-routed in the clock row Y0.
For this tutorial, we are using a 32-bit AXI Lite slave port and a 128-bit AXI full master port.

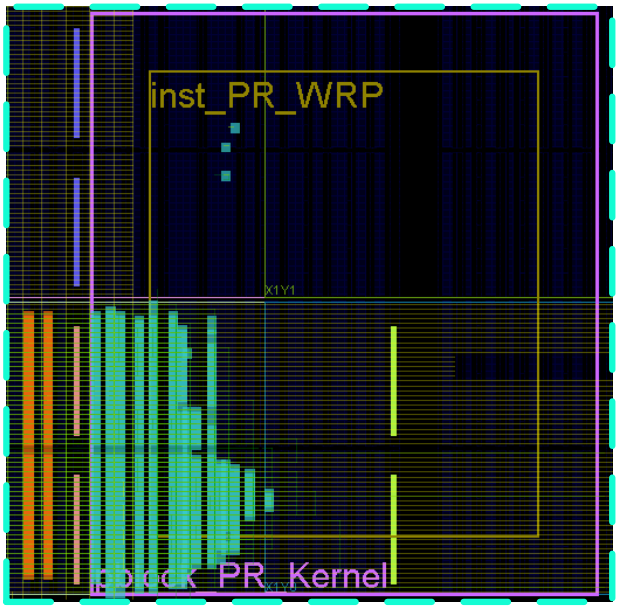
1. Based on the module synthesis report, choose how many slots are needed.
   Available resources per slot can be found in the following table:

Available resources	***Numbers***
CLB LUTs	17760
RAMB36/FIFO	60
RAMB18	120
DSPs	96

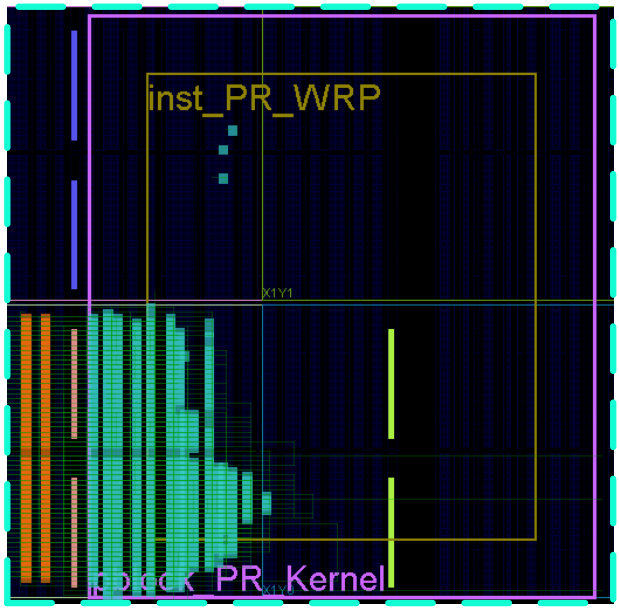
2. According to the necessary resources, choose to run: *pr_module_1_slot.tcl*, *pr_module_2_slots.tcl*, or *pr_module_3_slots.tcl.*
    - Change the top_module to the module name: ***set top_module xx***
    - *source ./pr_module_xx.tcl*

This will run the entire logic synthesis, as well as the physical implementation all the way to a (full) bitstream. The implementation scripts will use blocker macros (provided as DCPs) that constrain the routing of the module in strict bounding boxes.

The physical implementation results are as the following figure.



2-slot module with the blocker

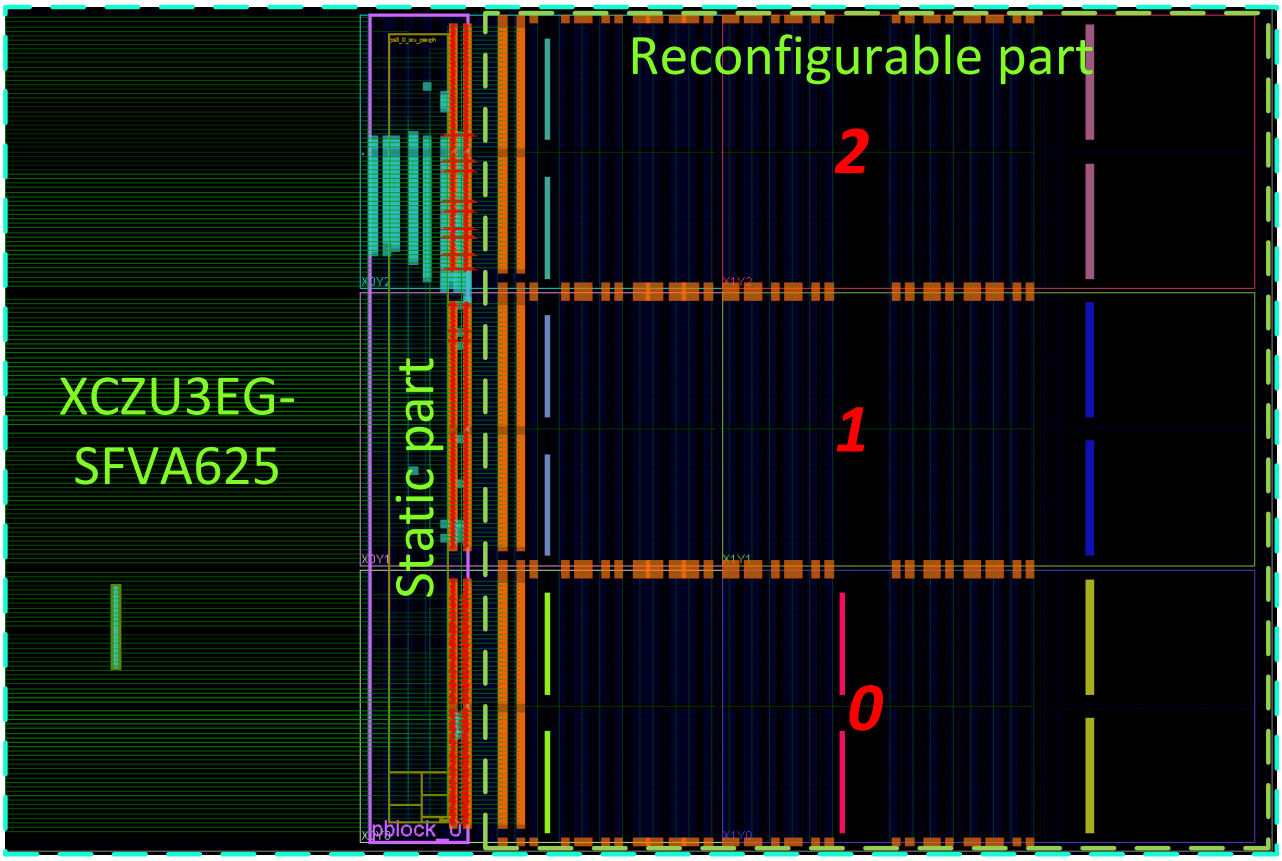


2-slot module without the blocker

Merge the PR module into the given static design bitstream:

Now, we start merging the PR module to the static design at bitstream level.

The static design is as the following figure.



The tool BitMan is used to conduct this step:

1. Merging the PR module which is allocated in the Slot 0 (1 slot) to the static:
    - *bitman_linux -m 21 0 99 59 ./{**module's top name**}_full.bit ./HTV_STC.bit -F ./Merge_{module's top name}_HTV_STC.bit*
2. Merging the PR module which is allocated in the Slot 0-1 (2 slots) to the static:
    - *bitman_linux -m 21 0 99 119 ./{**module's top name**}_full.bit ./HTV_STC.bit -F ./Merge_{module's top name}_HTV_STC.bit*
3. Merging the PR module which is allocated in the Slot 0-2 (3 slots) to the static:
    - *bitman_linux -m 21 0 99 179 ./{**module's top name**}_full.bit ./HTV_STC.bit -F ./Merge_{module's top name}_HTV_STC.bit*

The three commands are essentially the same and only differ in the number of slots that are cut out from the (full) module configuration bitstream that is then merged into the full static bitstream (*HTV_STC.bit*). This process is carried out in a way that will not touch the routing information of the global clock resources.

Create partial bitstreams from the merged full bitstream:

The partial bitstreams can be extracted from the aforementioned merged bitstream by the following BitMan commands:

1. In order to cut out the module and place it in one of the three slots, use one of the following lines. This will generate the corresponding partial bitstreams:
    - *bitman_linux -x 21 0 99 59 ./Merge_{module's top name}_HTV_STC.bit –M 21 0 ./Partial_{module's top name}_Slot_0.bit*
    - *bitman_linux -x 21 0 99 59 ./Merge_{module's top name}_HTV_STC.bit –M 21 60 ./Partial_{module's top name}_Slot_1.bit*
    - *bitman_linux -x 21 0 99 59 ./Merge_{module's top name}_HTV_STC.bit –M 21 120 ./Partial_{module's top name}_Slot_2.bit*
2. To perform the same for placing a two-slots module use:
    - *bitman_linux -x 21 0 99 119 ./Merge_{module's top name}_HTV_STC.bit –M 21 0 ./Partial_{module's top name}_Slot_0_1.bit*
    - *bitman_linux -x 21 0 99 119 ./Merge_{module's top name}_HTV_STC.bit –M 21 60 ./Partial_{module's top name}_Slot_1_2.bit*
3. To perform the same for placing a three-slots module use:
    - *bitman_linux -x 21 0 99 179 ./Merge_{module's top name}_HTV_STC.bit –M 21 0 ./Partial_{module's top name}_Slot_0_1_2.bit*

They are all Xilinx-compatible partial bitstreams. However, in order to use PCAP to load those partial bitstreams onto FPGA at runtime, we need to convert them into another Xilinx format. This step is automatically done by the Xilinx SDK tool (installed with Vivado Design Suite). We follow the following steps:

1. Put the below command into the XSCT Console:
    - *bootgen -image Bitstream.bif -arch zynqmp -o ./{module's top name}_Slot.bin –w*
2. The Bitstream.bif is as the following picture:

```
all:
{
        Partial_Keccak_128_01.bit /* Bitstream file name */
}
```

PetaLinux

Creating and building the PetaLinux project

We are going to create a PetaLinux image for the PR system with all necessary drivers and applications.

1. Source the PetaLinux environment variable to begin with:
    - *source* <**petalinux install dir**>*/settings.sh*
2. Create a project:
    - *petalinux-create --type project --template zynqMP --name* < **project** >
    - *cd* <**project**>
3. Configure the project with the given HDF file:
    - *cp* <**released dir**>*/HDF/HTV_STC_128bit.hdf* .
    - *petalinux-config --get-hw-description*
    - For the purposes of this tutorial, default settings should suffice. Hence, Select Exit and Yes.
    - Use the UltraZed system-user.dtsi into <**project**>/*project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi* for proper device tree generation
4. Create PetaLinux applications/user-space drivers:
    - In this stage, we provide 3 user-space device drivers to control PR decoupler, vector addition, and Keccak accelerators: decoupler.c, vadd.c, and sha3.c.
    - *petalinux-create -t apps --template c --name {app_name} –enable*
    - Replace contents of the file *{app_name}*.c under <**project**>/*project-spec/meta-user/recipes-apps/{app_name}/files* by the respective contents we provided.
5. Build the project:
    - *petalinux-build*
6. Copy the HTV_STC.bit bitstream to the <**project**>/*images/linux* directory
7. Package the BOOT.BIN
    - *petalinux-package --boot --fsbl zynqmp_fsbl.elf --u-boot --pmufw pmufw.elf --force --fpga HTV_STC.bit*
8. Copy BOOT.BIN and image.ub files to the boot partition of the SD card
9. Copy partial bitstreams of the module to the boot partition of the SD card as well

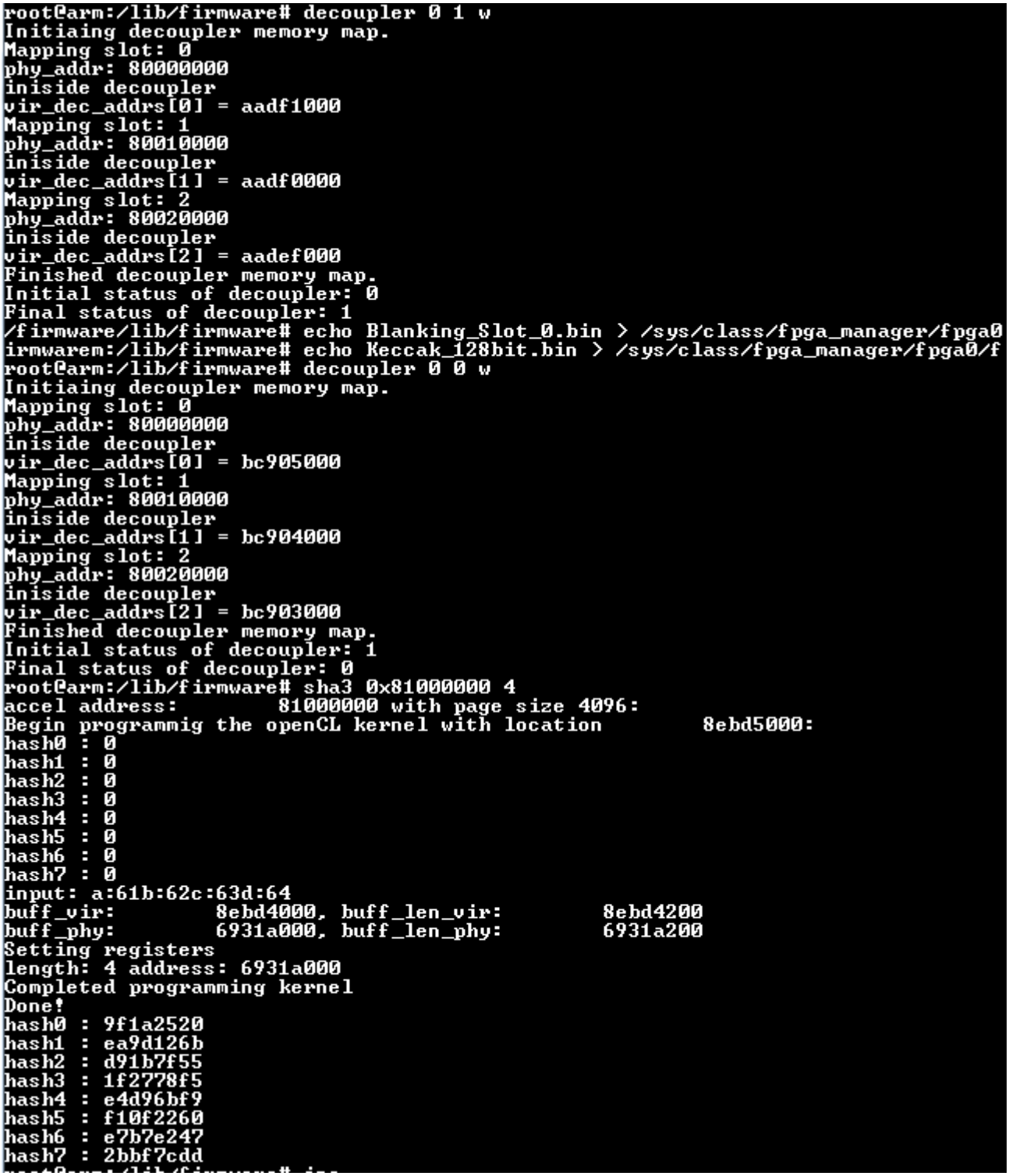
Launch the design on the board

Now we are launching the static design on the board and loading partial bitstream onto the FPGA at runtime and calling a user-space driver to control the accelerator:

1. Make the /lib/firmware directory for partial reconfiguration. As we use the initramfs format, so the /lib/firmware will disappear after rebooting the board:
    - *mkdir /lib/firmware*
    - *cd /lib/firmware/*
2. Mount the SD card and copy partial bitstreams to /lib/firmware (we use the Keccak module as an example!):
    - *mount /dev/mmcblk1p1 /media/*
    - *cp /media/Blanking_Slot_0.bin .*
    - *cp /media/Keccak_128bit.bin .*
3. Set the flag for the PCAP firmware to load partial bitstreams. For more detail info, please find in the link https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841847/Solution+ZynqMP+PL+Programming!
    - *echo 1 > /sys/class/fpga_manager/fpga0/flags*
4. Disable the communication between reconfigurable module and the static system (otherwise the system will hang) by activating the PR decoupler:
    - *decoupler {Slot} {Value} [w/r]*
        i. {Slot}: position of the reconfigurable region [0..2]
        ii. {Value}: 1 to activate the PR decouper (disable the communication), 0 to deactivate the PR decoupler (enable the communication)
        iii. [w/r]: write to set the PR decoupler register, read to get the PR decoupler register
    - *decoupler 0 1 w*
5. Load partial bitstream via PCAP firmware:
    - We load a blanking bitstream to clear the reconfigurable region before loading the actual module bitstream
    - *echo Blanking_Slot_0.bin > /sys/class/fpga_manager/fpga0/firmware*
    - *echo Keccak_128bit.bin > /sys/class/fpga_manager/fpga0/firmware*
6. Enable the communication by deactivating the PR decoupler:
    - *decoupler 0 0 w*
7. call the user-space device driver to control the Keccak module:

- We transfer the 4-byte *"abcd"* input to the Keccak module and get back the result
- *sha3 0x81000000 4*

The result of this example is as the following figure:



Booting Debian/Ubuntu

(This part was written in the previous tutorial but is reproduced here for completeness)

Creating PetaLinux project

petalinux-config –get-hw-description

- Set the Boot-args to be same as in device-tree (DTG Settings → Kernel Bootargs → Auto generated bootargs): console=ttyPS0,115200 earlyprintk uio_pdrv_genirq.of_id=generic-uio clk_ignore_unused root=/dev/mmcblk1p2 rw rootwait sdhci.debug_quirks=64 cpuidle.off=1

- Set uboot to load from SD Card: Image Packaging Configuration > Root filesystem type > SD Card
- Exit and Save configuration

Note, "mmcblk1p2" depends on the partition of the SD card where rootfs resides.

Now its time to setup the device tree to enable ethernet:

Copy the content from *system-user_ubuntu.dtsi* provided into the file located at: *Location: ./project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi*

Make sure nder *"sdhci1"*, the following line is append at the end:
disable-wp;

disable-wp makes sure write permissions are solved for rootfs.

Building the PetaLinux project

Execute "*petalinux-build*" (takes ~5/10 min for the first time).

Package the *BOOT.BIN* and *image.ub* files.

For creating debian rootfs

Create 2 partitions on SD Card with a small partition for BOOT and a large one for rootfs.

Get a debian tar from:
https://rcn-ee.com/rootfs/eewiki/minfs/debian-9.2-minimal-armhf-2017-10-07.tar.xz

Extract contents:
tar xf debian-9.2-minimal-armhf-2017-10-07.tar.xz

sudo tar xfvp ./-*-*-armhf-*/armhf-rootfs-*.tar -C /media/anuj/rootfs/*

Wait for the changes to be written to SD Card

Set the permissions correctly:

sudo chown root:root /media/{user_name}/rootfs/
sudo chmod 755 /media/{user_name}/rootfs/

Copy the following files from images/linux to SD Cards' BOOT partition
BOOT.bin, image.ub, system.dtb, system.bit, *.elf, Image

Copy {app}.c from the providing **<Apps>** directory into SD Card's BOOT partition.

Running and compiling the applications on Ubuntu

Booting the board with newly built image for first time:

- Interrupt the auto-boot
- To write the default environment we've just built: '*env default -f -a*'
- Set the default boot device to the SD Card instead of eMMC: '*env set sd_dev 1*', '*env set sdbootdev 1*'
- Write the environment to QSPI: '*env save*'
- Reboot the board: '*reset*'

Let the board boot normally and login with
username: *debian*
pwd: *temppwd*

We can copy the C files of apps/drivers from the BOOT partition of the SD card as mention in the previous PetaLinux section and build it natively on the board:

- *sudo mount /dev/mmcblk1p1 /media/*
- *cp /media/{app}.c .*
- *gcc –o {app} {app}.c*

Repeat the steps in the previous PetaLinux section. Note that we may need to root permission to write bitstreams to the PCAP firmware. It is better to switch to the root user by the command *"sudo su"*.

Side information links

How to setup mac address in Debian?

https://blog.sleeplessbeastie.eu/2013/01/11/how-to-change-the-mac-address-of-an-ethernet-interface/

How to setup ssh server?

http://www.configserverfirewall.com/debian-linux/install-debian-ssh-server-openssh/

Executing the following command from the host would copy file from host PC to FPGA.

scp <PATH_TO_FILE> debian@<IP_address>:~/

Example:

scp udmabuf.ko debian@130.88.193.80:~/udmabuf

Executing application which require large contiguous physical memory using UDMABUF

Copy the *udmabuf.ko* and *udma_exp.c* to the BOOT parition of SD Card

Boot linux and load the partial bitstream

change to *root* user using: *sudo su*

Copy the driver and application from the BOOT partition to home directory:

- *mount /dev/mmcblk1p1 /media/*
- *cp /media/udmabuf.ko ./*
- *cp /media/udma_exp.c ./*

Load udmabuf driver using the following command:

- *insmod udmabuf.ko udmabuf0=10000000*

"10000000" would allocate 10000000 bytes as contiguous physical memory

---

Compile *udma_exp.c*

- *gcc udma_exp.c -o udma_exp*

Launch the Keccak

- *./udma_exp 0x81000000 <num of bytes to hash>*

Note, the *udma_exp.c* hashes the buffer with ascii char 'a' incremeting as input.