

SYSTEM ON CHIP LABORATORY

December 23, 2017



Contents

1	Introduction to SoC FPGA	3
1.1	SoC FPGA	3
1.2	Workflow	3
1.3	Pulse Width Modulation (PWM) Slave Peripheral for LED control	3
2	Lab Setup	5
2.1	Specifications	5
2.2	Hardware	5
2.3	Gateware	5
2.4	Software	5
3	Lab exercises	6
3.1	Hardware assembly	6
3.2	Gateware development	6
3.2.1	Project Setup	6
3.3	Block Design for SOC	7
3.4	Block Design for FPGA Fabric	7
3.5	Synthesis, Implementation & Static timing analysis	9
4	Software development	11
5	Embedded Linux	12

1 INTRODUCTION TO SoC FPGA

1.1 SoC FPGA

MANU

The System on Chip (SoC) is getting more and more popular, exceeding the capabilities of a simple microcontroller. SoCs are used mostly in smartphones and tablets due to their low power consumption, much shorter wiring and high level of integration.

This laboratory will thus aim to familiarize you with the **All Programmable System on Chip (AP SoC)**. The board (presented in Figure 1) that you are going to use is called Z-TURN, built around the Xilinx Zynq-7010. It contains an FPGA and a dual-core ARM microcontroller, basically it is a **System on Chip**.

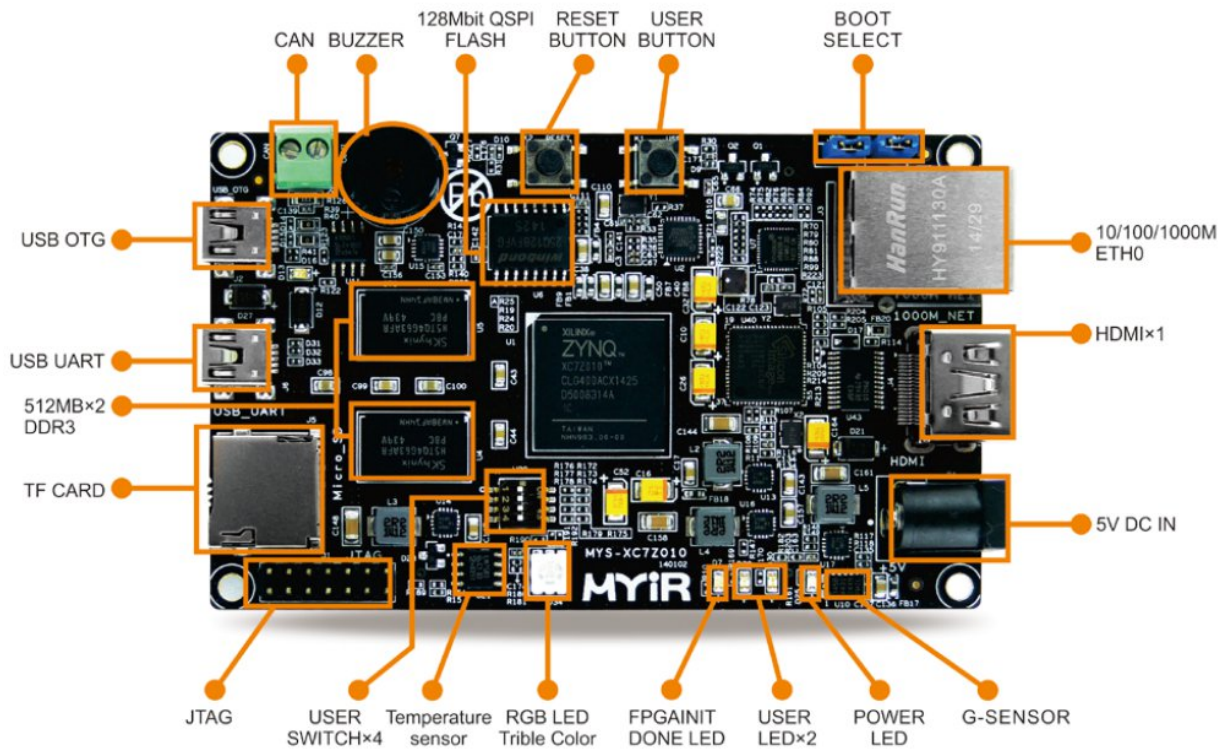


Figure 1: Z-Turn board capabilities

After fulfilling this lab, you are mainly going to learn:

- the workflow of designing an application on FPGA
- the interaction between an FPGA and an ARM microcontroller
- getting an overview of the challenges of using this system

1.2 Workflow

In this laboratory, you will implement the workflow for designing one application used in High Energy Physics.

ADD MANU's WORK

Explain to the students what is PWM and how does it work to control an LED.

1.3 Pulse Width Modulation (PWM) Slave Peripheral for LED control

In Figure 2, you can see a basic PWM waveform.

The FPGA runs at a clock frequency (for this application, we will choose 50 MHz). We can create a PWM waveform from this clock frequency, by counting a number of clock cycles. There are two important parameters that characterize the PWM waveform: the **duty cycle** and the **PWM period**. The PWM period is set by "waiting/counting" a number of clock cycles, while the duty cycle tells us, in one period, how many clock cycles the PWM signal is low.

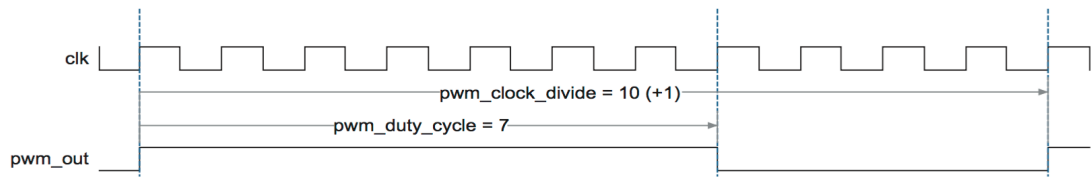


Figure 2: PWM waveform (Source: Application Note 333, Altera)

Now let's see how we are going to use these two simple parameters to make an RGB LED change its color.

RGB LED

In the case of RGB LED, we need 3 PWMs signals (one per each color), such that the brightness of each of the three LEDs can be controlled independently.

How do we choose the PWM period? The driving frequency of the PWM should be fast enough to avoid the flicker effect. A normal human being sees this effect until up to 100 to 150 Hz, so a higher frequency should be better to avoid this effect. For this exercise, we will use a frequency of 1.5 kHz. Let's make some calculations.

The main clock frequency of the FPGA is 50 MHz. We want to use a 1.5 kHz frequency. How much do we need to count?

$$counter = \frac{50000000}{15000} = 30000 \quad (1)$$

How do we choose the PWM duty cycle? The brightness of the LEDs is controlled by the duty cycle. This is up to you to do experiments. You can choose basically any duty cycle in the range of 0% to 100%.

2 LAB SETUP

2.1 Specifications

MANU

HEP TDAQ system

2.2 Hardware

MANU

Front-End board: (ALS-GEVB) Board with sensor Back-End board (MYIR Z-turn) Board with SoC FPGA

2.3 Gateware

MANU

Vivado is used in this part. The gateware is divided in SoC and Fpga fabric parts

SoC

Vivado block schematics

FPGA fabric

HDL language

2.4 Software

MANU

Xilinx SDK is used in this part

Stand-Alone

C++

Embedded Linux

Very complex, provided by MYIR

3 LAB EXERCISES

3.1 Hardware assembly

Connect the board and the laptop.

- Ensure that your board is powered on by connecting the Mini-USB to USB cable from computer to the USB_UART mini-usb socket. This connectivity will also allow us transmit and receive data through serial communication.
- Connect the programmer on the JTAG port. Make sure the connector matches the pinout from Figure 3.

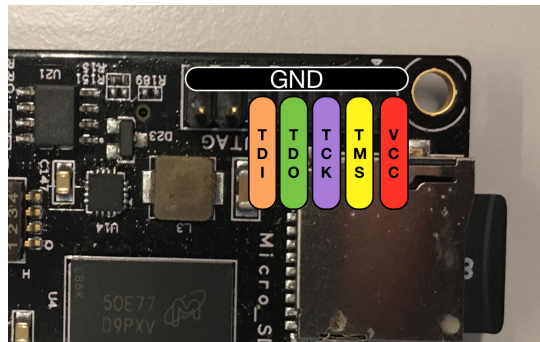


Figure 3: JTAG Pinout

3.2 Gateware development

3.2.1 Project Setup

1. Open an Ubuntu terminal(CTRL-T) and type the following to launch Vivado Design Suite.

```
vivado &
```

2. Create a new project and call it **detector**.

Click **Next** and tick **RTL Project**. Next, you will be asked to add sources to your design. For the moment, we will keep the project empty. However, we need to specify **VERILOG** for both *Target Language* and *Simulator Language*. Press **Next**, and you will be asked about Existing IP and Constrains files. Keep them both empty.

3. Choose the hardware platform where we will run the applications.

The development board is entitled MYS-7Z010-C-S Z-turn (Figure 4). Press **Next** and then **Finish**.

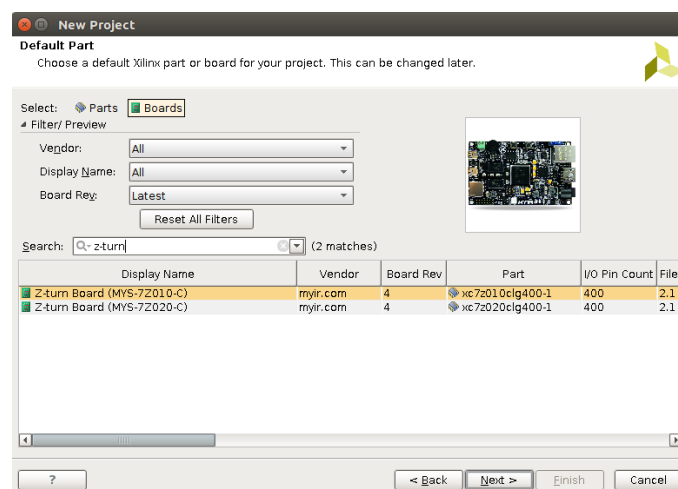


Figure 4: Hardware Selection

The starting page of the project should like in Figure 5. In this project, we will implement the necessary files to control an RGB LED.

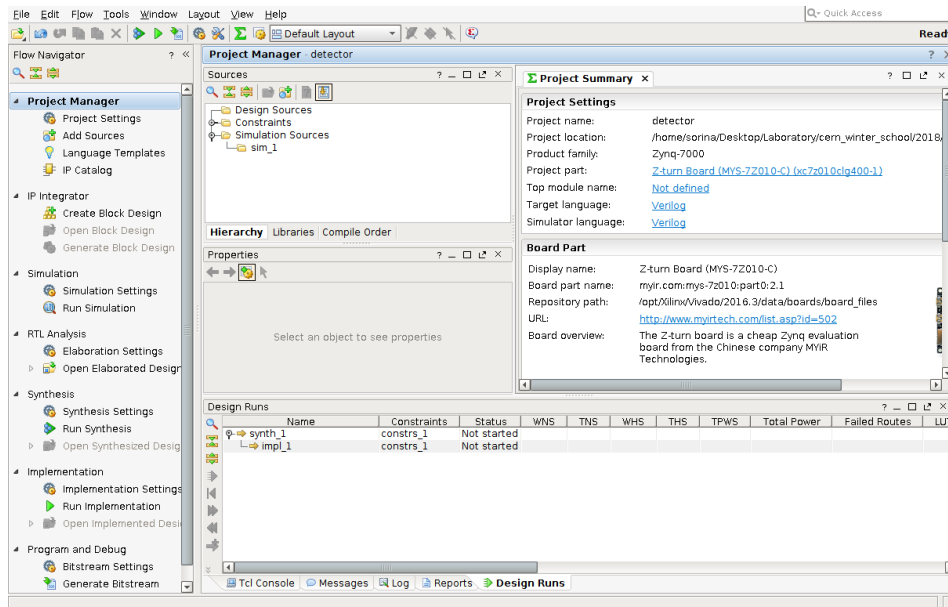


Figure 5: Initial Project Structure

3.3 Block Design for SOC

1. Create the Block Design

On the left of the window, in the **Flow Navigator**, choose **Create Block Design** and call it **detector**.

2. Define SoC with the wizard

Press **Add IP** and choose **Zynq7 Processing System**. Click [Run Block Automation](#) to auto-configure it. Double click on it to open the wizard.

3. Discuss with your tutor the structure of the System on Chip. We will leave most of the settings as default, apart from several.

- Click on PS-PL Configuration -> HP Slave Axi Interface and De-select S AXI HPO INTERFACE (Figure 6 a))
- Click on Peripherals I/O Pins and leave ticked only the peripherals which we are using (Figure 6 b):
 - UART1 - for communication
 - I2C1 - for interface with the Light Sensor
- Clock Configuration. Select only FCLK_CLK0 and change its value to 50 MHz. (Figure 6 c))

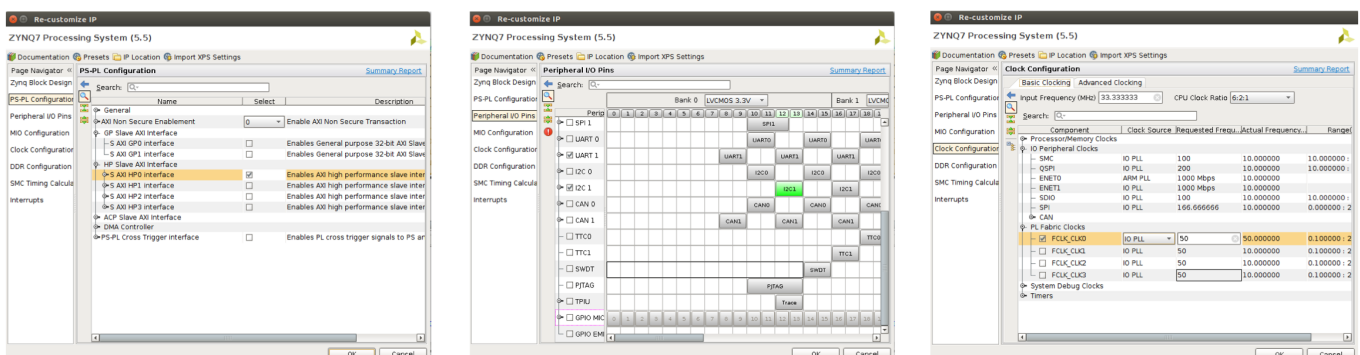




Figure 6: Zynq Wizard a)PS-PL Configuration b) Peripherals c)Timing

3.4 Block Design for FPGA Fabric

1. Add the PWM IP module to the project

The PWM IP was already implemented for you, but you need to add it to the library. Click on the **IP settings** icon . Choose **Repository Manager** from the upper tabs. Click on + symbol and add the content found at the following path relative to your main folder:

`/ip/pwm_verilog_1.0`

Now the PWM folder is added to your project so you can choose it from the IP Library. Press **Add IP**  and type "pwm".

2. Let's dig into how the PWM Peripheral was designed

Right click on the **pwm_verilog_v1 IP** and select **Edit in IP Packager**. Another project (Figure 7) that contains the files written by the tutors for this IP will open. In the Source part, you will see an hierarchical design in Verilog. The top level is called `pwm_verilog_v1_0.v` and contains an instantiation of the `pwm_verilog_v1_0_S00_AXI.vhd` where the logic is implemented. If you open `pwm_verilog_v1_0_S00_AXI.v`, you will see the implementation of PWM code outlined by the following comments:

```
-- User to add parameters/ports/logic here
...
-- User parameters/ports/logic ends
```



Discuss the code with the tutor.

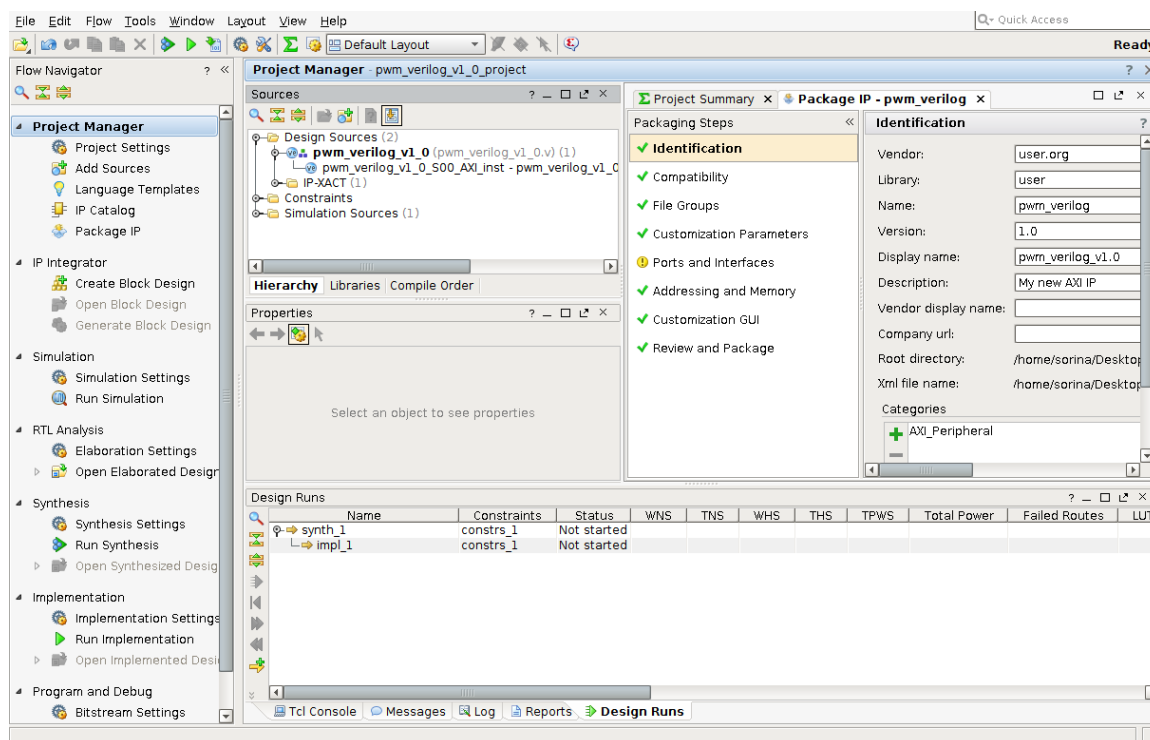



Figure 7: PWM IP Project Structure

3. Connect the PWM controller HDL module to the SoC

We need to create PWM outputs, in order to connect them to our SoC. Click on each PWM output and press CTRL-K or **Create Port** from the Menu. (Figure 8) Do so for all the 3 ports.

Right now, both systems should look like in Figure 9.

Last step is to choose [Run Connection Automation](#), let the settings as default, press OK and let the Designer Assistance to do this work on your behalf. The window should now look similar to Figure ???. However, the Design Assistant is placing the blocks in a not-very-organized way. For a better view, we advise you to click on the Regenerate Design button  in the right Menu. Looks better, right?

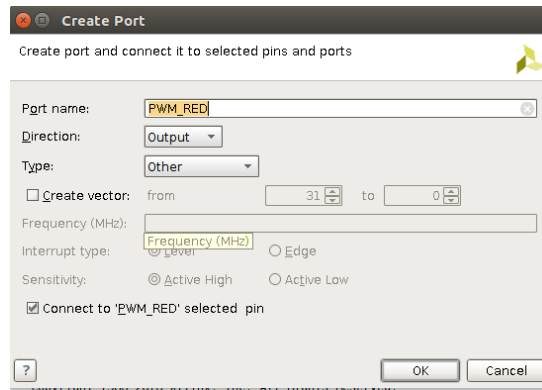


Figure 8: Port Creation for each PWM output

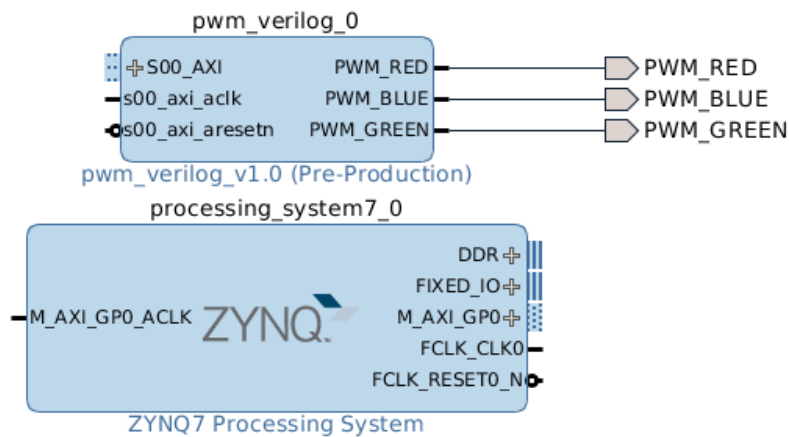


Figure 9: Both IPs



Discuss the block diagram with your tutor.

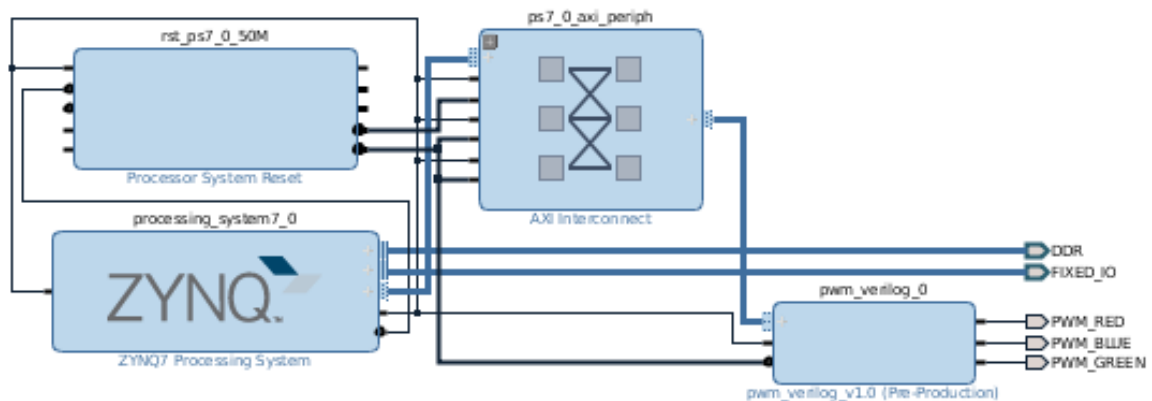


Figure 10: Block Diagram of our System

3.5 Synthesis, Implementation & Static timing analysis



1. Create Wrapper

From the block design, we need to create the Verilog code. Therefore, in the Project Sources - Design Sources, you will right click on the **detector.bd** and select *Create HDL Wrapper*. Let the default option and press OK. Now the VHDL Wrapper should have been created. Verify that the signals PWM_RED, PWM_BLUE and PWM_GREEN appear instantiated.

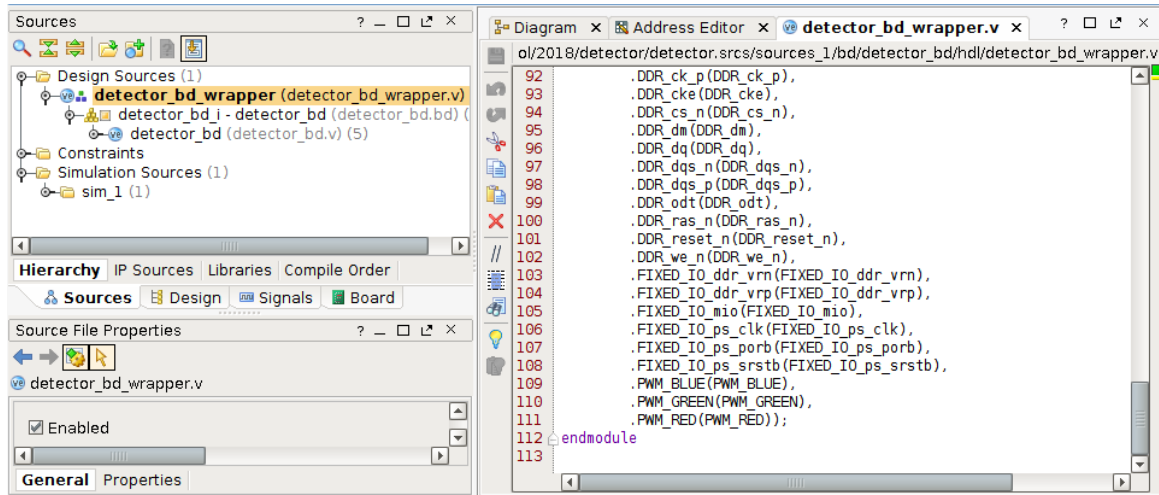


Figure 11: Wrapper with the PWM outputs instantiated

2. Create Constraint File

The Constraint File is used in the implementation phase. After the synthesis will be run, and the available top-level nets are "known", they will be matched to the "real" pins.

Right click on the Constraints directory and click **Add Source** (Add or Create Constraints). The constraints file are present in `/constraints/` directory. Open it and check it together with the tutor

3. Generate the bitstream

The last step in FPGA design is the generation of the bitstream. To generate the bitstream, look into the Flow Manager on the left of the screen and press **Generate Bitstream** (Figure 12). You will need to wait a bit longer until the bitstream is created. Check for errors in the process and solve them.

4. Export the hardware

Now, that the bitstream was successfully created, we need to export the hardware such as we can use it together with the ARM dual-core microcontroller.

To export the hardware, press **File > Edit > Export > Export Hardware**. Tick *Include bitstream* like in Figure 12.

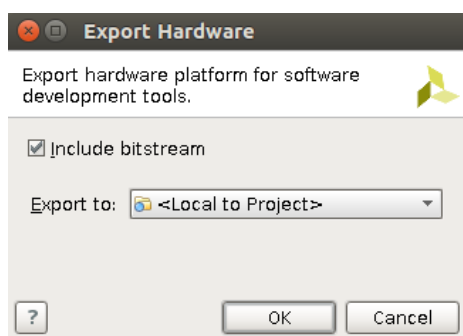


Figure 12: a) Export Hardware Window b) Generate Bitstream in Flow Manager

4 SOFTWARE DEVELOPMENT

1. Create the BSP

In Vivado, press **File > Launch SDK**. Now, the hardware shall be automatically imported in Xilinx SDK and you can create a software application (Figure ??).

From now on, we will create the link between the FPGA and the dual-core ARM microcontroller. In our case, the ARM microcontroller is used to read the detector sensor and send signals to the FPGA fabric that further controls an RGB LED.

2. Create a New Application Project

Select **File -> New -> Application Project**. Select Empty Application and call it **detector_controller**.

3. Add the Cpp file

Some files were already written to make your life easier, such as the driver to read the light detector through I2C.

In the **Project Explorer**, go to folder *detector_controller/src* and add the source files available in **sw** folder.

4. Modify the Cpp file

Open the **main.c**. Read the instructions and solve the simple state machine.

```
/*
 * i2c_getDataDetector(IIC_DEVICE_ID) returns the value
 * read by the sensor as an integer
 *
 * Xil_Out32(COLOR, brightness) sets the color of the LED
 * from 0 to MAX_BRIGHTNESS
 *
 * xil_printf("%d", value) prints an integer using UART
 */

/* Write your code here
 * Implement a state machine that turns on the RED LED
 * when the value exceeded certain threshold
 *
 * Otherwise, make the BLUE LED's brightness follow the sensor
 * readout (e.g. brightness = x*sensor_value
 *
 * Print the sensor_value
 */
```

5. Execute the code on the SoC FPGA

Ask your tutor for help in running the application and programming both the FPGA and the microcontroller.

6. Verify the results



5 EMBEDDED LINUX

1. Insert the SD card with the file system on the slot of the Z-turn
2. Find Jumper 1 and Jumper 2 (JP1 and JP2) and make sure they are in the following order:

JP1 OFF

JP2 ON

3. Execute the program

To run the webserver, you need to type in the emulator the following:

```
bokeh serve --host 192.168.2.2:5006 read_event.py
```

Then, in your host computer, open a webbrowser and type the IP, as seen in Figure 14.

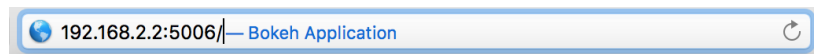


Figure 13: Host address

4. Check the webserver and verify the results

Then, the plot will be displayed interactively! Have fun!

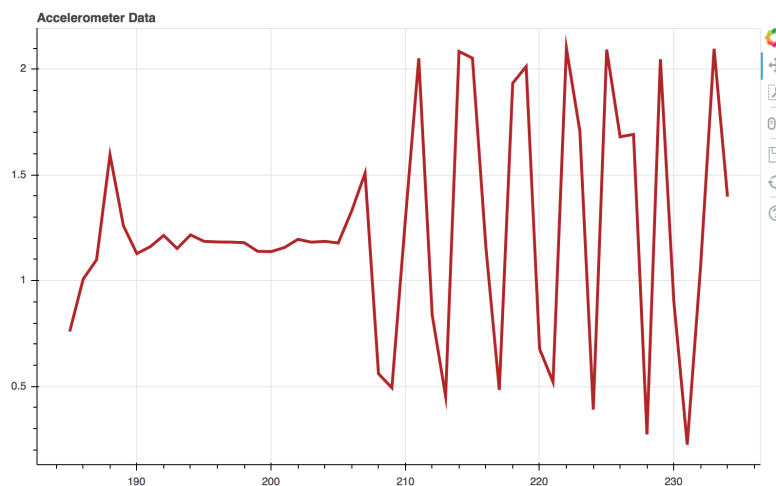


Figure 14: Host address

Now you can play with the setup and discuss about it with the tutor