The International School of Trigger and Data Acquisition (ISOTDAQ), London, 2019

System on Chip (SoC) Field-Programmable Gate Array (FPGA) Laboratory

(Version: 1.1)

Tutor:
Johannes Wuethrich (CERN) (johannes.wuethrich@cern.ch)
Lab Developers:
Manoel Barros-Marin (CERN) (manoel.barros.marin@cern.ch),
Elena-Sorina Lupu (Caltech) (eslupu@caltech.edu)

Contents

1 INTRODUCTION

The aim of the **SoC FPGA laboratory** at the **International School Of Trigger Data AcQuisition (ISOTDAQ)** is to provide students a brief overview of the different stages in the SoC FPGA design workflow and the knowledge to determine when a SoC FPGA is the most appropriate core for their project. After the completion of the lab, the students should be able to understand the interaction between the two main building blocks of a SoC FPGA (FPGA fabric and Hard Processor (HCPU)) and assess the challenges of implementing such a system.

1.1 SoC FPGA

Processors (CPU) and **Field Programmable Gate Arrays (FPGAs)** are the hardworking cores of most Trigger DAQ systems. Integrating the high-level management functionality of processors and the stringent, real-time operations, extreme data processing, or interface functions of an FPGA into a single device forms a more powerful embedded computing platform. **System on Chip (SoC) FPGA** devices integrate both processor and FPGA architectures into a single chip. Consequently, they provide higher integration, lower power, smaller board size and higher bandwidth communication between the processor and FPGA. They also include a rich set of peripherals, on-chip memory, an FPGA-style logic array and high speed transceivers. As a result of the previously mentioned qualities, coupled with a wide range in terms of cost and performance, the SoC FPGA devices are becoming more and more popular among digital electronics and software designers.

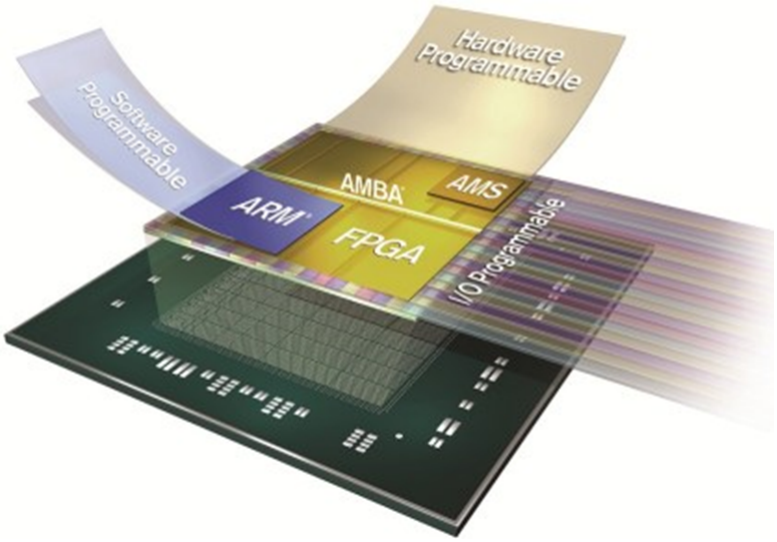

Figure 1: Model of SoC FPGA

1.2 SoC FPGA workflow

A typical SoC FPGA workflow is illustrated in Figure 2.

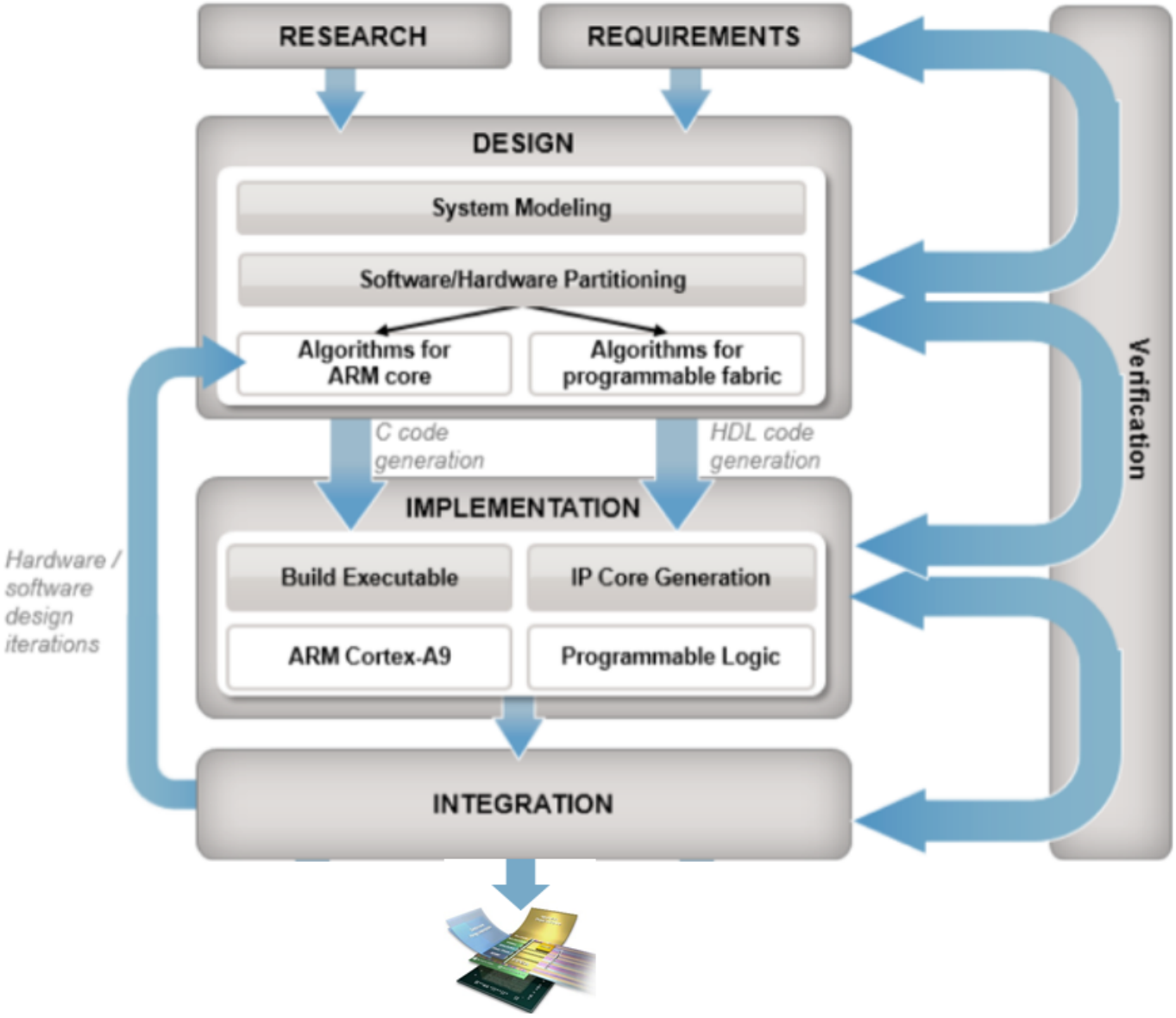

Figure 2: SoC FPGA workflow

2 LAB SETUP

2.1 Specifications

With the help of your team, you are going to implement an **emulator of a typical TDAQ system in High-Energy Physics (HEP) experiments**. For simplicity reasons, this TDAQ system may be divided in two groups. On one side, the **front-end (FE)** electronics, placed "close to the experiment". On the other side, the **back-end (BE)** electronics, placed "close to the control room". In this case, the communication between the FE and the BE is performed through a bidirectional serial link over copper cable. In a typical HEP experiment, the analogue signal from the sensor is filtered and shaped by the analogue FE electronics. This conditioned signal is digitized by an Analog to Digital Converter (ADC). Once in the digital domain, the raw data from the ADC is evaluated by the digital FE electronics (please note that this evaluation may be done in the analogue domain instead). When an event of interest occurs (e.g. particle crosses the sensor), the value of the raw data surpasses a preset threshold, triggering its transmission to the BE through the serial link. In the BE, the serial data is deserialized, processed, stored in memory and sent to the farm of computers though Ethernet by a CPU. Once in the farm of computers, this data may be post-processed, analysed and plotted by the users. In this lab, a devkit featuring an ambient light sensor (ALS-GEV) plays the role of the HEP experiment sensor and its related analogue and digital FE electronics. The communication between the FE and the BE is performed through a bidirectional Inter-Integrated Circuit (I2C) link over copper cable. A SoC FPGA devkit (MYIR Z-turn) is used as BE electronics. This SoC FPGA devkit communicates through Ethernet with a laptop running Python scripts over Linux, which emulates the farm of computers and the user's computer. The block diagram of the emulated HEP experiment is depicted in Figure 3.

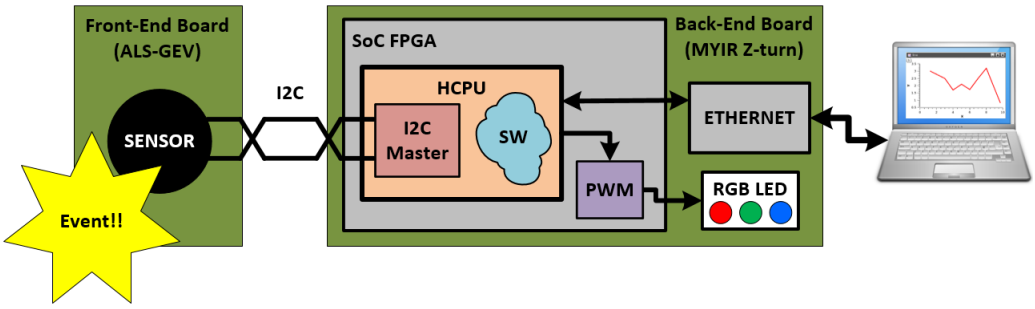

Figure 3: HEP experiment emulation block diagram

2.2 Hardware

2.2.1 ALS-GEVB

The **Front-End electronics** of our emulated HEP experiment is based on the **Ambient Light Sensor (ALS) Shield Evaluation Board (ALS-GEVB)**, ilustrated in Figure 4). This board is the devkit of the NOA1305, an ambient light sensor (ALS) designed for handheld applications. The NOA1305 integrates a 16-bit ADC, a 2-wire I2C digital interface, internal clock oscillator and a power down mode. The built in dynamic dark current compensation and precision calibration capability coupled with InfraRed (IR) and 50-60 Hz flicker rejection enables highly accurate measurements from very low light levels to full sunlight. The device can support simple count equals lux readings in interrupt-driven or polling modes. The NOA1305 employs proprietary CMOS image sensing technology from ON Semiconductor to provide large signal to noise ratio (SNR) and wide dynamic range (DR) over the entire operating temperature range. The optical filter used with this chip provides a light response similar to that of the human eye.

Figure 4: Front-End board (ALS-GEVB)

2.2.2 MYIR Z-turn Board

The **Back-End electronics** of our emulated HEP experiment is based on the **MYIR Z-turn Board**, which is a low-cost and high-performance System On Chip FPGA devkit. This board is based on the Xilinx Zynq-7000 family, featuring integrated dual-core ARM Cortex-A9 processor with Xilinx 7-series Field Programmable Gate Array (FPGA) logic.

The MYIR Z-turn Board takes full features of the Zynq-7010 (or Zynq-7010) SoC FPGA, it has 1GB DDR3 SDRAM and 16MB QSPI Flash on board and a set of rich peripherals including USB-to-UART, Mini USB OTG, 10/100/1000Mbps Ethernet, CAN, HDMI, TF, JTAG, Buzzer, G-sensor and Temperature sensor. On the rear of the board, there are two 1.27mm pitch 80-pin SMT female connectors to allow the availability of 96 / 106 user I/O and configurable as up to 39 LVDS pairs I/O.

The Z-turn Board is capable of running Linux operating system. MYIR has provided Linux 3.15.0 SDK, the kernel and many drivers are in source code. An image of the MYIR Z-turn Board, highlighting its main components is illustrated in Figure 5).

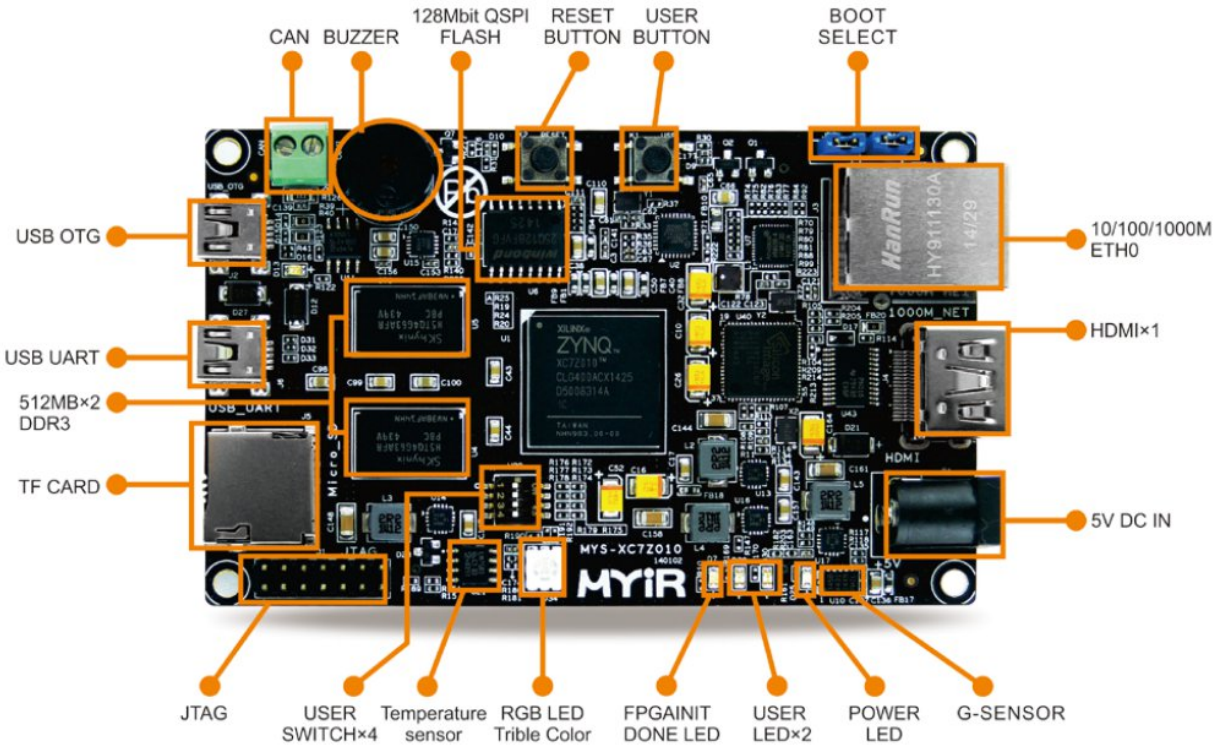

Figure 5: Z-Turn board capabilities

2.2.3 Laptop

The **farm of computers and user's laptop** of our emulated HEP experiment is based on a **laptop running Linux** (Ubuntu in this case). Here the different Electronic Design Automation (EDA) tools for implementing the SoC FPGA project and software scripts for data analysis and plotting are executed. An example of laptop running Linux is illustrated in Figure 6.

Figure 6: Laptop running Linux

2.2.4 Miscellaneous

The rest of the hardware components required for this lab are the following:

- **Custom cable** (ALS-GEVB power & ALS-GEVB/MYIR Z-turn I2C communication) (see Figure 7)

Figure 7: Custom cable

- **Mini-USB to USB cable** (MYIR Z-turn power & MYIR Z-turn/Laptop UART communication) (see Figure 8)

Figure 8: Mini-USB to USB cable

- **RJ45 CAT5e cable** (MYIR Z-turn/Laptop Ethernet communication) (see Figure 9)

Figure 9: RJ45 CAT5e cable (Ethernet cable)

- **Xilinx Platform Cable USB II** (MYIR Z-turn Zynq JTAG programming) (see Figure 10)

Figure 10: Xilinx Platform Cable USB II

2.3 GateWare

A typical SoC FPGA GateWare (GW) is composed by two main parts. On one hand, the SoC, with the HCPU and its peripherals (e.g. I2C, timers). On the other hand, FPGA fabric and its hard blocks (e.g. BRAMs, Multi-Gigabit Transceivers (MGT)).
For this lab, you have to implement the GW for MYIR Z-turn. As previously mentioned, the MYIR Z-turn features a Xilinx Zynq SoC FPGA (please note that there are other vendors in the market such as Altera, Microsemi, etc.). For that reason, the Xilinx EDA tool for FPGA and SoC FPGA (Vivado) is used throughout this lab.

2.3.1 SoC

The typical approach for implementing the SoC part is through **schematics and wizards**. This facilitates the configuration of such a complex, but well defined, architecture. An example of SoC FPGA schematic and wizard are illustrated in Figure 11.

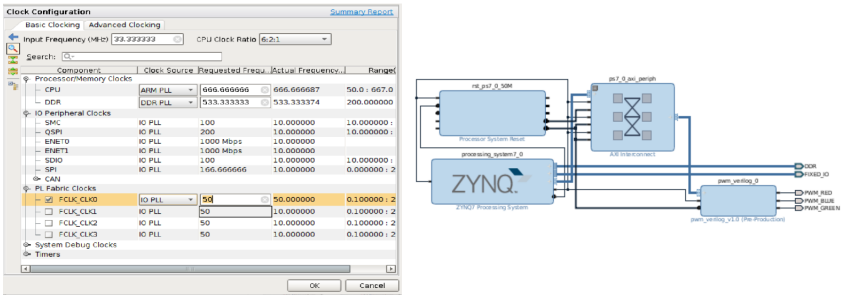

Figure 11: SoC FPGA wizard (left) & schematic (right) example

The SoC part of the GW for our emulated HEP experiment requires a HCPU with an I2C master for communicating with the ALS-GEVB and a UART for communicating with the Laptops (an Ethernet interface for communicating with the laptop will be added at the end of the lab).

2.3.2 FPGA fabric

The modules on the FPGA fabric are usually implemented through **Hardware Description Language (HDL)**, such as SystemVerilog or VHDL. This allows to exploit the versatility of the FPGA fabric. In some specific cases, it may bee interesting to implement these modules using other techniques (e.g. schematics, high-level synthesis), but HDL is the most common by far. An example of HDL code (SystemVerilog in this case) is illustrated in Figure 12.

```
9   module up_counter    (
10  out     ,  // Output of the counter
11  enable  ,  // enable for counter
12  clk     ,  // clock Input
13  reset      // reset Input
14  );
15  //----------Define Ports-------------
16  //----------Output Ports-------------
17  output [7:0] out;
18  //-----------Input Ports-------------
19  input enable, clk, reset;
```

Figure 12: HDL code example

The FPGA fabric side of the GW for our emulated HEP experiment will feature a **Pulse Width Modulation (PWM) block** for controlling an on-board **RGB LED**. For the details of the PWM module, please refer to Appendix A.

2.3.3 SoC/FPGA fabric interface

The most common approach for interfacing the modules of the FPGA fabric with the SoC part is to generate a **schematic wrapper** for the HDL module and connect it to the HCPU as any other SoC peripheral (this is the approach used in this lab). Please note that the other way around would be possible too, using a HDL wrapper for the SoC part and add it to the rest of the HDL code.

2.4 Software

The HCPU of the SoC FPGA requires software to execute for performing the tasks assigned to it. There are two main approaches when implementing software for a SoC. This software can be either **stand-alone** or executed over an **Embedded Operating System (EOS)**, such as embedded Linux. It is important to mention that either approach you chose for your project, it is necessary to export the SoC FPGA GW to the Software Development Kit (SDK) in order to generate the required software drivers. As previously mentioned, the SoC FPGA of our emulated HEP experiment is base on Xilinx Zynq. For that reason, a dedicated SDK from this vendor (Xilinx SDK), based on Eclipse, is used for throughout the lab.

2.4.1 Stand-alone

In the stand-alone approach, user software scripts are directly executed on the HCPU, just having the software drivers between the hardware and the software scripts. This approach is very useful when dealing with **single-threaded** and **real time** systems because simplifies the implementation and facilitates time determinism. The different tiers of the stand-alone software approach are illustrated in Figure 13

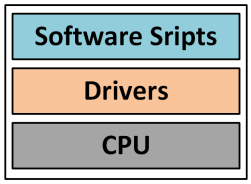

Figure 13: Different tiers of the stand-alone software approach

Before start writing the operational software, it is very important to verify the integrity of the the different hardware components. The stand-alone software approach comes in very handy for that. As first step in the software development you will write a code in python for reading the data from the FE board. When an event of interest is detected (the value o the raw data crosses the predefined threshold), the software will change the colour of an on-board RGB LED.

2.4.2 Embedded Operating System

The use of an embedded operating system (EOS) is required (or highly recommended) when implementing systems **running several software processes in parallel**. In these cases, the integrated arbitration capabilities of the EOS will handle the execution of these processes without requiring interaction from the developer, although at the cost of a more complex software implementation. Moreover, the Real-Time Operating Systems (RTOS), a specific type of EOS, are also capable of executing the processes with a deterministic latency. The different tiers of the EOS approach are illustrated in Figure 14.

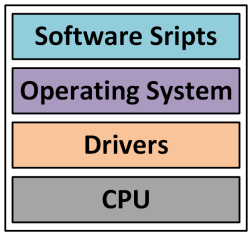

Figure 14: Different tiers of the EOS approach

Once the hardware components of our emulated HEP experiment have been tested, it is time to implement the operational software. In this case, we need a way to read the data from the Front-End while board transferring data through Ethernet to the computers farm (just one laptop in this case ;b). For that reason you need to add an operating system and dedicated software scripts to accomplish this tasks. Besides it is also necessary the software scripts for analysing and plotting the data on the user's laptop. Most likely you are thinking how are you going to implement all that in such a limited amount of time... do not worry. You will get some help from the tutor.

3 LAB EXERCISES

3.1 Hardware assembly

The hardware connection of the SoC FPGA lab is illustrated in Figure 15.

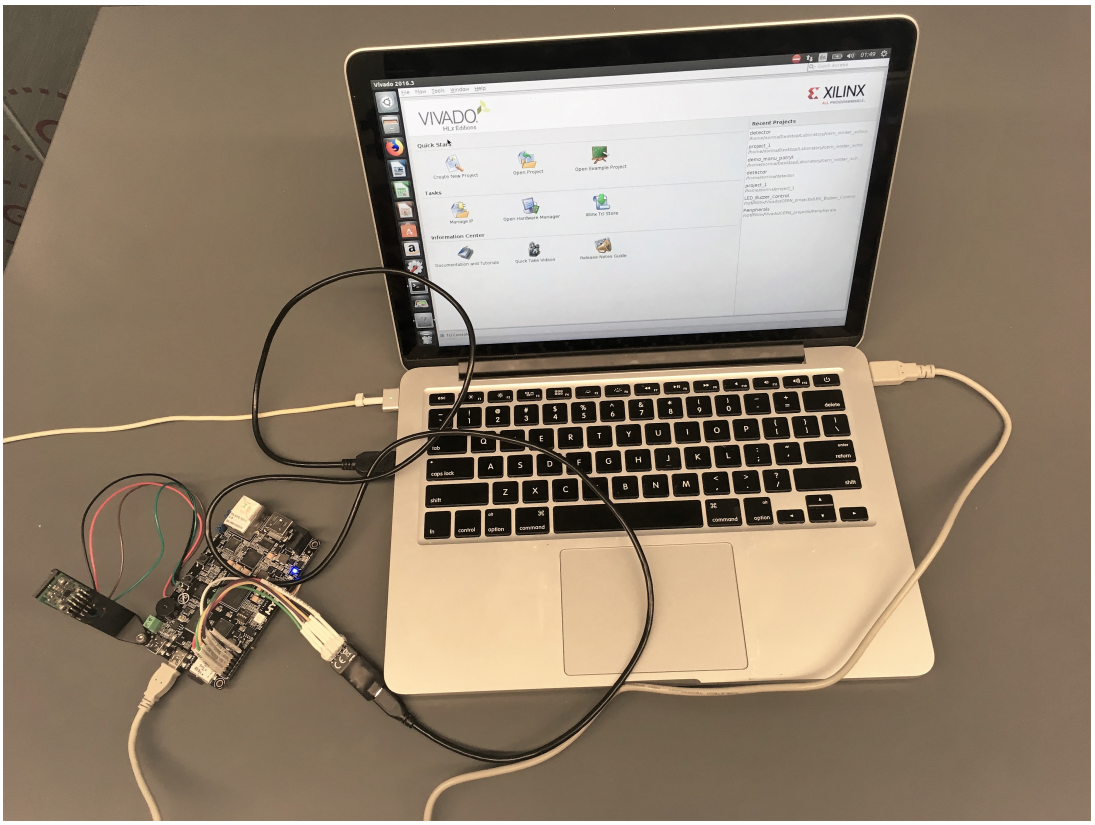

Figure 15: SoC FPGA lab hardware

- 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 (See Figure 16).

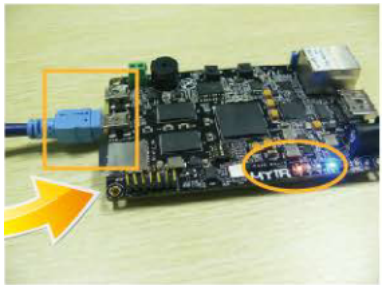

Figure 16: USB_UART Mini-USB cable

- Connect the programmer on the JTAG port. Make sure the connector matches the pinout from Figure 17.

3.2 GateWare development

The GateWare for this lab is developed using **Vivado**, the vendor specific EDA software from Xilinx.

3.2.1 Project setup

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

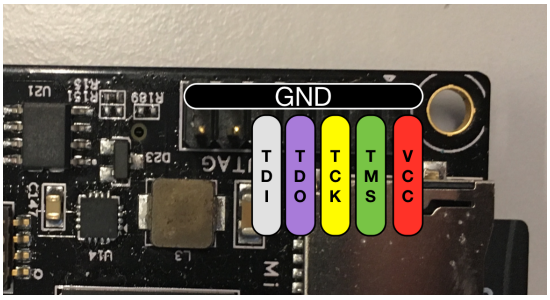

Figure 17: JTAG pinout

```
vivado &
```

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

   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 (see Figure 18). Press **Next** and then **Finish**.

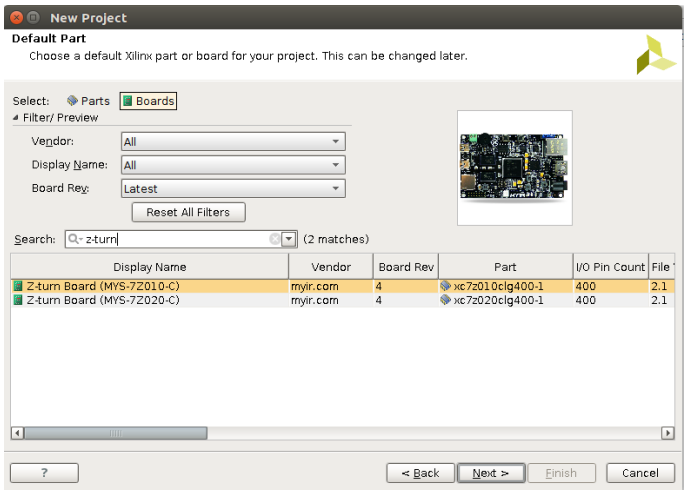

Figure 18: Hardware Selection

The starting page of the project should be like in Figure 19.

3.2.2 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 (see Figure 20 (left))
   - Click on Peripherals I/O Pins and leave ticked only the peripherals which we are using (see Figure 20 (centre)):

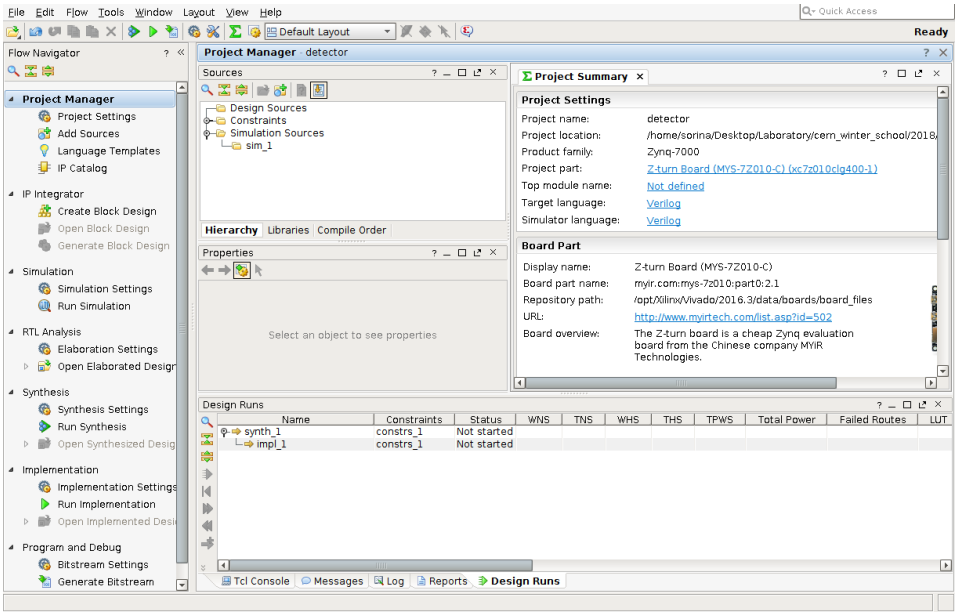

Figure 19: Initial Project Structure

- UART1 - for communication with the laptop
- I2C1 - for interface with the Light Sensor

- Clock Configuration. Select only FCLK_CLK0 and change its value to 50 MHz. (see Figure 20 (right))

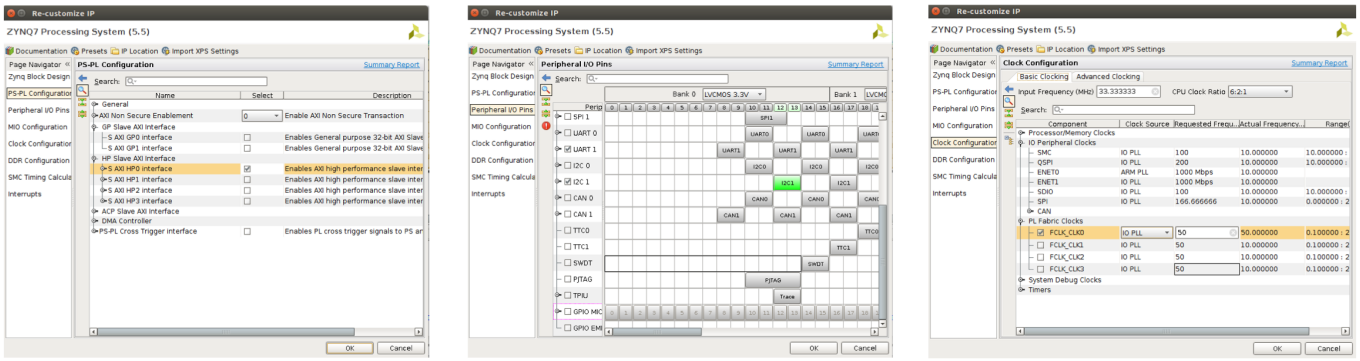

Figure 20: Zynq Wizard (left) PS-PL Configuration (centre) Peripherals Timing (right)

3.2.3 Block design for FPGA Fabric

1. Add the PWM IP module to the project

   Please see the details of the PWM module in Appendix A

   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 (see Figure 21) 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.

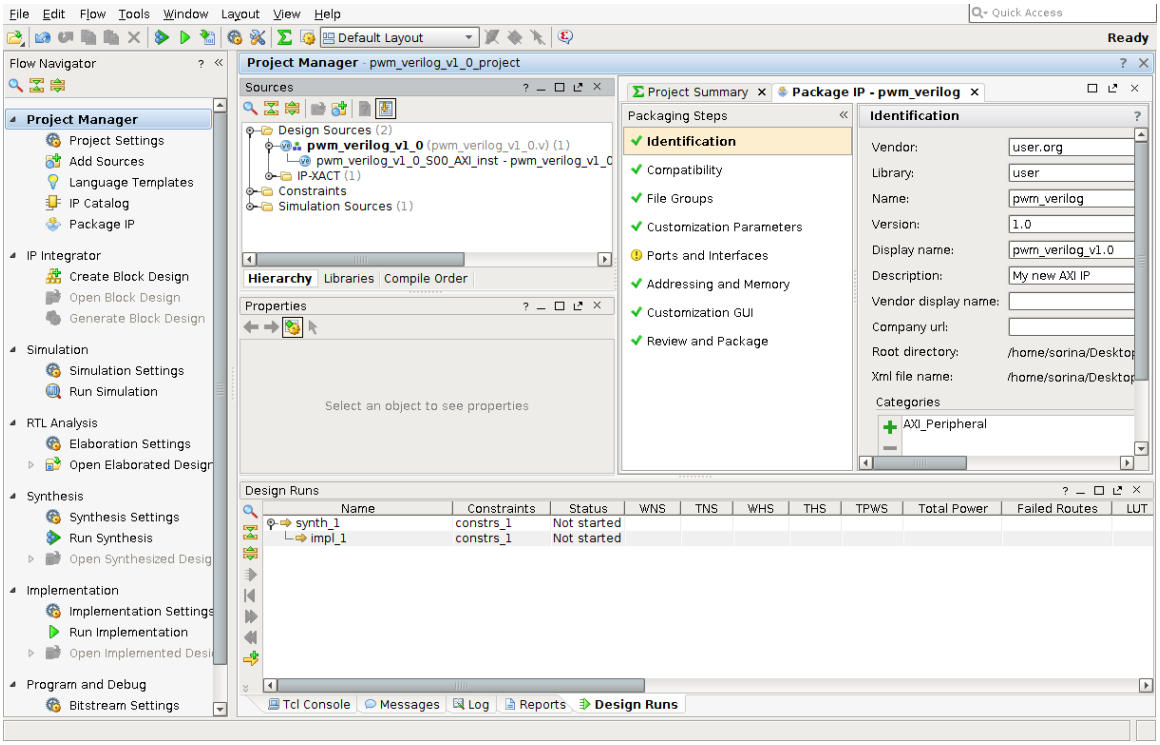

Figure 21: PWM IP Project Structure

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 (see Figure 22). Do so for all the 3 ports.

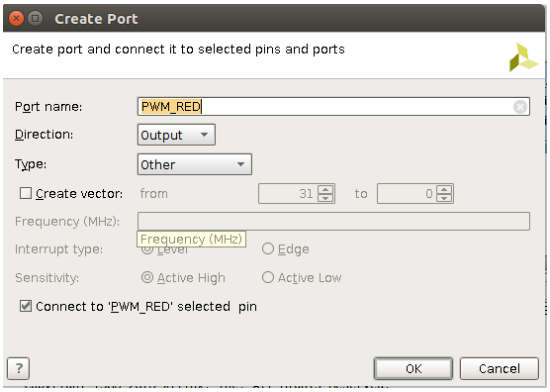

Figure 22: Port Creation for each PWM output

Right now, both systems should look like in Figure 23.

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 24. However, the Design Assistant is placing the blocks in a non-very-organised way. For a better view, we advise you to click on the Regenerate Design button in the right Menu. Looks better, right?

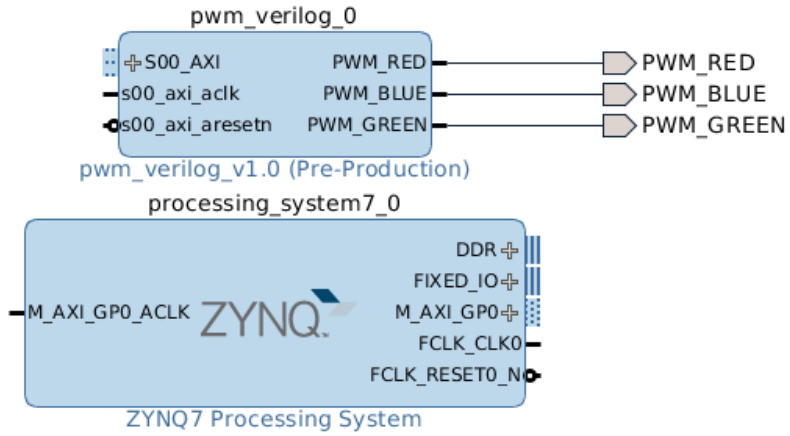

Figure 23: Both IPs

Discuss the block diagram with your tutor.

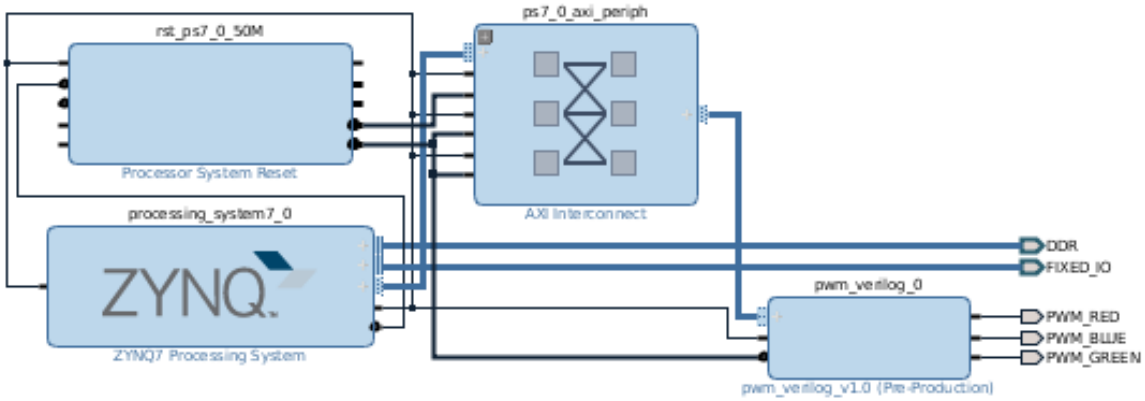

Figure 24: Block Diagram of our System

4. 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 (see Figure 25).

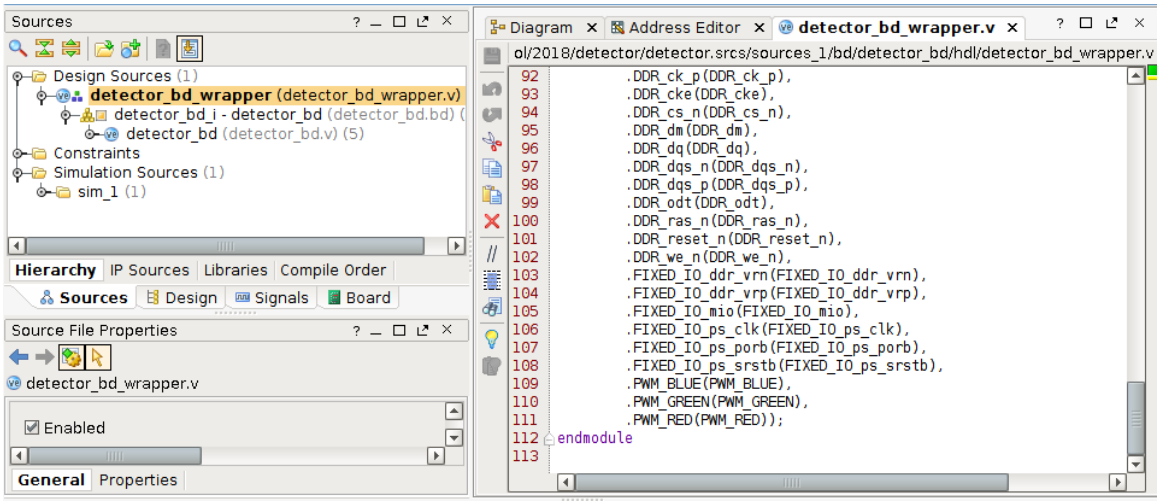

Figure 25: Wrapper with the PWM outputs instantiated

3.2.4 Synthesis, implementation & static timing analysis

Discuss with your tutor the differences between Synthesis, Implementation and Bitstream generation for an FPGA.

1. 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

2. 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 27)**. You will need to wait a bit longer until the bitstream is created. Check for errors in the process and solve them.

3. Static timing analysis

   Please do not forget to check the Static Timing Analysis report and verify that all constraints have been met (see Figure 26)

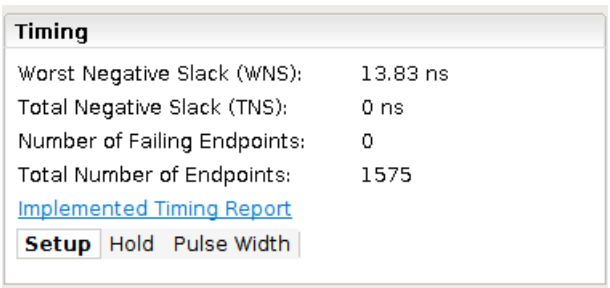

Figure 26: Static Timing Analysis report

3.2.5 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 27 (left).

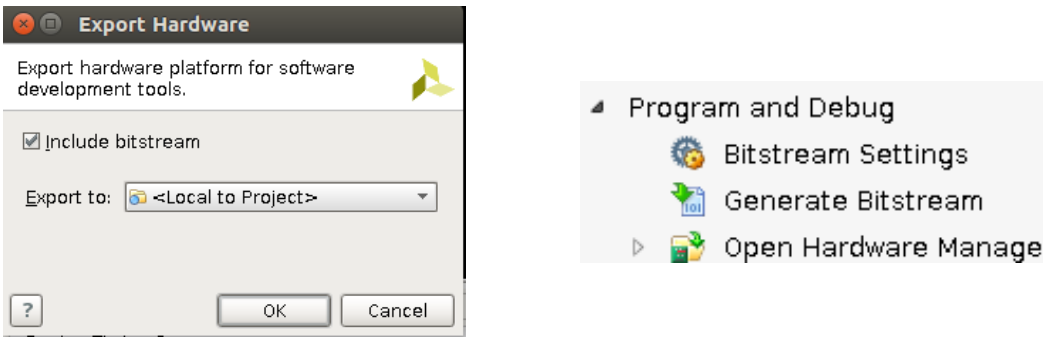

Figure 27: Export Hardware Window (left) Generate Bitstream in Flow Manager (right)

3.3 Software development

The Software for this lab is developed using **Xilinx SDK**, the vendor specific EDA software from Xilinx.

3.3.1 Stand-alone

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.

   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 colour 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 micro-controller.

6. Verify the results

3.3.2 Embedded Operating System

The implementation of an EOS is a complex task. For that reason, SoC FPGA vendors provide pre-implemented EOS (typically Embedded Linux) which just need to be loaded on the SoC FPGA. In this lab, you are going to use the Embedded Ubuntu Linux provided by MYIR for their devkit.

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 for loading the embedded Linux from the SD card:

   ```
   JP1 OFF
   JP2 ON
   ```

3. Execute the program

   To run the web server, 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 web browser and type the IP, as seen in Figure 28.

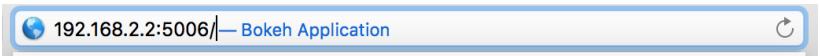

Figure 28: Host address

4. Check the webserver and verify the results

Then, the plot will be displayed interactively (see Figure 29). Have fun!!
Now you can play with the setup and discuss about it with the tutor.

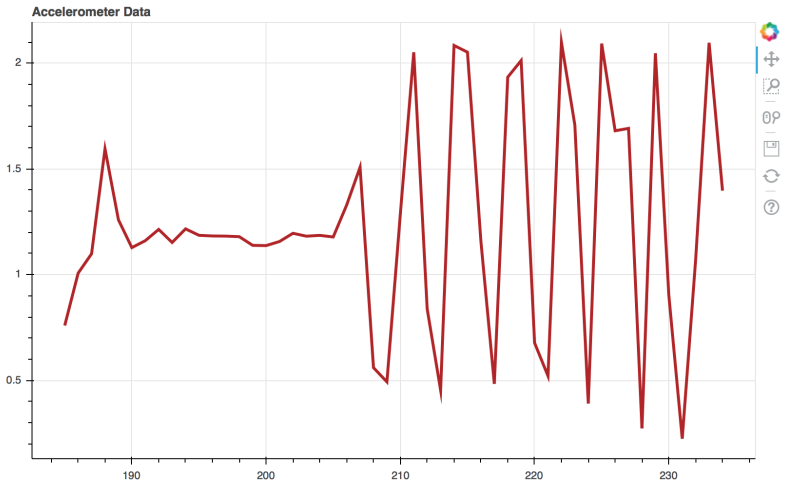

Figure 29: Interactive plot

Appendices

A Pulse Width Modulation (PWM) slave peripheral for LED control

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 characterise 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. In Figure 30, you can see a basic PWM waveform.

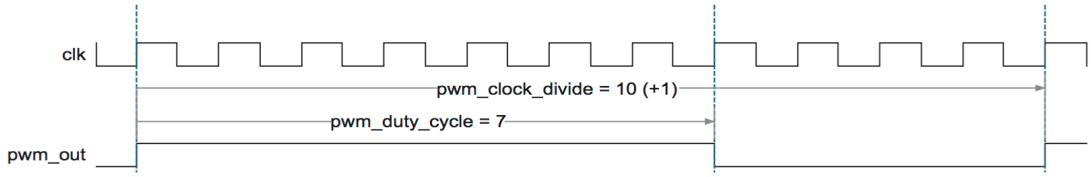

Figure 30: PWM waveform (Source: Application Note 333, Altera)

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

RGB LED

In the case of RGB LED, we need 3 PWMs signals (one per each colour), 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?

$$counter = \frac{50000000}{15000} = 30000 \tag{1}$$

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%.

B Acknowledges

- Markus Joos (CERN) & other organisers of ISOTDAQ
- Ton Damen (NIKHEF)
- Peter Jansweijer (NIKHEF)
- The other members of the ISOTDAQ SoC FPGA Lab Team:
  - Elena-Sorina Lupu (EPFL/Caltech)
  - Andrea Borga (NIKHEF)
  - Manoel Barros Marin (CERN)