The International School of Trigger and Data Acquisition 2017

SYSTEM ON CHIP LABORATORY

February 1, 2017

Contents

1 OVERVIEW

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.

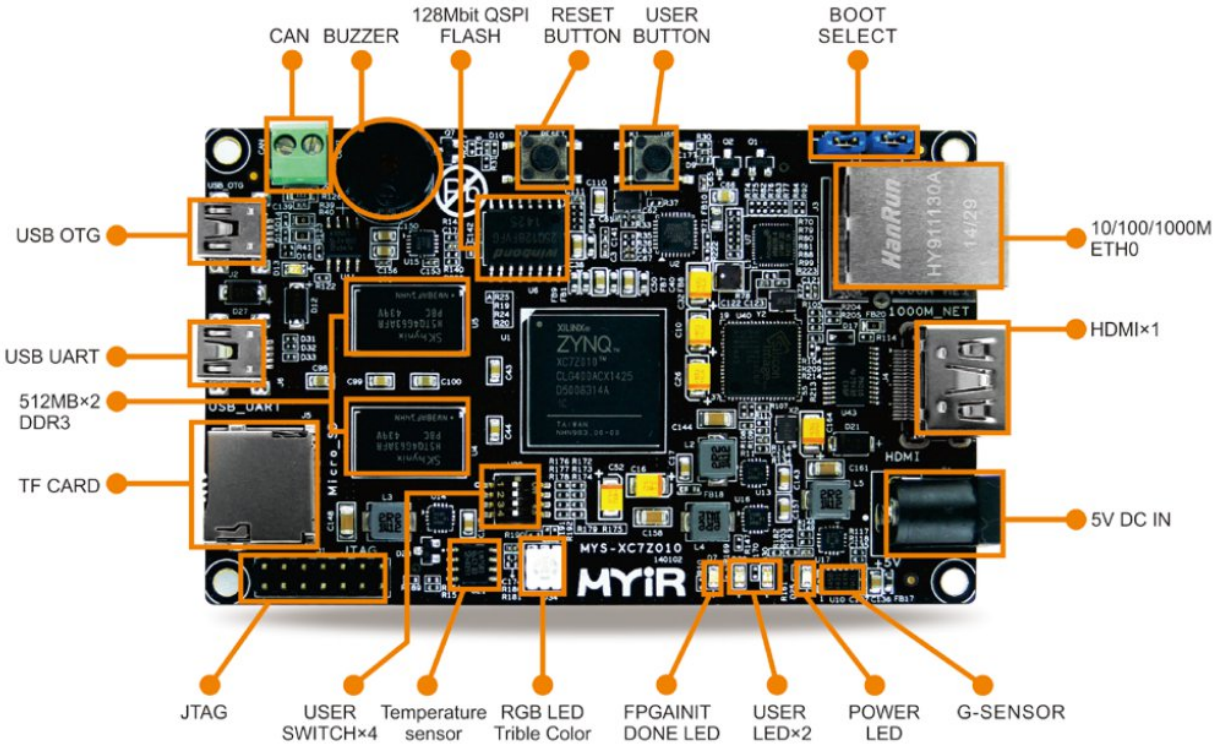

Figure 1: Z-Turn board capabilities

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
- running and interacting with an operating system on the ARM microcontroller
- writing Python scripts for communicating with various pheripherals from the board
- getting an overview of the challenges of using this system

2 APPLICATIONS

In this laboratory, you will implement the workflow for designing two applications.

PROJECT 1. Controller of the LED and the buzzer developed on FPGA

This part will make use of the FPGA and a microcontroller, as well as the interaction between the two. On the FPGA side, we will develop a controller, for interacting with the on-board buzzer and the RGB LED, while the MCU will take care of the sequencial logic of the application. Specifically, the application shall generate a number between 1 and 3 and will control the LED and the buzzer accordingly:

- 1 - LED RED - ON; BUZZER - ON
- 2 - LED GREEN - ON; BUZZER - OFF
- 3 - LED BLUE - ON; BUZZER - OFF

PROJECT 2. Accelerations displayed on a Webserver

The second part of the laboratory consists of reading accelerometer data from the ADXL accelerometer available on the board. This data will be further displayed on a webserver running on the device. When the acceleration will exceed a certain threashold (the system will feel hypergravity), the RED LED and the buzzer will be turned on, similar to a warning system.

2.1 Getting started

Vivado Design Suite, along with the **Xilinx Software Development Kit (XSDK)** are the main applications used in this laboratory. The first one will be mainly used for the hardware development(talking to the FPGA), while XSDK will be used for embedded C development (talking to the ARM processor). The Vivado Design Suite has two important modes: *project mode* and *non-project mode*. In project mode, Vivado tool will automatically manage the design flow and directory structure. As you will become more advanced, the non-project mode will allow you to automatize your design flow using Tcl-language commands and scripts.

2.1.1 System on Chip Design Flow in Vivado Design Suite

Vivado design flow is based on Advanced eXtensible Interface (AXI) standard protocol which provides great flexibility and customization for each of your applications. There are three important concepts related to the AXI interface that you need to know: the AXI Master, the AXI Interconnect and the AXI Slave. Their connectivity can be seen in the next diagram.

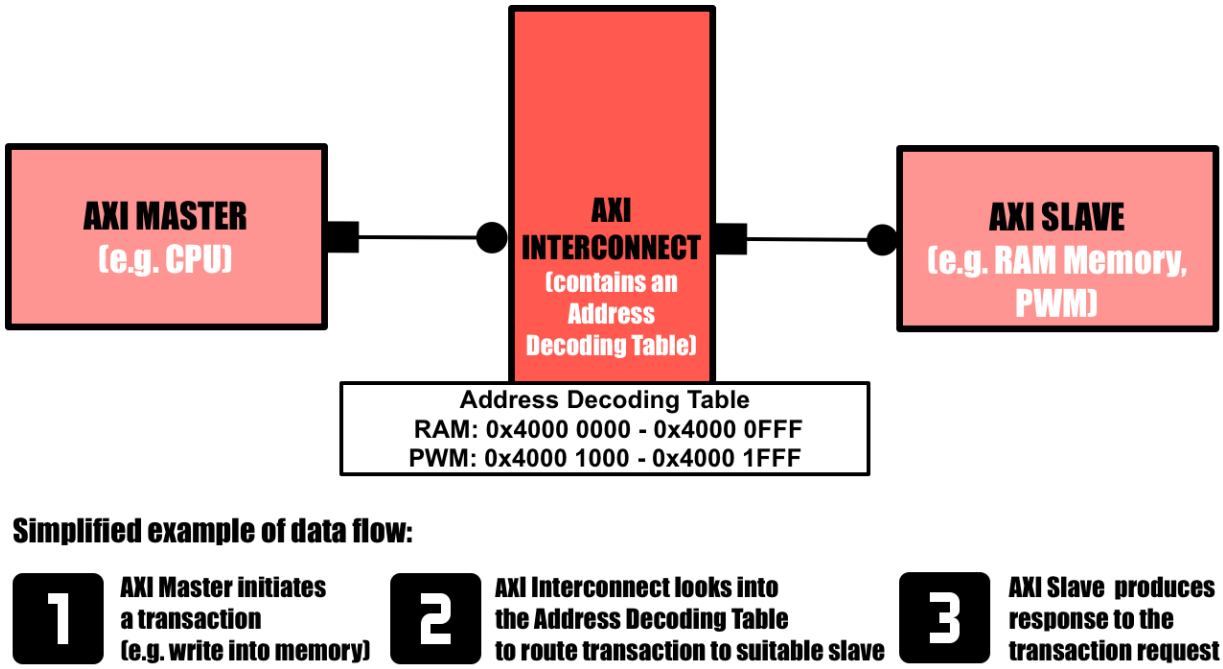

Figure 2: AXI Architecture

As we have talked about customization per application, you can use two types of advanced eXtensible interfaces (AXIs): AXI Stream and AXI Memory Mapped (MM). Basically, a simple application such as computation of some stream of data can make use of AXI Stream, while a more complex one (e.g. writing on a memory) might need access to addresses (so AXI MM comes in place). Their diagrams can be seen in Figure 3. **In our case, we will use an AXI Memory Mapped architecture, and PWM (Pulse Width Modulation) slave module to control LEDs and a buzzer.**

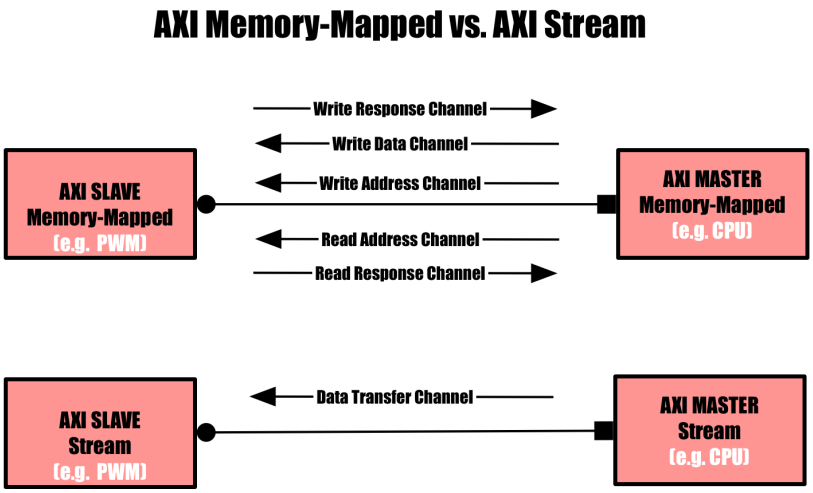

Figure 3: AXI Memory Mapped and AXI Stream arhitecture

2.1.2 Pulse Width Modulation (PWM) Slave Peripheral

INTRODUCTION

In Figure 4, you can see a basic PWM waveform. 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 .

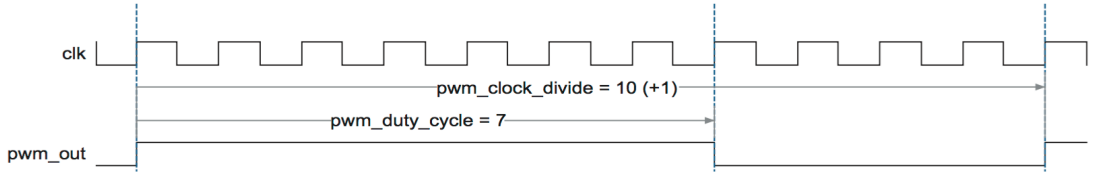

Figure 4: PWM waveform (Source: Application Note 333, Altera)

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

- **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.

  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%.

- **BUZZER**

  Previously, for the LED, we kept constant the frequency and varied the duty cycle. Now, for the buzzer, it will be the opposite: a constant duty cycle and a varying frequency.

  The buzzer is based on the reverse of piezoelectricity which means that if you produce a voltage, the piezoelectric material from the buzzer gets distorted. When you stop producing that voltage, it gets back to its original shape. When producing this voltage fast enough (like in the PWM waveform), the buzzer will start to oscillate and "sing". This is the reason why we are currently controlling the frequency of PWM, and not the duty cycle. In this case, the duty cycle can be interpreted as the volume of the buzzer.

  How do we choose the PWM frequency? The frequency range that the human ear is able to hear is between 20 Hz and 20.000Hz such as the PWM frequency for our buzzer will also fall in this range.

How do we choose the PWM duty cycle? Maximum sound is achieved with a 50% duty cycle. In this exercise, we suggest a duty cycle of 10% such as you will not bother your neighbors when testing your application, but you are free to experiment with other values as well.

PWM SLAVE LOGIC

The PWM waveform can be generated either using the ARM processor and a Timer, or implemented on the FPGA. In this exercise, we will use this slave interface implemented in VHDL (VHSIC Hardware Description Language) language, directly on the fabric.

In Figure 5, the block diagram of the slave peripheral is showed. The explanation of each output/input as well as the internal registers is presented in Table 1. Read it quickly now, but refer to it when you will start implementing the design.

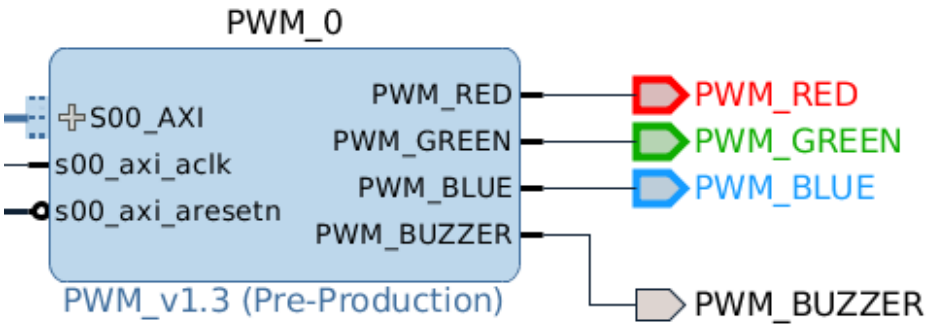

Figure 5: PWM Slave IP

Signal/register	**Type**	**Description**
PWM_RED PWM_BLUE PWM_GREEN PWM_BUZZER	std_logic (1-bit)	Signal set 1 or 0 to create the PWM waveform to control the LEDs and Buzzer
counter_pwm	integer (16-bit)	Register that increments up to PWM_LEDS_PERIOD_COUNTER in order to set the PWM period, and reset to 0 once its value equals that period
PWM_LEDs_PERIOD_COUNTER	integer (16-bit)	Parameter that can be set from the User Interface to choose the period of the PWM. This value is calculated using equation (1).
counter_pwm_buzzer	integer(16-bit)	Register similar to {counter_pwm}
slv_reg0 slv_reg1 slv_reg2	std_logic_vector (32-bit)	Local signals memory mapped to the address space, and used to set the duty cycle of the PWM waveform for red, blue and green color.
slv_reg3	std_logic_vector (32-bit)	Local signal memory mapped to the address space, and used to set the PWM period of the PWM waveform for the buzzer
S00_AXI	set of registers	This interface pin is connected to the AXI_Interconnect which makes the link to the AXI Master, as explained conceptually above.
s00_axi_aclk	std_logic (1-bit)	Synchronized the logic peripheral. Currently, it is connected to the Master Clock.
s00_axi_aresetn	std_logic (1-bit)	Resets the peripheral. Connected to the reset of the Master reset which is mapped to the reset button on the Z-Turn board.

Table 1: Register definition for the PWM Slave

2.1.3 Implementation

In order to start our implementation, the next steps should be followed:

1. Open an Ubuntu terminal and type

   ```
   vivado &
   ```

   Vivado Design Suite is launched and a Getting Started Page is displayed.

2. Create a new project and set the project name to **Peripherals**.

   Select **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 **VHDL** for both *Target Language* and *Simulator Language*. Press **Next**, and you will be asked about Existing IP and Constrains files. Keep them both empty again.

   Now, we need to choose the platform where we implement our design. The development board is entitled MYS-7Z010-C-S Z-turn (Figure 6). Press **Next** and **Finish**. Now, the project will initialize.

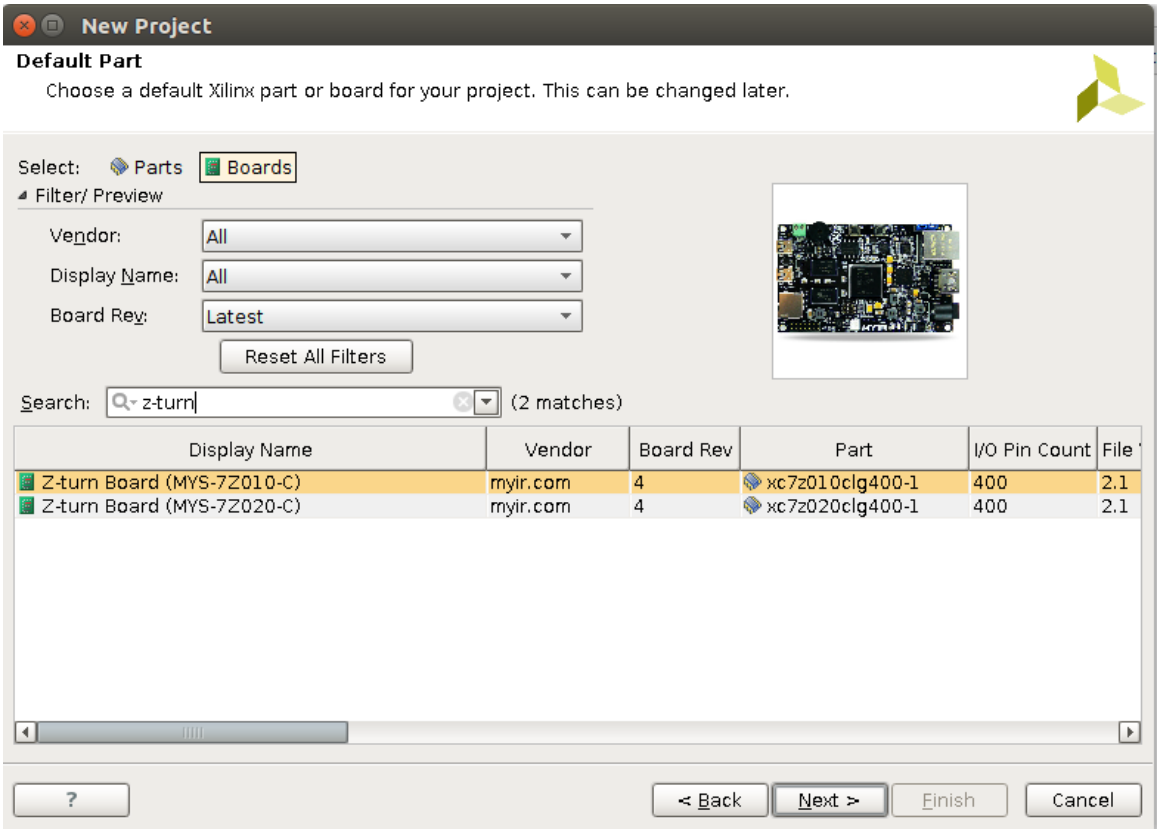

Figure 6: Board Selection

3. The **Peripherals** project should now look similar to Figure 7. In this project, we will implement the necessary files to control an RGB LED and a buzzer.

4. Create the Block Design. On the left, in the Flow Navigator window, choose **Create Block Design** and call it **Peripherals**. Press OK, and now the environment should look like in Figure 8. Vivado is based on an IP-centric design flow, in order to rapidly configure the AXI interconnect, AXI Masters and Slaves.

5. Now we will add the AXI Master (Zynq7 Processing System) and the PWM IP (PWM_v1.3) in the block diagram. The PWM IP was already implemented for you, such as you can choose it from the library, as in the pictures below. Press Add IP .

   Of course, there are other important IPs that should be added, such as AXI Interface or Processor System Reset, as well as the interconnectivity between them. However, the advantage of using Block Design is that these extra work is automated for you. Therefore, you should press Run Block Automation and Run Connection Automation,

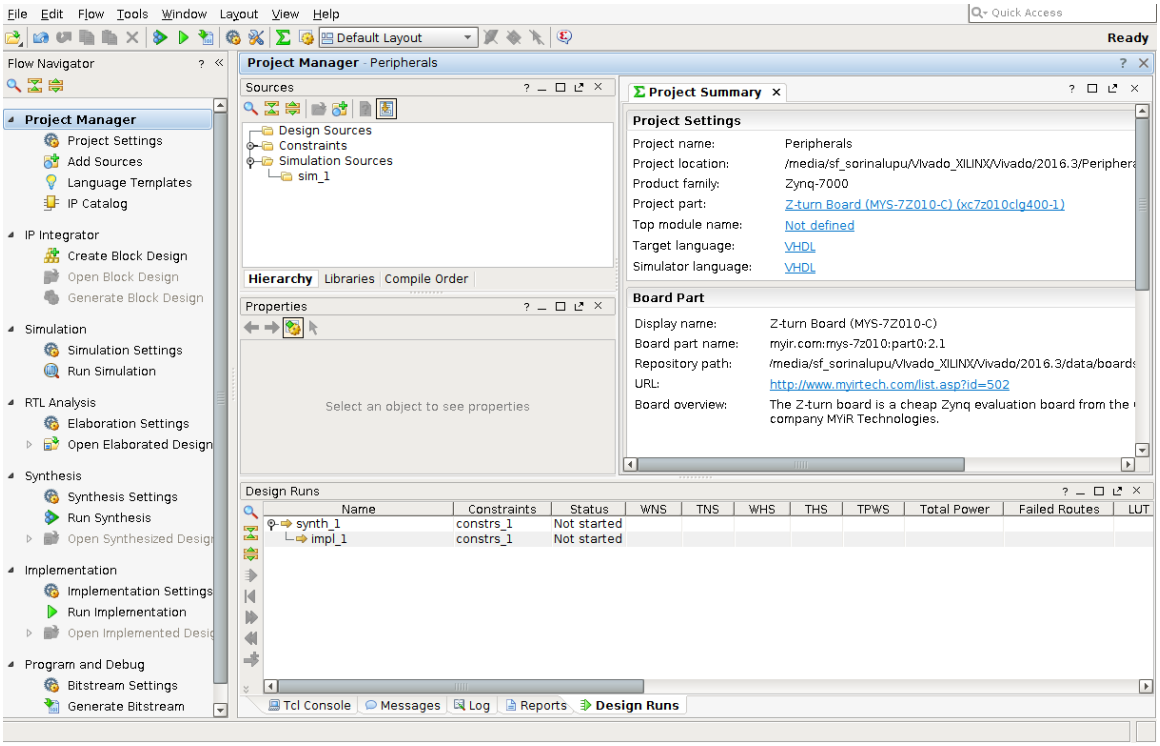

Figure 7: Project Structure

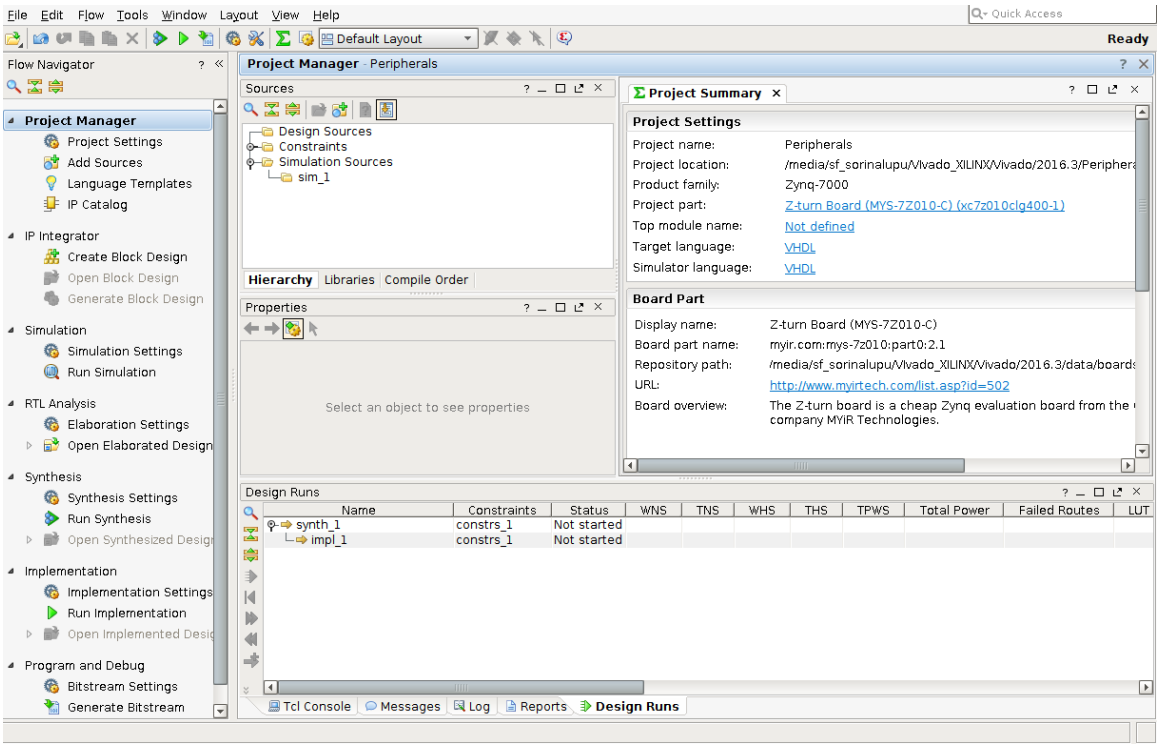

Figure 8: Project Structure

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 10. 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?

6. Let's dig into how the PWM Peripheral was designed in VHDL.

   Right click on the PWM_0 IP and select *Edit in IP Packager*. Another project (Figure 11) will open that contains the files generated for this IP. Vivado Design Suite provides you with *Packaging Steps* to configure the current IP. In the Source part, you will see an hierarchical design in VHDL. The top level is called *PWM_v1_0.vhd*

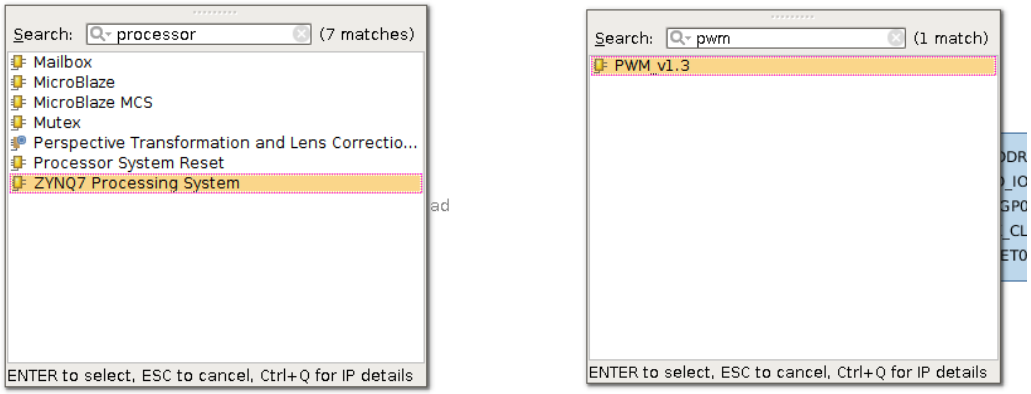

Figure 9: IP Blocks for Master and Slave

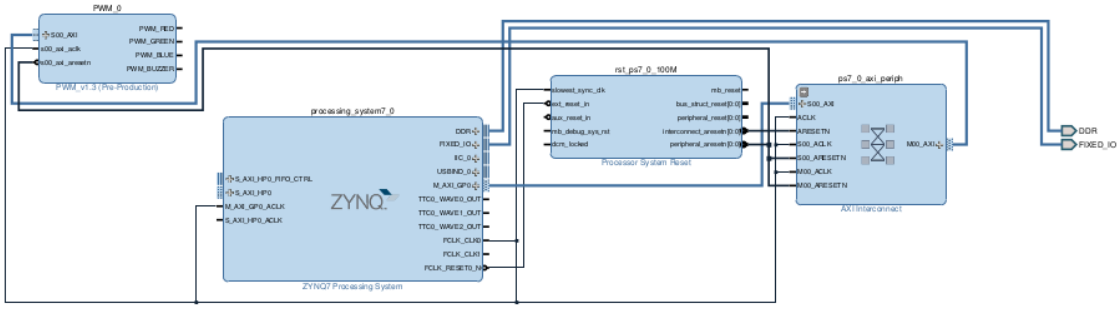

Figure 10: Block Diagram of our System

and contains an instantiation of the *PWM_v1_0_S00_AXI.vhd* where the logic is implemented. If you open *PWM_v1_0_S00_AXI.vhd*, you will see the registers specific to the AXI-Memory Map architecture (Figure 3). In addition, the user can add its own logic and signals. Thus, you can see the implementation of PWM outlined by the following comments:

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

Check Table 1 for a better understanding of the code.

In this exercise, you will need to change the following parameter **PWM_LEDs_COUNTER_PWM** to obtain a PWM frequency for the LEDs of 2kHz. Check its usage in this file and consider an input clock frequency of 50MHz.

After modifying the frequency, you need to re-package the design. In order to do so, go back in the Package IP window and click on the Steps that are not checked green. Merge the changes and finally press the *Review and Package* step and follow the indications from the window. Afterwards, the program will ask you if you want to exit. Press OK, and now you should be back in the previous project.

Now, the environment will detect a change in the IP and will ask you to update your component, as in Figure 12. Press **Upgrade Selected**, check in the **Synthesis Option: Global** and now the changes should take place. Don't panic if you see the following error: ERROR: BD 41758 The following clock pins are not connected to a valid clock source: HP0_ACLK, we have not configured the Processing System yet.

7. Create PWM output ports

   We need to create PWM outputs, in order to assign them to our board. Click on each PWM output and press CTRL-T or the Make External  symbol from the Menu. Do so for all the 4 ports. Now your IP should look like in Figure 5. You can also assign Highlight colors if you want.

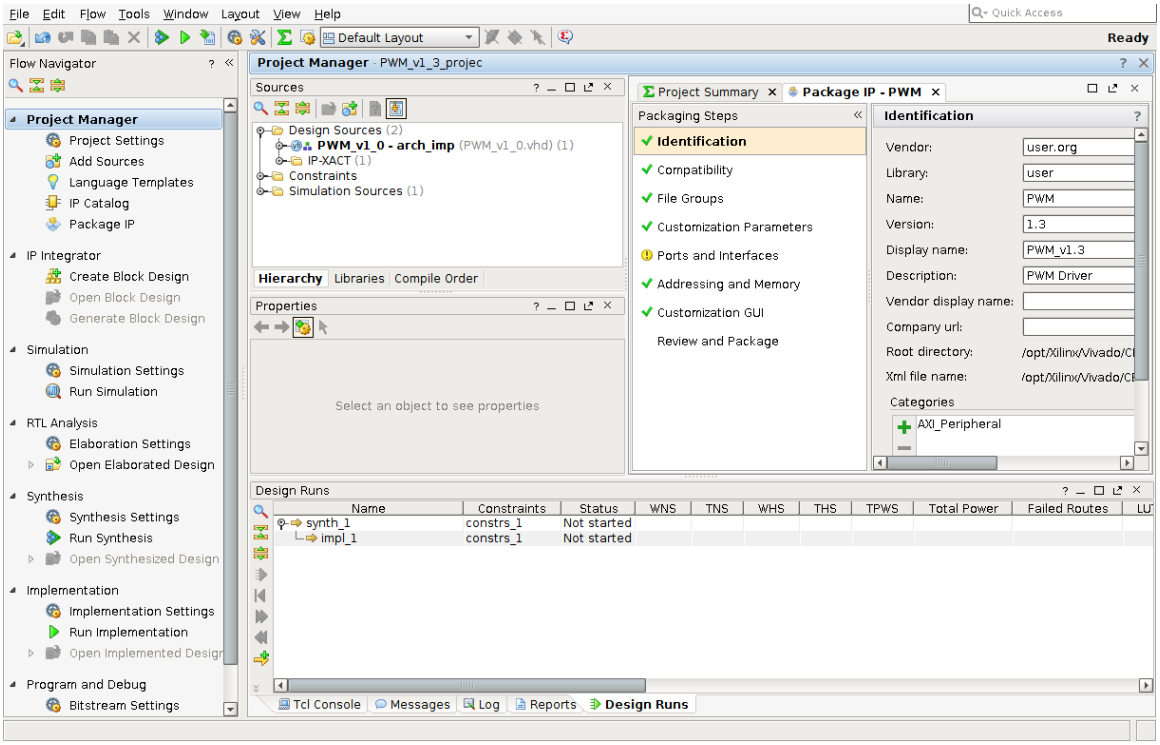

Figure 11: IP Project Structure

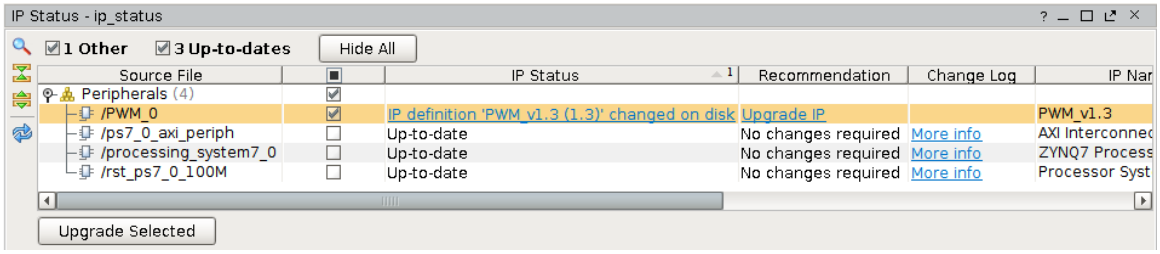

Figure 12: Upgrade IP Window

8. Configure the ZYNQ7 Processing System

   Double click on the ZYNQ7 Processing System IP will open a user-friendly interface for selecting the clocks/memory/IOs. Now, you need to follow the next sub-steps:

   - **Zynq Block Diagram**

     Take some time to look at the diagram in order to understand the interface between PS and PL. Ask your tutor for further details, as this diagram is extremely important for future development with this board.

   - **PS-PL Configuration**

     This part will solve our previous error. We don't need high performance slave interface, so just deselect the *S AXI HPO Interface*. Make sure everything is deselected as well.

   - **Peripherals I/O Pins** Deselect everything apart from UART, as we want to transmit through serial interface(USB-UART) different messages from the board. Make sure both Bank 0 and Bank 1 is set to LVCMOS 3.3V.

     

     Verify which UART (0 or 1) can be used in our case using the USB-UART cable. Look into the datasheet of the board: http://www.myirtech.com/download/Zynq7000/Z-TURN_SCH_V4_20150326.pdf

   - **MIO Configuration**

     If you did everything correctly in the previous step, now you should be able to see only UART selected.

   - **Clock Configuration** Let everything as default, apart from **PL Fabric Clocks**. Roll the clocks down and deselect FCLK_CLK1. As well, change the FCLK_CLK0 to 50MHz.

- **DDR Configuration** Delect **Enable DDR**

Now press OK. Your ZYNQ7 Processing System should look more simplistic now, and easier to understand.

9. Create VHDL Wrapper

   From the block design, we need to create the VHDL code. Therefore, in the Project Sources - Design Sources, you need to right click on the **Peripherals Peripherals.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, PWM_GREEN and PWM_BUZZER appear instantiated.

10. 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 thus "known", they will be thus matched to the "real" pins.

    Right click on the Constraints directory and Add Source (Add or Create Constraints), then click finish, after you give the file a name.

    Open it and write the following code that will define the nets and their dedicated ports.

```
## LED RED
set_property IOSTANDARD LVCMOS33 [get_ports PWM_RED]
set_property PACKAGE_PIN XX [get_ports PWM_RED]

## LED BLUE
set_property IOSTANDARD LVCMOS33 [get_ports PWM_BLUE]
set_property PACKAGE_PIN XX [get_ports PWM_BLUE]

## LED GREEN
set_property IOSTANDARD LVCMOS33 [get_ports PWM_GREEN]
set_property PACKAGE_PIN XX [get_ports PWM_GREEN]

## LED BUZZER
set_property IOSTANDARD LVCMOS33 [get_ports PWM_BUZZER]
set_property PACKAGE_PIN XX [get_ports PWM_BUZZER]

set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
```

Open the next design file of the board and check Table 1-2 (page 5). Match the Default Function with our application and replace the XX with the specific pin (BGA column) http://www.myirtech.com/download/Zynq7000/Z-turnBoard.pdf

11. Generate the bitstream and export the hardware

    Now, the project needs to be synthesized, implemented and finally the bitstream must be generated. The bitstream is, as the name says, a binary sequence which indicates which connections inside the FPGA must be activated. That's why it is politically corect to call "configure an FPGA" and not "programme an FPGA".

    To generate the bitstream, look into the Flow Manager on the left of the screen and press **Generate Bitstream (Figure 13)**. You will need to wait a bit longer until the bitstream is created. Check for errors in the process and solve them. Now, that the bitstream was successfully created, we need to export the hardware such as we can interact with it using the ARM dual-core microcontroller.

    To export the hardware, press **File** > **Edit** > **Export** > **Export Hardware**. Tick *Include bitstream* like in Figure 13.

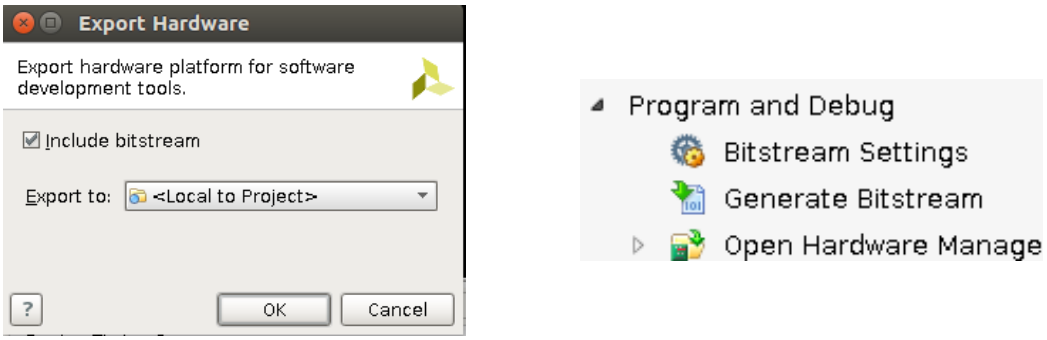

Figure 13: a) Export Hardware Window b) Generate Bitstream in Flow Manager

12. Let's write some C code

    From now on, we will create the link between the fabric and the dual-core ARM microcontroller. For this, we will use **Xilinx Software Development Kit (XSDK)** which is the Integrated Design Environment for creating embedded applications. If you worked in Eclipse before, XSDK shall be a piece-of-cake, as it is build on the standard Eclipse IDE, with plug-ins and Xilinx-dedicated tools.

    To open it, press **File** > **Launch SDK**. Now, the hardware shall be automatically imported in XSDK and now you can create a software application (Figure 14).

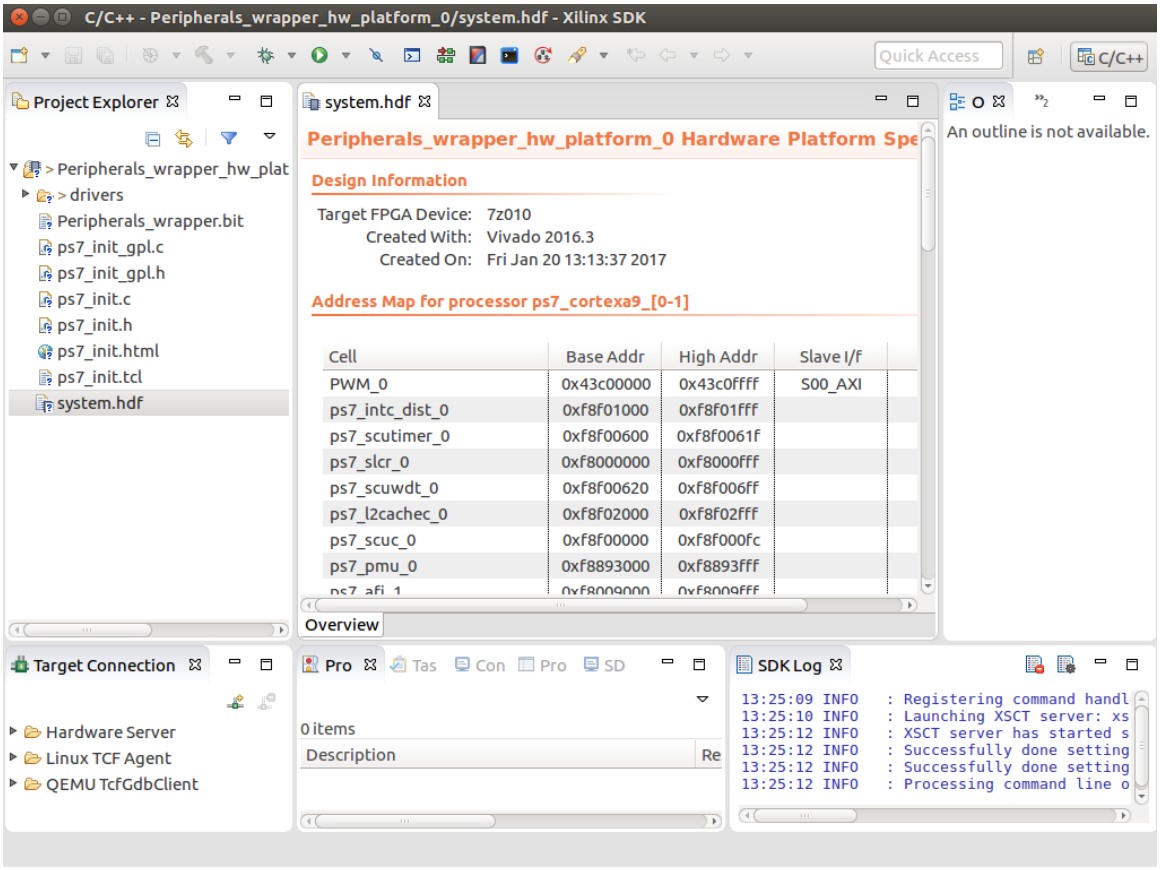

Figure 14: XSDK First Page after bitstream was exported

13. Our first "Hello World Application"

    Select **File** > **New** > **Application Project**.

    The New Project dialog box opens. As Project Name, type the name desired, for example *LED_Buzzer_Controller* (Figure 15). Press **Next**. From the Available Templates, choose "*Hello World*".

    Click **Finish**.

14. Connect the board

    To run the first application, you need to do the following simple steps on the hardware site:

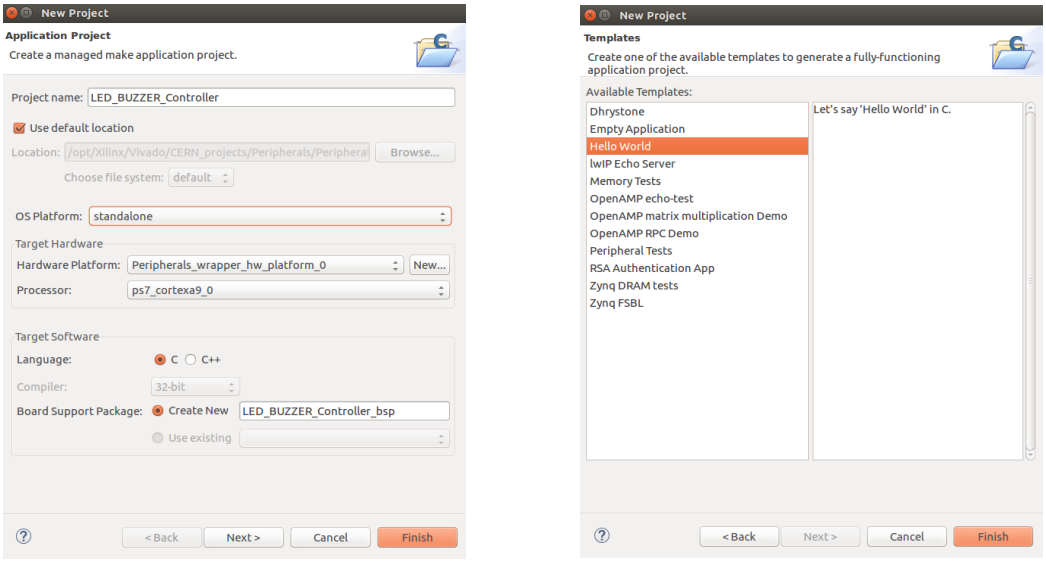

Figure 15: Starting a project in XSDK

- 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 16.

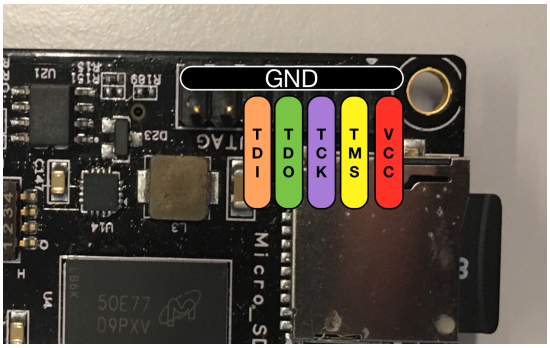

Figure 16: JTAG Pinout

15. Run the first application

    Open the project source files (in the **src** folder), and double-click on the helloworld.c. In that file, add the next sequence before the printing line of code. In this way, you will see the output running all the time on the terminal.

    ```
    while(1)
    ```

    Find the following icon to build the project  and click on the downward arrow. Make sure the Release version is selected and then build it. Verify that the build is successful.

    Next, we need to setup the programming chain, as firstly the FPGA is programmed (bitstream loaded) and then the application is loaded (.elf file).

    Go into **Run > Run configurations** and do the settings from Figures 17 and 18 below:

    Wait until the system is configured and programmed.

16. See the output

    Open a terminal **CTRL+ALT+T** and open the picocom terminal as following:

    ```
    picocom -b 115200 /dev/ttyUSB0
    ```

    Make sure you put the correct serial device "ttyUSBX", according to your system.

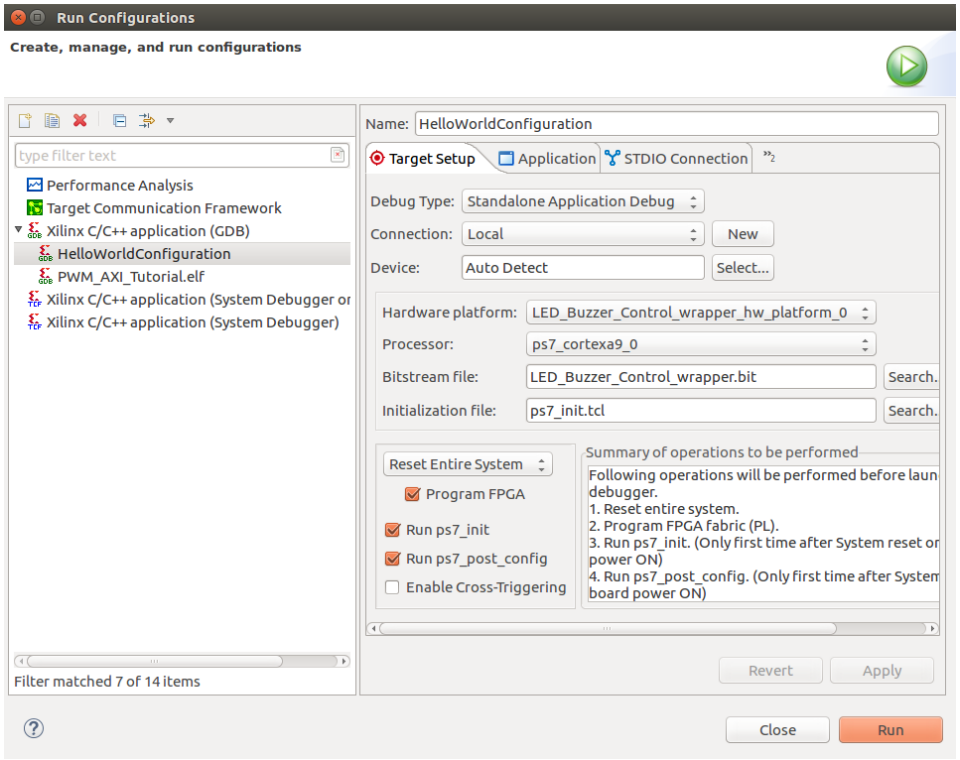

Figure 17: Target Setup

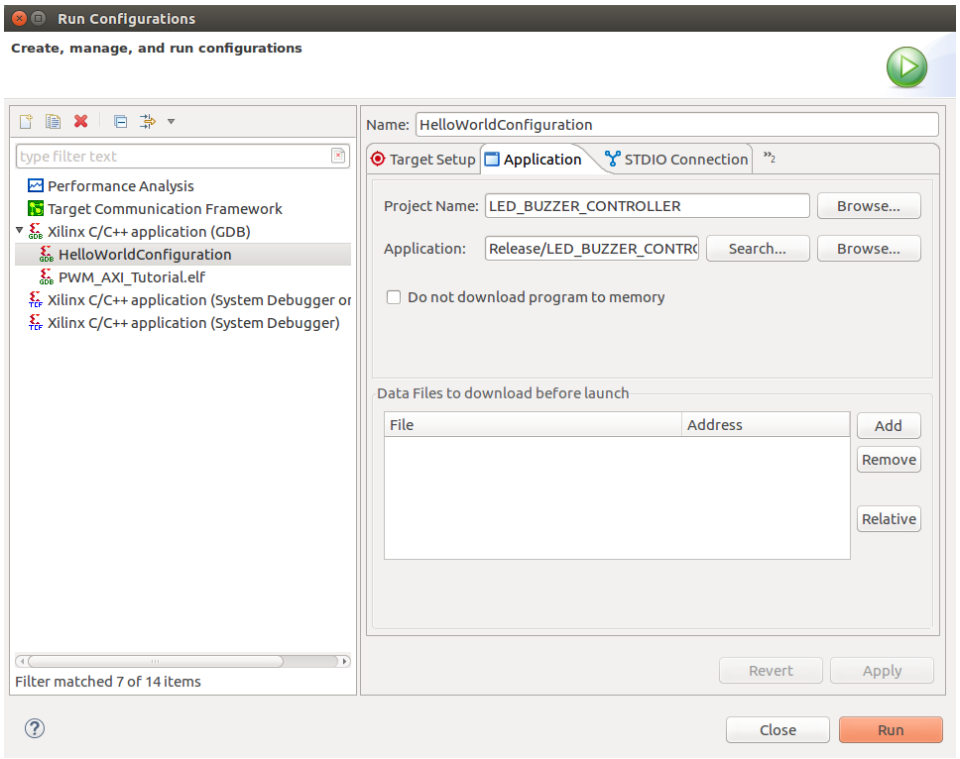

Figure 18: Application Setup

Now, you shall see a long list of "Hello World!"

17. Let's run a more complex application.

    At the beginning of this laboratory, we presented two applications we are going to implement. For simplicity, we provided you with the source code for this application that can be seen in the Laboratory folder or in Appendix C. You just need to run it and understand the code.

    The source code contains two $*.c$ files and two header files.

    $itoa_fcn.c$ - contains utility function used to convert a number to a string to be sent to UART using "print" function

main.c - contains the main logic of the programme

To run it, firstly, you need to delete the *helloworld.c*. Then, select the *src* folder, and import the aforementioned code by right click and selecting **Import > General > File System**. Add the Laboratory folder like in Figure 19.

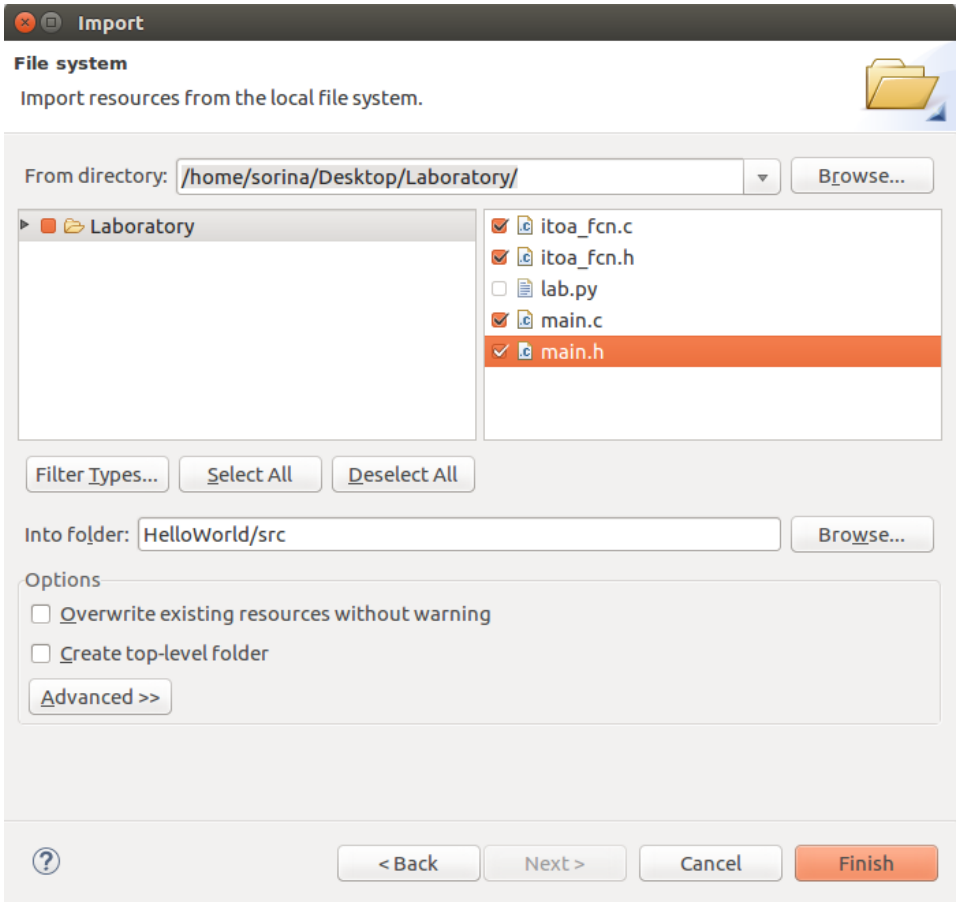

Figure 19: Import source files

Now, run the application similarly and HAVE FUN! (before moving to the LINUX part)

3 LINUX OS ON SYSTEM ON CHIP

3.1 Introduction

The current kit comes with Ubuntu 12.04 installed on a MicroSD card. Using this operating system, we will learn to implement the following steps:

1. we will communicate with the devkit through UART
2. we will transfer of files between the host computer and the devkit through Ethernet
3. we will control the LEDs and the buzzer mounted on the devkit
4. we will read accelerometer data from the ADXL346 accelerometer mounted on the board
5. we will implement a webserver to visualize the accelerometer data, using **bokeh**, which is a Python interactive visualization library

Before getting started, read again Project 2 from Chapter 2.Applications.

3.2 Let's get started!

Now, we will move on with running an operating system on the System on Chip. For this, we will do the following steps:

- disconnect the JTAG cable used previously to programme the board
- insert the mini SD Card in its socket if it is not there (the filesystems are resided on a 4GB miniSD memory card where the board also boots from.)
- find Jumper 1 and Jumper 2 (JP1 and JP2) and make sure they are in the following order:

  ```
  JP1 OFF
  JP2 ON
  ```

3.3 Communication with the devkit through UART

1. Take the development kit and plug the microUSB into the USB_UART socket (J6).

   This connection enables serial communication between the host computer and the board, as well as power to the board.

2. Open a terminal by pressing CTRL+ALT+T

3. To communicate with the board, we will use **picocom** which is a very simple and straightforward terminal emulator. In your console, type:

   ```
   picocom -b 115200 /dev/ttyUSB0
   ```

   Now, what you will see (Figure 20) is the terminal emulator of Ubuntu 12.04 running on the board. Type **reboot** and the system will re-start. After booting the board it gives you directly a bash shell as user root.

4. Close the communication by pressing CTRL-A and then CTRL-X

5. You shall see Appendix A for other settings and installation process.

Optional: If you have a screen with an HDMI port, you can connect it to the board (Figure 21). You can also connect a mouse and a keyboard, but you will need a USB-OTG cable (miniUSB - to - standard USB).

```
picocom v1.7

port is        : /dev/ttyUSB0
flowcontrol    : none
baudrate is    : 115200
parity is      : none
databits are   : 8
escape is      : C-a
local echo is  : no
noinit is      : no
noreset is     : no
nolock is      : no
send_cmd is    : sz -vv
receive_cmd is : rz -vv
imap is        :
omap is        :
emap is        : crcrlf,delbs,

Terminal ready

root@localhost:~#
root@localhost:~#
```

Figure 20: Picocom Terminal Emulator

Figure 21: Screen connected to the board

3.4 Transfer of files between the host computer and the devkit

In the current lab, we will develop source code on the host computer that needs to be transfered to the board.

There are three methods to transfer files:

- Through UART serial protocol
- Using SCP (Secure CoPy) - a simple transfer tool used to copy one or more files, often with known names, from host A (the computer) to host B (the board) or viceversa
- Using SFTP (SSH File Transfer Protocol) - which is also a transfer tool used to copy files, but in addition it allows directory listings, remote directories and files removal, creation of files etc.

To simplify life, we will use SCP. Never use UART unless you have a passion to encode/decode data or if other options are not available.

The SSH is already installed on the board. The settings can be seen in Appendix B.

The following steps shall be performed:

1. Connect the Ethernet cable between your computer and the board
2. Type in a new terminal the following command to connect to the board through SSH protocol.

   ```
   ssh root@192.168.2.2
   ```

   You will see basically the same shell as before 20. Don't forget that you also need to configure the IP on your computer. For this lab, this work was already done for you.
3. Next, we will transfer a file to the board.

   On the Desktop, in folder *Laboratory*, you will find a file entitled *lab.py*. This part will contain the final application that you are going to run. Right now, we will just take this file and transfer it to the board, and later on run it.

   For this, **open a new terminal window** and type the next command in the box. The two points ":" at the end of the next command will place the file in the User folder.

   ```
   scp -r ~/Desktop/Laboratory/lab.py root@192.168.2.2:
   ```

3.5 Control the LEDs and the buzzer mounted on the devkit

1. Make sure you have the emulator opened, by typing:

   ```
   ssh root@192.168.2.2
   ```
2. Control the LEDs

   In its simplest form, the LED class just allows control of LEDs from userspace. LEDs appear in $/sys/class/leds/$. Let's check them.

   Type in the terminal:

   ```
   ls /sys/class/leds/
   ```

   Right now, you shall see the following options:

   ```
   led_b  led_g  led_r  mmc0::  usr_led1  usr_led2
   ```

   The first three refer to the RGB LED available on the platform, while the last two refer to LED D29 and D30 on the same board. The LED can be turned on and off using the 'brightness' file. The minimum brightness is 0, and

the maximum is 255. As there is no variable brightness support, any value greater than 0 will turn the LED on. Now, let's turn on LED D29 and LED D30. For this, you need to type in the emulator:

```
echo 1 > /sys/class/leds/usr_led1/brightness
echo 1 > /sys/class/leds/usr_led2/brightness
```

Check the result on the board.

Optional: Implement your own toggle (on and off) of the RGB LED in a python script. In case you get stuck, check the Appendix Appendix D.

3. Control the BUZZER

The buzzer is seen as a device. "In Unix-like operating systems, a device file or special file is an interface for a device driver that appears in a file system as if it were an ordinary file.They allow software to interact with a device driver using standard input/output system calls, which simplifies many tasks and unifies user-space I/O mechanisms. "(Source: Wikipedia). They can be found in $/dev/input/eventX$. Let's see them by typing:

```
ls /dev/input/
```

You can thus see, a list of events. But how do we know which one is applicable to the Buzzer?

Let's write a Python script to access them. We will use the python-evdev library which allows you to read and write input events on Linux.

Open a new file, call it "buzzer.py" and write the following lines of code to get information for event0, event1 and event2

```
from evdev import InputDevice, ecodes
import time

device0 = InputDevice('/dev/input/event0')
device1 = InputDevice('/dev/input/event1')
device2 = InputDevice('/dev/input/event2')

print(device0)
print(device1)
print(device2)
```

You can see that the first two are actually the buzzer device and the accelerometer device, the ones that we need to use.

Let's turn the buzzer ON. For this, you need to add the following lines of code to the previous ones:

```
device0.write(ecodes.EV_SND, ecodes.SND_BELL, 1)
time.sleep(2)
device0.write(ecodes.EV_SND, ecodes.SND_BELL, 0)
print(device0.capabilities(verbose=True))
```

We understood how InputDevice works, but, what are **ecodes** and why they are used in the previous lines of code to access the buzzer?

An input event is represented by standard $input\_event$ structure defined in $linux/input.h$ header file. According to this header file, the structure is:

- timestamp
- type $->$ EV_SND (for the buzzer)
- code $->$ SND_BELL (for the buzzer)
- value $->$ (1 or 0 for ON and OFF)

3.6 Read accelerometer data

In the previous part, we saw that the accelerometer can be accessed from $/dev/input/event1$.

Now it's your turn to read the device and display the accelerometer data. To understand how to do this, you need to read the documentation of python-evdev library. http://python-evdev.readthedocs.io/en/latest/. If you get stuck, check Appendix Appendix E.

3.7 Webserver

Now, we will see how to display the accelerometer values on an webserver running on the host computer.

For this part, Bokeh was used. Bokeh is a Python interactive visualization library that targets modern web browsers for presentation. Its goal is to provide elegant, concise construction of novel graphics.

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 23.

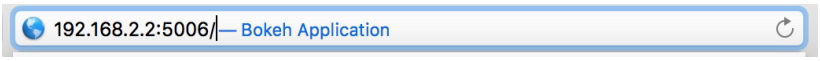

Figure 22: Host address

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

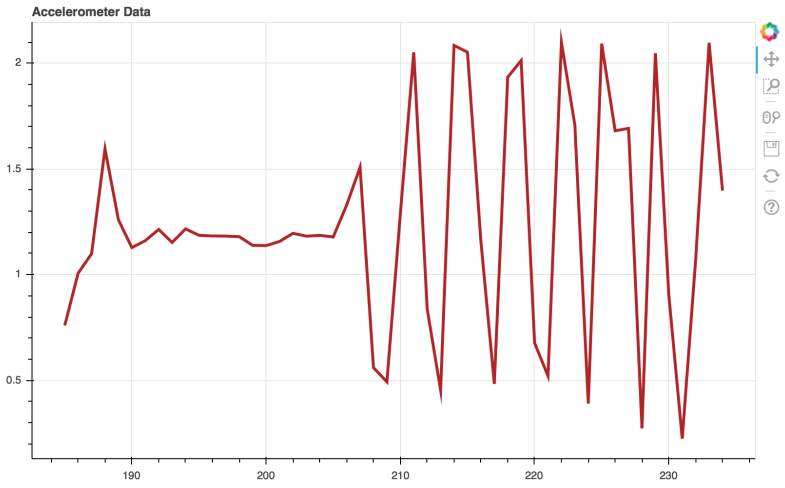

Figure 23: Host address

Appendices

A Setting up the picocom terminal

- Install Picocom:

```
sudo apt-get install picocom
```

- Find the device dev path for the board:

```
dmesg | grep tty
```

- Give permission to users to write and read, presuming that the device dev path is ttyUSB0

```
sudo chmod 666 /dev/ttyUSB0
```

- Start picocom with a baudrate of 115200

```
picocom -b 115200 /dev/ttyUSB0
```

- To exit Picocom: CTRL-A and CTRL-X

B CONNECTING THE BOARD TO A NETWORK

- Get the board connected to Internet
  To be able to get the board connected to the Internet, you need to use the laptop as a NAT box. This is done by enabling âĂIJShare internet connection via ethernet portâĂİ on the laptop. The (only) ethernet interface of the laptop is connected to the board. The connection to the internet is done through the wireless (which should not have 802.1x enabled).
  On the side of the board, we configure the network adaptor in the file /etc/network/ interfaces as following:

```
auto eth0
iface eth0 inet static
address 192.168.2.2
netmask 255.255.255.0 gateway 192.168.2.1 dns-nameservers 192.168.2.1
```

  After adding these lines to the file, the network stack must be started:

```
service networking start
```

  (this will also be done automatically at boot time).
- Test the network by running:

```
apt-get update
ping 8.8.8.8
```

- Getting ssh to work
  A bit more convenieÌĹnt way to have a terminal connection is directly over the network with ssh. This makes the installation of ssh necessary on the board:

```
apt-get install ssh
```

  Next, set the root password with the "passwd" command. Now, from the NAT host (i.c. your laptop), it is possible to connect though ssh:

```
ssh -l root 192.168.2.2
```

C Cheating Sheet - Final Code for FPGA PART

main.c

```
#include "xparameters.h"
#include "xil_io.h"
#include "stdlib.h"
#include "itoa_fcn.h"

#define MY_PWM_MEMORY_MAP 0x43C00000 //This value is found in the Address editor tab in Vivado (next to D
#define MY_PWM_MEMORY_MAP_OFFSET 4
#define FREQUENCY_FPGA 50000000 // 50 MHz

void print_memory_mapped_registers();

// Generate a random number between 1 and 3 (1 - red; 2 - green;  3 - blue)
// Every time the red color is showed, activate the buzzer for 1 seconds
int main(){

    int red_pwm = 0;
    int blue_pwm = 0;
    int green_pwm = 0;
    int buzzer_pwm = 0;


    int led_duty_cycle_max = 12000;
    int buzzer_pwm_value =  led_duty_cycle_max;

    int rand_choose_color;
    int i;
    while(1){
        rand_choose_color = (rand() % 3) + 1;
        print("Generated random color: ");
        switch (rand_choose_color){
                case 1:
                    red_pwm = 0;
                    blue_pwm = led_duty_cycle_max;
                    green_pwm = led_duty_cycle_max;
                    buzzer_pwm = buzzer_pwm_value;
                    print("RED");
                    print("\n");
                break;

                case 2:
                    red_pwm = led_duty_cycle_max;
                    blue_pwm = 0;
                    green_pwm = led_duty_cycle_max;
                    buzzer_pwm = 0;

                    print("BLUE");
                    print("\n");
                break;
```

```
                case 3:
                    red_pwm = led_duty_cycle_max;
                    blue_pwm = led_duty_cycle_max;
                    green_pwm = 0;
                    print("GREEN");
                    print("\n");
                default:
                    red_pwm = led_duty_cycle_max;
                    blue_pwm = led_duty_cycle_max;
                    green_pwm = 0;
                    buzzer_pwm = 0;

        }


        Xil_Out32(MY_PWM_MEMORY_MAP, red_pwm);
        Xil_Out32((MY_PWM_MEMORY_MAP + MY_PWM_MEMORY_MAP_OFFSET), blue_pwm);
        Xil_Out32((MY_PWM_MEMORY_MAP + 2 * MY_PWM_MEMORY_MAP_OFFSET), green_pwm);
        Xil_Out32((MY_PWM_MEMORY_MAP + 3 * MY_PWM_MEMORY_MAP_OFFSET), buzzer_pwm);

        print_memory_mapped_registers();

        for(i=0;i < 3 * FREQUENCY_FPGA; i++){
            if (i == FREQUENCY_FPGA)
                Xil_Out32((MY_PWM_MEMORY_MAP + 3 * MY_PWM_MEMORY_MAP_OFFSET), 0);
        }
    }
}

void print_memory_mapped_registers(){
    int register_red_pwm;
    int register_green_pwm;
    int register_blue_pwm;
    char buffer_red[10];
    char buffer_blue[10];
    char buffer_green[10];

    register_red_pwm = Xil_In32(MY_PWM_MEMORY_MAP);
    register_green_pwm = Xil_In32(MY_PWM_MEMORY_MAP);
    register_blue_pwm = Xil_In32(MY_PWM_MEMORY_MAP);

    itoa_fcn(register_red_pwm, buffer_red);
    itoa_fcn(register_green_pwm, buffer_green);
    itoa_fcn(register_blue_pwm, buffer_blue);

    print("(REG READ at address 0x43C00000): Duty Cycle for RED: = ");
    print(buffer_red);
    print("\n");
    print("(REG READ at address 0x43C00004): Duty Cycle for GREEN: = ");
    print(buffer_green);
    print("\n");
    print("(REG READ at address 0x43C00008): Duty Cycle for BLUE: = ");
```

```
    print(buffer_blue);
    print("\n");
    print("------------------------------------------------------------");

}
```

itoa.c

```
#include "itoa_fcn.h"
#include "string.h"
void reverse(char s[]);
void itoa_fcn(int n, char s[])
 {
     int i, sign;

     if ((sign = n) < 0)  /* record sign */
         n = -n;          /* make n positive */
     i = 0;
     do {       /* generate digits in reverse order */
         s[i++] = n % 10 + '0';   /* get next digit */
     } while ((n /= 10) > 0);     /* delete it */
     if (sign < 0)
         s[i++] = '-';
     s[i] = '\0';
     reverse(s);
}

void reverse(char s[])
{
    int i, j;
    char c;

    for (i = 0, j = strlen(s)-1; i<j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

itoa.h

```
 void itoa_fcn(int n, char s[]);
```

D Cheating Sheet - Toggle LEDs

```
# LEDs toggle every 1 second

import time


def ledRGBon():
        red = open("/sys/class/leds/led_r/brightness", "w")
        green = open("/sys/class/leds/led_g/brightness", "w")
        blue = open("/sys/class/leds/led_b/brightness", "w")

        red.write(str(1))
        green.write(str(1))
        blue.write(str(1))

        red.close()
        green.close()
        blue.close()


def ledRGBoff():
        red = open("/sys/class/leds/led_r/brightness", "w")
        green = open("/sys/class/leds/led_g/brightness", "w")
        blue = open("/sys/class/leds/led_b/brightness", "w")

        red.write(str(0))
        green.write(str(0))
        blue.write(str(0))

        red.close()
        green.close()
        blue.close()


while(1):
    ledRGBoff()
    time.sleep(1)
    ledRGBon()
    time.sleep(1)
```

E Cheating Sheet - Read Accelerometer Data

```
# Accelerometer Reading

from evdev import InputDevice, ecodes
import time
import threading


def check_empty(value):
    if not value:
        value = 0
    return value


def run_acc_readout():
    global G_acc
    dev = InputDevice('/dev/input/event1')
    print(dev)
    x = 0
    y = 0
    z = 0

    print("Accelerometer Data")
    while True:
        try:
            for event in dev.read():
                if event.type == ecodes.EV_ABS:
                        if (event.code == ecodes.ABS_X):
                            x = event.value
                        if (event.code == ecodes.ABS_Y):
                            y = event.value
                        if (event.code == ecodes.ABS_Z):
                            z = event.value

                        x = check_empty(x)
                        y = check_empty(y)
                        z = check_empty(z)

                        print('x :' + str(x) + ' y = ' + str(y) + ' z = ' + str(z))
        except IOError as e:
            time.sleep(0.01)


try:
    t = threading.Thread(target=run_acc_readout)
    t.start()

except (KeyboardInterrupt, SystemExit):
    print('\n! Received keyboard interrupt, quitting threads.\n')
```

F Cheating Sheet - Run the webserver

```
from evdev import InputDevice, ecodes
import time
import math
from bokeh.plotting import figure, curdoc
from bokeh.driving import linear
import threading

def ledRGBon():
        red = open("/sys/class/leds/led_r/brightness", "w")
        green = open("/sys/class/leds/led_g/brightness", "w")
        blue = open("/sys/class/leds/led_b/brightness", "w")

        red.write(str(1))
        green.write(str(1))
        blue.write(str(1))

        red.close()
        green.close()
        blue.close()

def ledRGBoff():
        red = open("/sys/class/leds/led_r/brightness", "w")
        green = open("/sys/class/leds/led_g/brightness", "w")
        blue = open("/sys/class/leds/led_b/brightness", "w")

        red.write(str(0))
        green.write(str(0))
        blue.write(str(0))

        red.close()
        green.close()
        blue.close()

def ledRedOn():
    red = open("/sys/class/leds/led_b/brightness", "w")
    red.write(str(1))
    red.close()

def buzzerOn(buzzer):
    buzzer.write(ecodes.EV_SND, ecodes.SND_BELL, 1)

def buzzerOff(buzzer):
    buzzer.write(ecodes.EV_SND, ecodes.SND_BELL, 0)

def check_empty(value):
    if not value:
        value = 0
    return value

def resultant_acc(x, y, z):
    g = math.sqrt(x * x + y * y + z * z) * 4 / 1000
```

```
    print('total acc = ' + str(g))
    return g

global G_acc
G_acc = 0

@linear()
def update(step):
    global G_acc
    ds1.data['x'].append(step)
    ds1.data['y'].append(G_acc)
    if len(ds1.data['x']) > 50:
        ds1.data['x'].pop(0)
        ds1.data['y'].pop(0)
    ds1.trigger('data', ds1.data, ds1.data)

try:
    ledRGBoff()
    ledRGBon()
    time.sleep(1)
    p = figure(plot_width=800, plot_height=500, title = 'Accelerometer Data')
    acc_values = p.line([], [], color = "firebrick", line_width = 3)
    ds1 = acc_values.data_source

    curdoc().add_root(p)
    curdoc().add_periodic_callback(update, 50)

except (KeyboardInterrupt, SystemExit):
    print ('\n! Received keyboard interrupt, quitting threads.\n')

def run_acc_readout():
    global G_acc
    dev = InputDevice('/dev/input/event1')
    buzzer = InputDevice('/dev/input/event0')
    print(dev)
    x = 0
    y = 0
    z = 0
    print("Accelerometer Data")
    while True:
        try:
            for event in dev.read():
                if event.type == ecodes.EV_ABS:
                        if (event.code == ecodes.ABS_X):
                            x = event.value
                        if (event.code == ecodes.ABS_Y):
                            y = event.value
                        if (event.code == ecodes.ABS_Z):
                            z = event.value

                        x = check_empty(x)
                        y = check_empty(y)
```

```
                        z = check_empty(z)

                        print('x :' + str(x) + ' y = ' + str(y) + ' z = ' + str(z))
                        G_acc = resultant_acc(x, y, z)
                        if (G_acc > 1.5):
                            ledRedOn()
                            buzzerOn(buzzer)
                        else:
                            ledRGBoff()
                            buzzerOff(buzzer)
        except IOError as e:
            time.sleep(0.01)

try:
    t = threading.Thread(target=run_acc_readout)
    t.start()

except (KeyboardInterrupt, SystemExit):
    print('\n! Received keyboard interrupt, quitting threads.\n')
```

Reading List