

The International School of Trigger and Data Acquisition 2017

## **SYSTEM ON CHIP - FPGA**

January 31, 2017

# Contents

<b>1</b>	<b>OVERVIEW</b>	<b>3</b>
<b>2</b>	<b>IMPLEMENTATION IN HARDWARE</b>	<b>4</b>
2.1	Block diagram of the study case . . . . .	4
2.2	Getting started with the environment . . . . .	5
2.2.1	System on Chip Design Flow in Vivado Design Suite . . . . .	5
2.2.2	Pulse Width Modulation (PWM) Slave Peripheral . . . . .	6
2.2.3	Implementation . . . . .	8
<b>3</b>	<b>LINUX OS on System on Chip</b>	<b>15</b>
3.1	Introduction . . . . .	15
3.2	Let's get started! . . . . .	15
3.3	Communication with the devkit through UART . . . . .	15
3.4	Transfer of files between the host computer and the devkit . . . . .	17
3.4.1	How is LINUX OS interacting with both the accelerometer and the buzzer? . . . . .	17
3.4.2	How is LINUX OS interacting with the LEDs? . . . . .	17
3.4.3	Screen . . . . .	17
3.4.4	File transfer between host computer and the board . . . . .	17
3.5	Interact with the RGB LED and the Buzzer . . . . .	17
3.6	Webserver . . . . .	17
3.7	Booting LINUX with System on Chip . . . . .	17
3.8	Interfacing the accelerometer + webserver . . . . .	17
<b>Appendices</b>		<b>18</b>
<b>A</b>	<b>Setting up the picocom terminal</b>	<b>18</b>
<b>B</b>	<b>Connecting the board to a network</b>	<b>19</b>
<b>C</b>	<b>Skeleton of the Code for the LINUX Part</b>	<b>20</b>
<b>D</b>	<b>Cheating Sheet - Final Code for FPGA PART</b>	<b>22</b>
<b>E</b>	<b>Cheating Sheet - Final Code for LINUX PART</b>	<b>23</b>
<b>Reading List</b>		<b>24</b>

# 1 OVERVIEW

System on Chip are getting more and more popular, exceeding the capabilities of a simple microcontroller. They 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)** processor-centric platform which offers software, hardware and I/O programmability in one single chip.

You are going to programme two applications and understand the workflow and the challenges that a system on chip requires.

To be added 2-3 paragraphs

## **2 IMPLEMENTATION IN HARDWARE**

### **2.1 Block diagram of the study case**

put picture FPGA

put picture block diagram

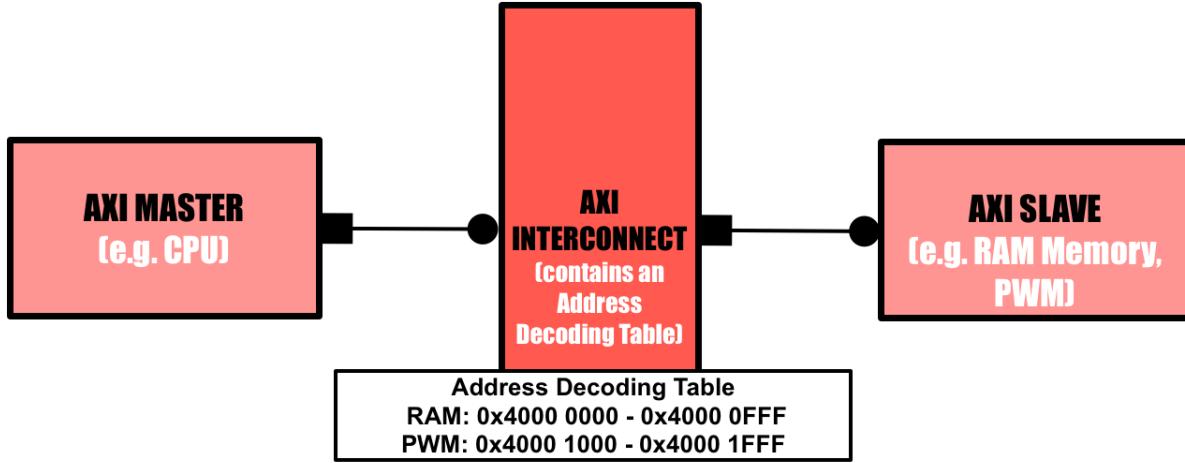
We are going to introduce all the concepts of the block diagram throughout the next pages of the laboratory.

## 2.2 Getting started with the environment

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). 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.

### 2.2.1 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.



#### Simplified example of data flow:

- 1** AXI Master initiates a transaction (e.g. write into memory)
- 2** AXI Interconnect looks into the Address Decoding Table to route transaction to suitable slave
- 3** AXI Slave produces response to the transaction request

Figure 1: 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 2. In our case, we will use an AXI Memory Mapped architecture, and PWM (Pulse Width Modulation) slave module to control LEDs and a buzzer.

### AXI Memory-Mapped vs. AXI Stream

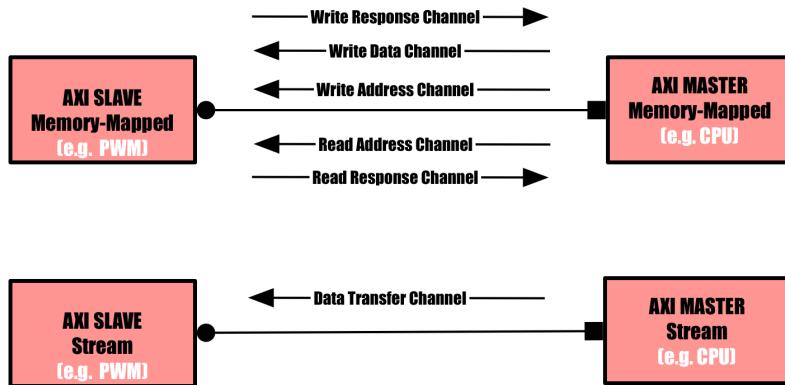


Figure 2: AXI Memory Mapped and AXI Stream arhitecture

## 2.2.2 Pulse Width Modulation (PWM) Slave Peripheral

### INTRODUCTION

In Figure 3, 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 .

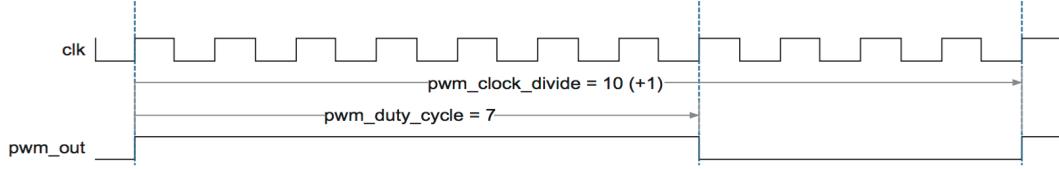


Figure 3: PWM waveform (Source: Application Note 333, Altera)

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 4, 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.

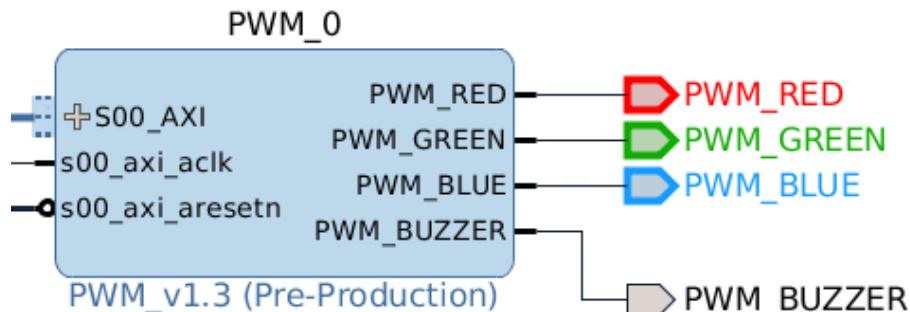


Figure 4: PWM Slave IP

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

### 2.2.3 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 Constraints 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 5). Press **Next** and **Finish**. Now, the project will initialize.

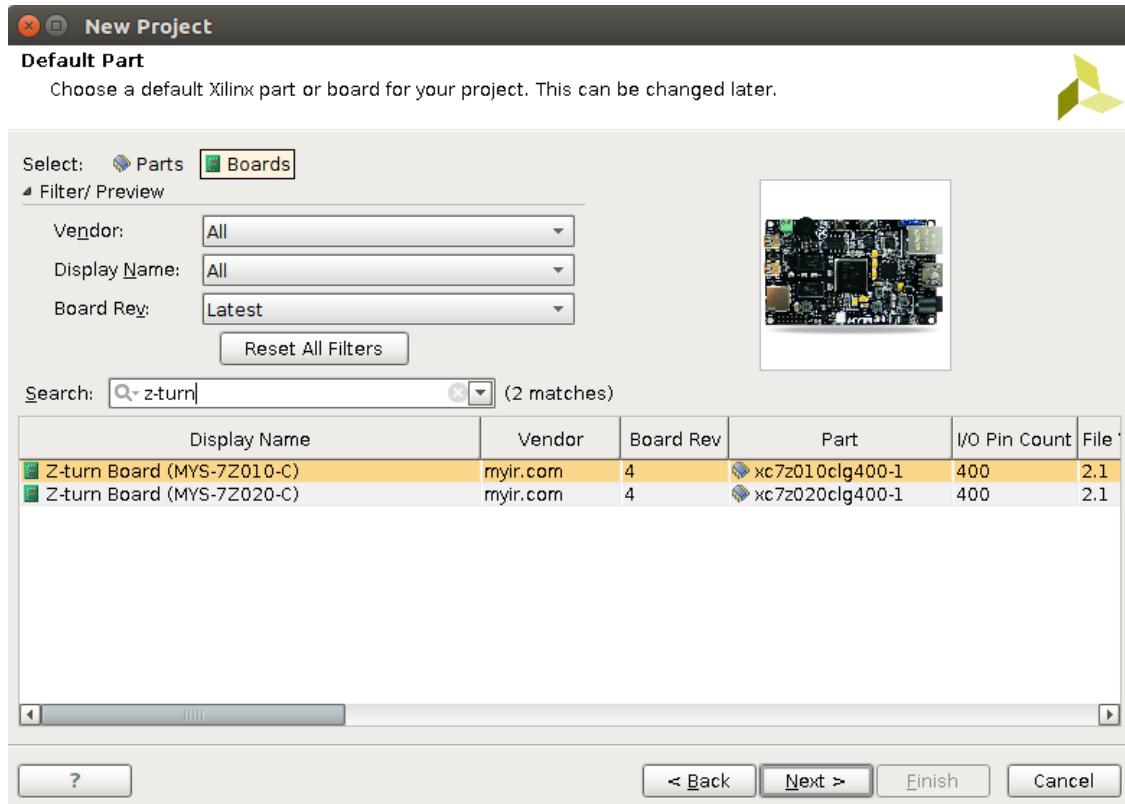


Figure 5: Board Selection

3. The **Peripherals** project should now look similar to Figure 6. 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 7. 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](#), 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 9. 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 10) 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*

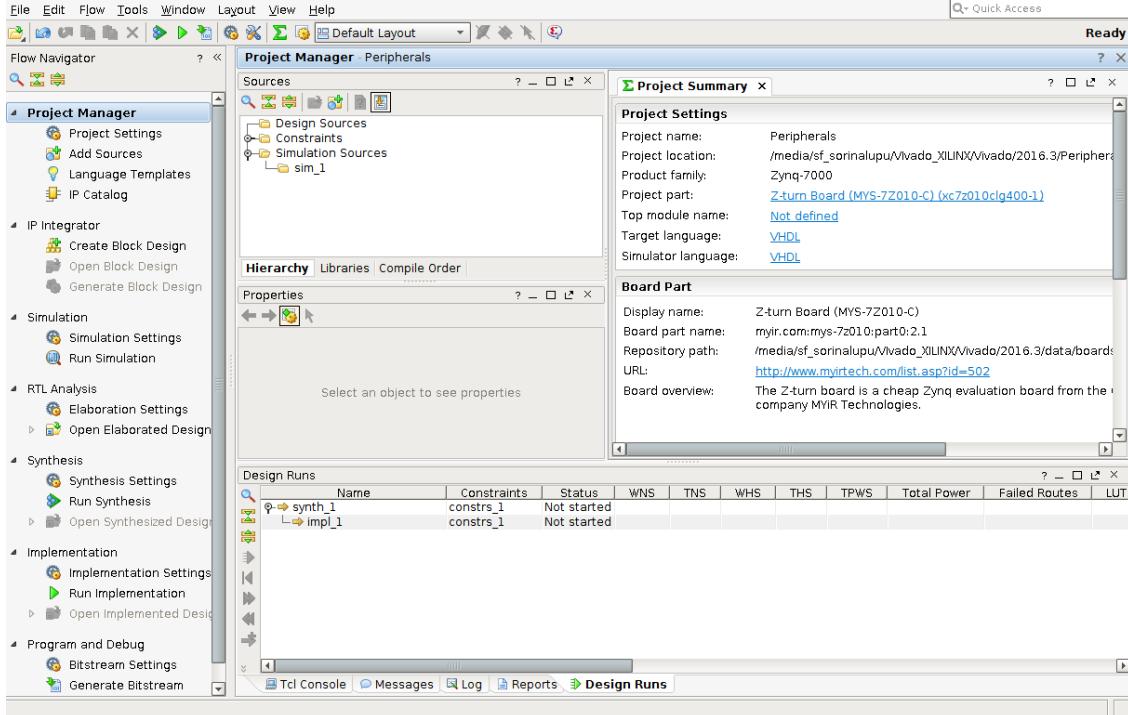


Figure 6: Project Structure

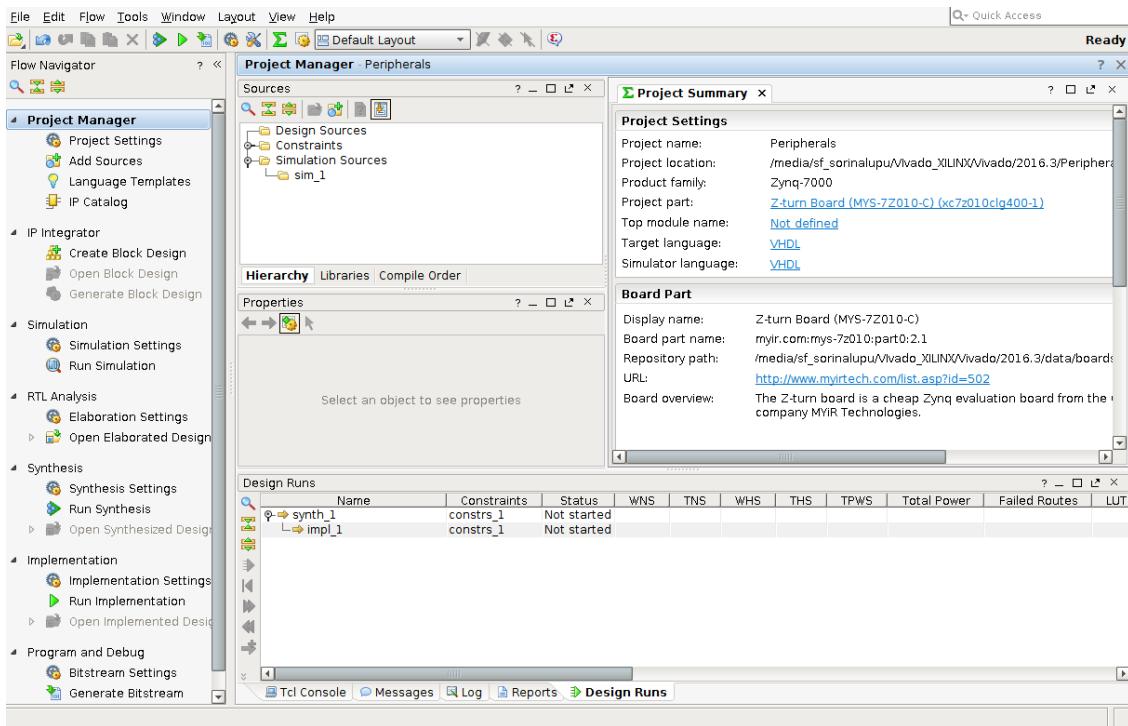


Figure 7: Project Structure

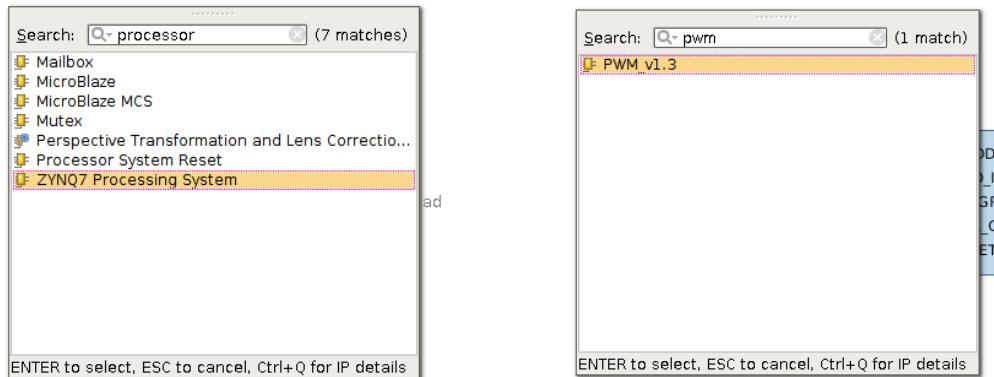
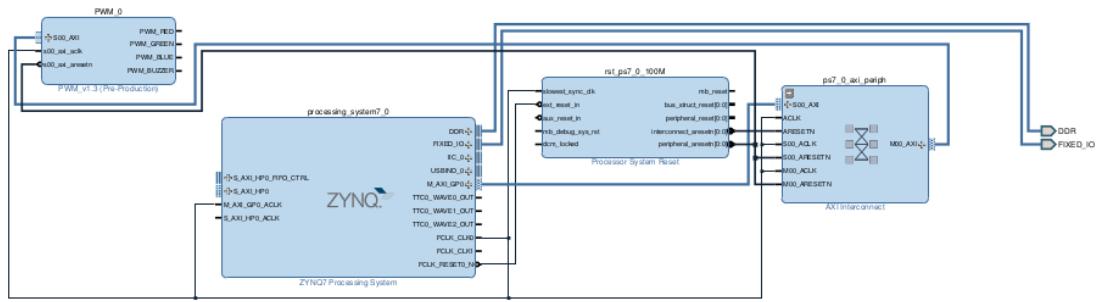


Figure 8: IP Blocks for Master and Slave

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 2). In addition, the user can add its own logic and signals. Thus, you can see the implementation of PWM outlined by the following comments:

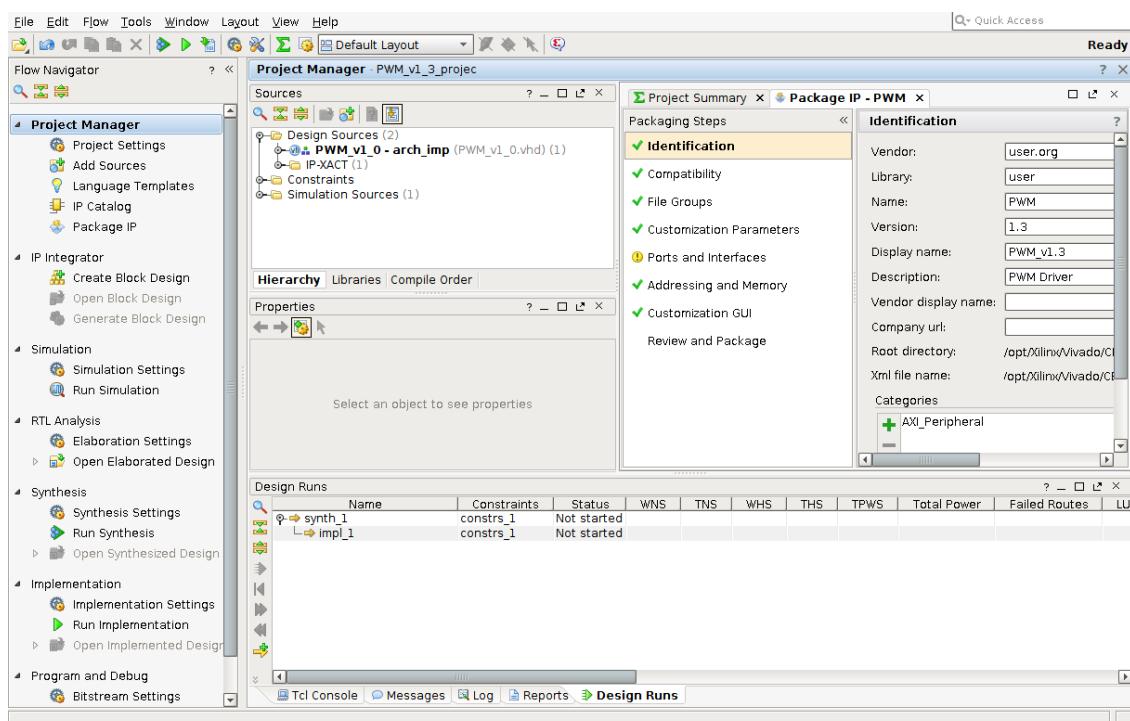
```
-- User to add parameters/ports/logic here
```



**Figure 9:** Block Diagram of our System

-- User parameters/ports/logic ends

Check Table 1 for a better understanding of the code.



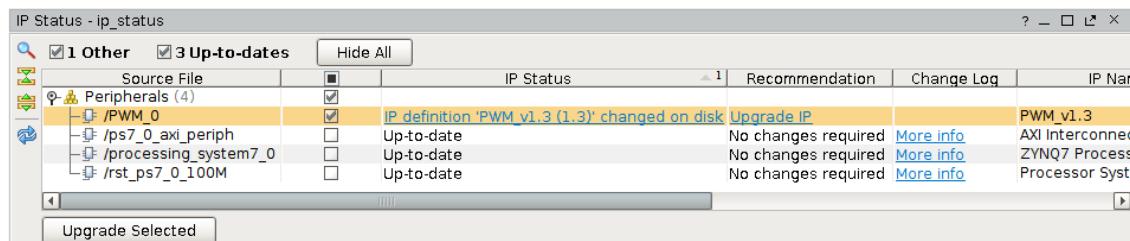
**Figure 10:** IP Project Structure



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 11. 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: HPO\_ACLK**, we have not configured the Processing System yet.



**Figure 11:** Upgrade IP Window

## 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 4. You can also assign Highlight colors if you want.

## 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/I/Os. 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 LVC MOS 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](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** Deselect **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]
```



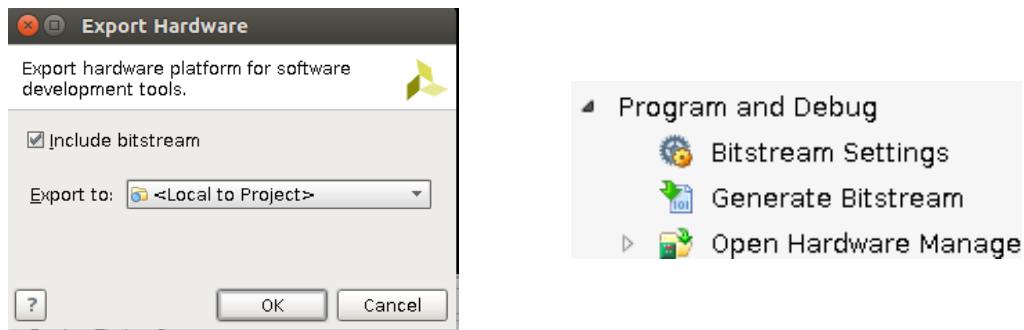
Open the next design file of the board and check Table 1-2 (page 5). Match the Default Function with our application and replace the XX with the specific pin (BGA column) <http://www.myirtech.com/download/Zynq7000/Z-turnBoard.pdf>

## 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 correct 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 12**). 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 **12**.

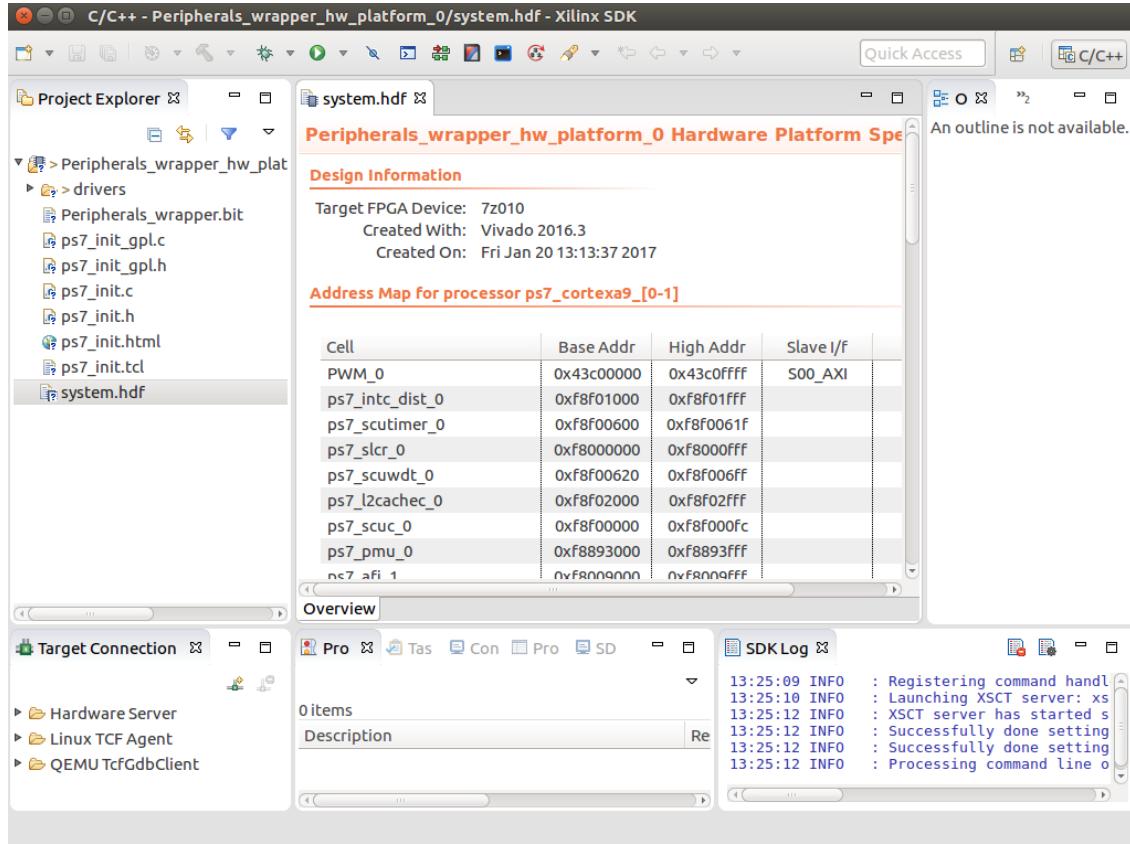


**Figure 12:** a) Export Hardware Window b) Generate Bitstream in Flow Manager

## 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 13).



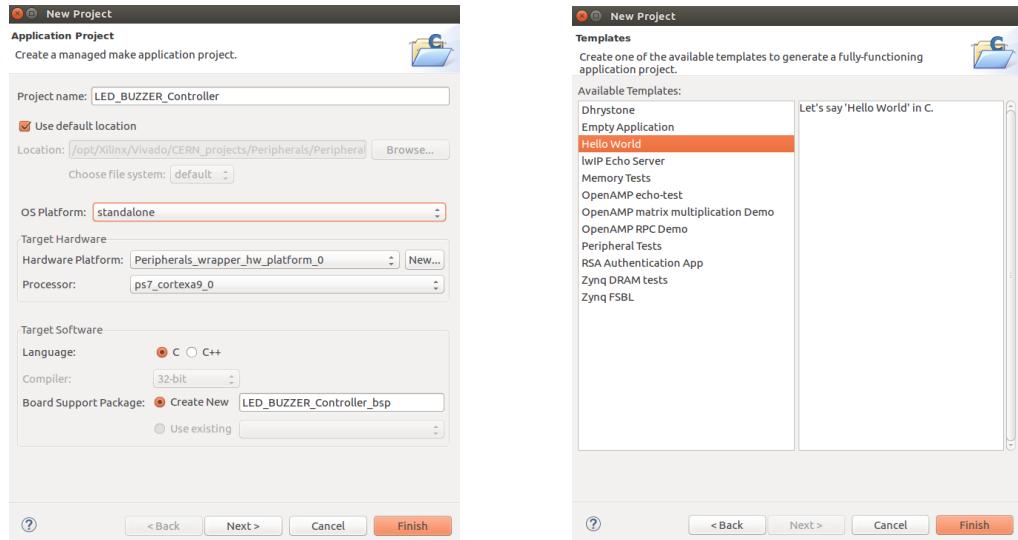
**Figure 13:** XSDK First Page after bitstream was exported

## 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 14). Press **Next**. From the Available Templates, choose "*Hello World*".

Click **Finish**.



**Figure 14:** Starting a project in XSDK

## 14. Connect the board

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

- 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.

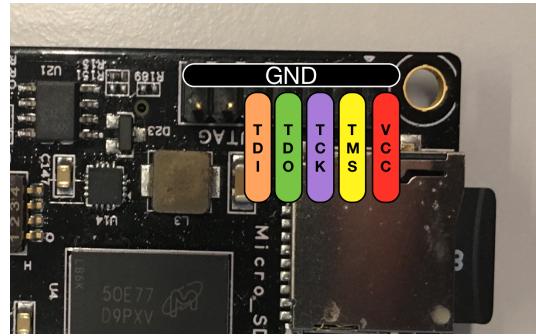


Figure 15: JTAG Pinout

- Connect the programmer on the JTAG port. Make sure the connector matches the pinout from Figure 15.

15. Run the first application

Open the project source files (in the **src** folder), and double-click on the `helloworld.c`. In that folder, 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. Make sure the build is successful.

Afterwards, 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 below:

16. See the output

Open a terminal **CTRL+ALT+T** and type:

17.

### 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:

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

#### 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:

- you disconnected the JTAG cable used previously to programme the board
- the mini SD Card is inserted in its socket (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 16) is an emulator of the terminal 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.

A screenshot of the Picocom terminal emulator window. The window title is "picocom v1.7". Inside, the configuration settings are listed:

```
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
```

At the bottom, the prompt shows a root shell:

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

Figure 16: Picocom Terminal Emulator

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 17). You can also connect a mouse and a keyboard, but you will need a USB-OTG cable (miniUSB - to - standard USB).



Figure 17: 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 transferred 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. 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 emulator as before.

2. 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 skeleton used to develop the next steps. Right now, we will just take this skeleton file and transfer it to the board.

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. Run *lab.py*

```
python lab.py
```

#### 3.4.1 How is LINUX OS interacting with both the accelerometer and the buzzer?

The I/O subsystem is the part of the Linux kernel that manages the various devices such as keyboards and mouse, devices usually used by any users. On top of this, it comes the Industrial I/O subsystem (IIO) that are analog to digital or digital to analog convertors, for example **the accelerometer, gyroscopes, proximity sensors** with the aim to extend the I/O subsystem. A typical device falling into the IIO category would be connected via SPI or I2C.

#### 3.4.2 How is LINUX OS interacting with the LEDs?

When previously steps are done, you will see the RGB LED blinking on the board. It is very important to make these configurations before moving on to the next section of the lab.

The SD card contains the image of Ubuntu 12.04 and will boot it on the board at every restart.

#### 3.4.3 Screen

#### 3.4.4 File transfer between host computer and the board

### 3.5 Interact with the RGB LED and the Buzzer

#### 3.6 Webserver

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

#### 3.7 Booting LINUX with System on Chip

#### 3.8 Interfacing the accelerometer + webserver

# 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 "JShare internet connection via ethernet port" 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 convenient 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.e. your laptop), it is possible to connect though ssh:

```
ssh -l root 192.168.2.2
```

## C SKELETON OF THE CODE FOR THE LINUX PART

```
# Skeleton of the laboratory you will implement in the next 2 hours

# DO NOT MODIFY NEXT LINES
from evdev import InputDevice, ecodes
import time
import math
from bokeh.plotting import figure, curdoc
from bokeh.driving import linear
import threading

def ledRGBon():
    # PUT YOUR CODE HERE
    pass

def ledRGBoff():
    # PUT YOUR CODE HERE
    pass

def ledRedOn():
    # PUT YOUR CODE HERE
    pass

def buzzerOn():
    # PUT YOUR CODE HERE
    pass

def buzzerOff():
    # PUT YOUR CODE HERE
    pass

# UTILITY FUNCTION
# DO NOT MODIFY
def checkEmpty(value):
    if not value:
        value = 0
    return value

# UTILITY FUNCTION
# DO NOT MODIFY
def resultantAcc(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

def runAccReadout():
    global G_acc
    acc = InputDevice('/dev/input/event1')

    x = 0
    y = 0
    z = 0

    while True:
        try:
            # PUT YOUR CODE HERE
            pass
```

```

except IOError as e:
    time.sleep(0.01)

# PLOTTING AND THREADING PART
# DO NOT MODIFY AFTER THIS PART

@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:
    print ("Let's kickoff this lab")
    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')

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

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

```

## D CHEATING SHEET - FINAL CODE FOR FPGA PART

## E CHEATING SHEET - FINAL CODE FOR LINUX PART

## **READING LIST**