

THE INTERNATIONAL SCHOOL OF TRIGGER AND DATA ACQUISITION
2017

0.5pt

SYSTEM ON CHIP - FPGA 2pt

December 14, 2016

Contents

1	INTRODUCTION	3
2	IMPLEMENTATION IN HARDWARE	4
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	9
3	IMPLEMENTATION using LINUX OS on FPGAs	15
3.1	Booting LINUX with System on Chip	15
3.2	Interfacing the accelerometer + webserver	15
	Reading List	16

1 INTRODUCTION

This laboratory will 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.

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

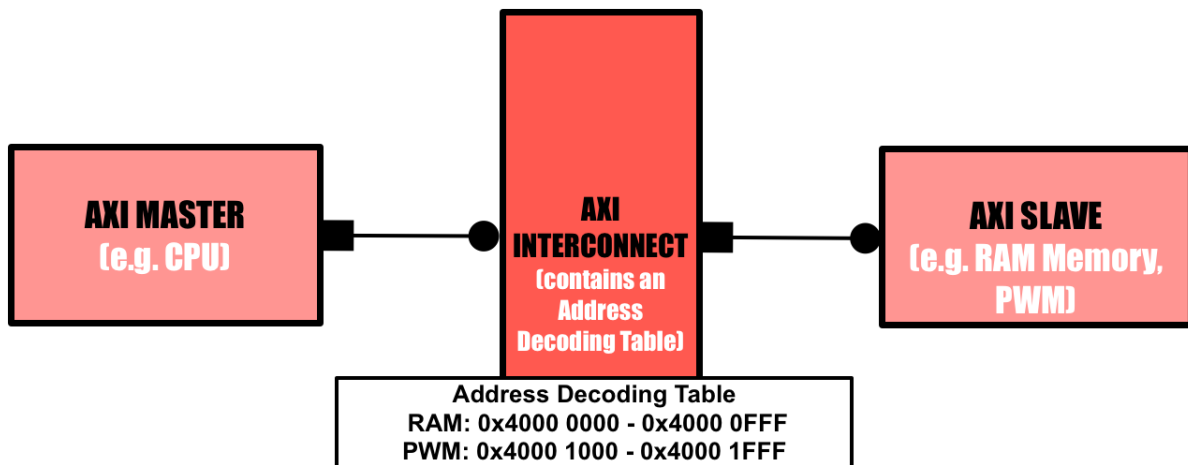
We are going to introduce all the concepts of the block diagram throughout the next pages of the laboratory.

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:



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

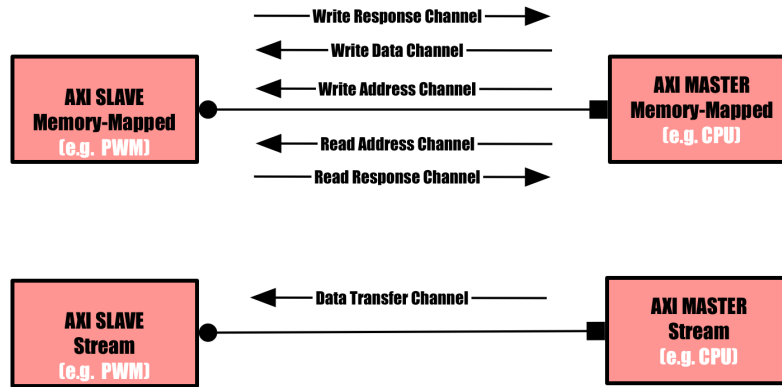


Figure 2: AXI Memory Mapped and AXI Stream architecture

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.

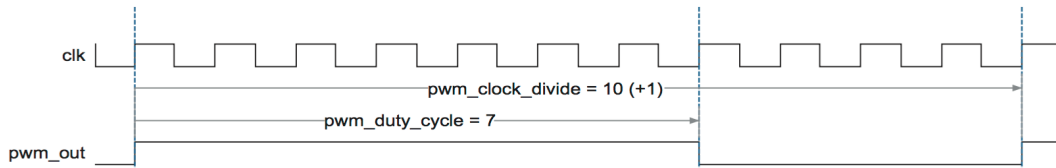


Figure 3: PWM waveform (Source: Application Note 333, Altera)

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.

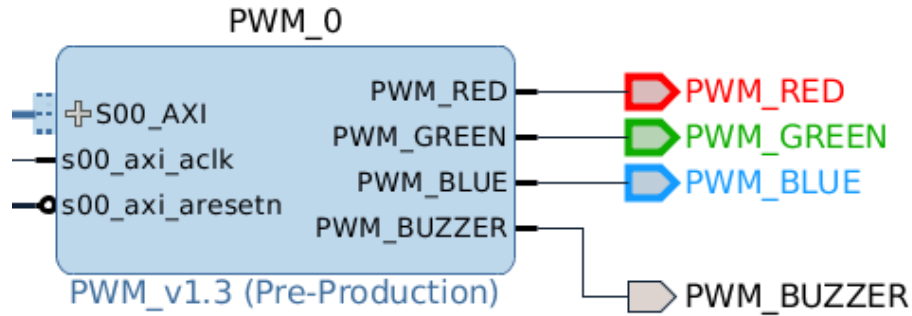


Figure 4: PWM Slave IP

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

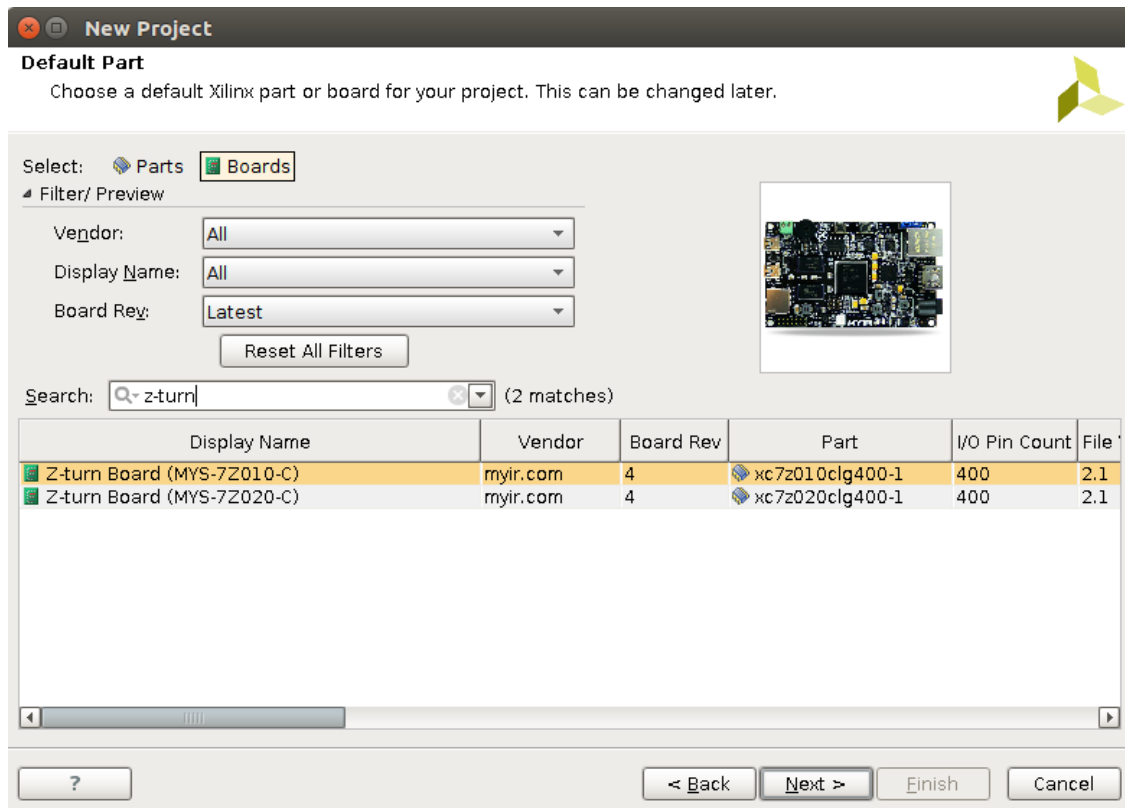


Figure 5: Board Selection

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.

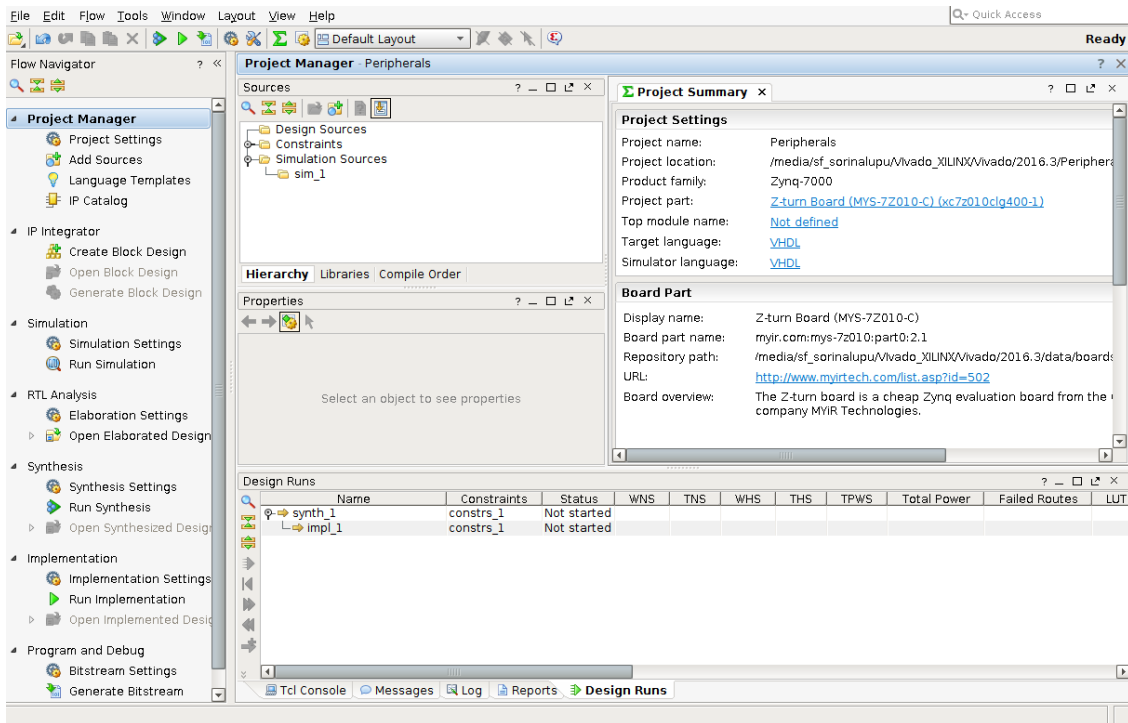


Figure 6: Project Structure

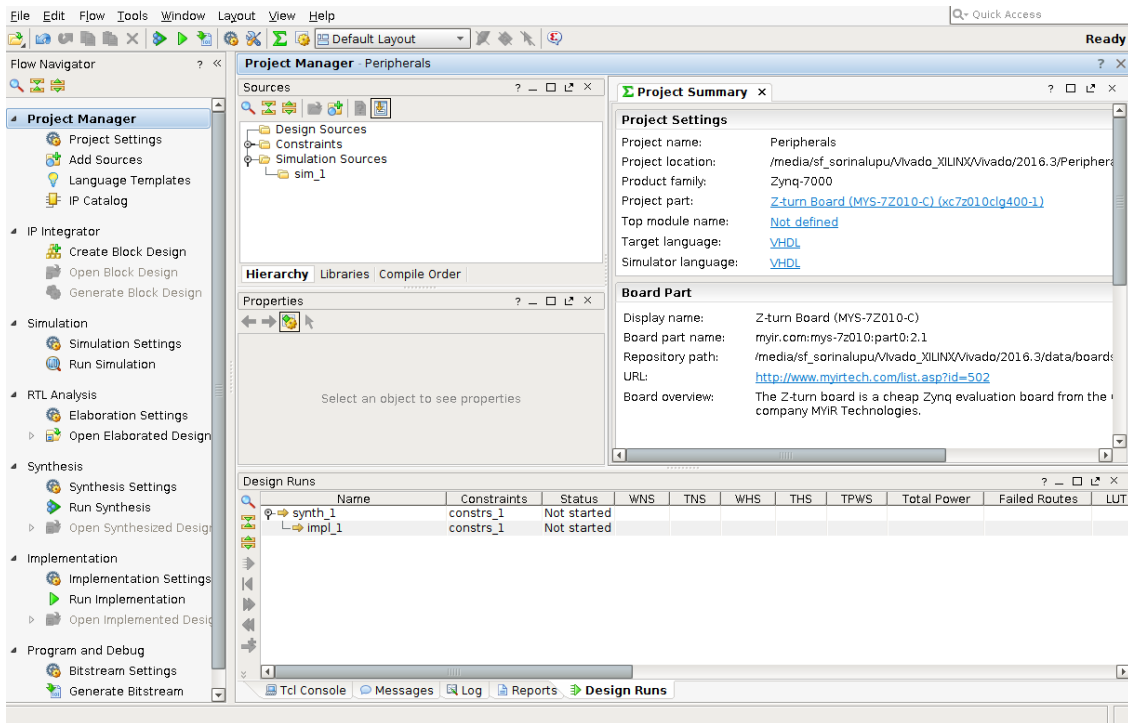



Figure 7: Project Structure

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 Pro-

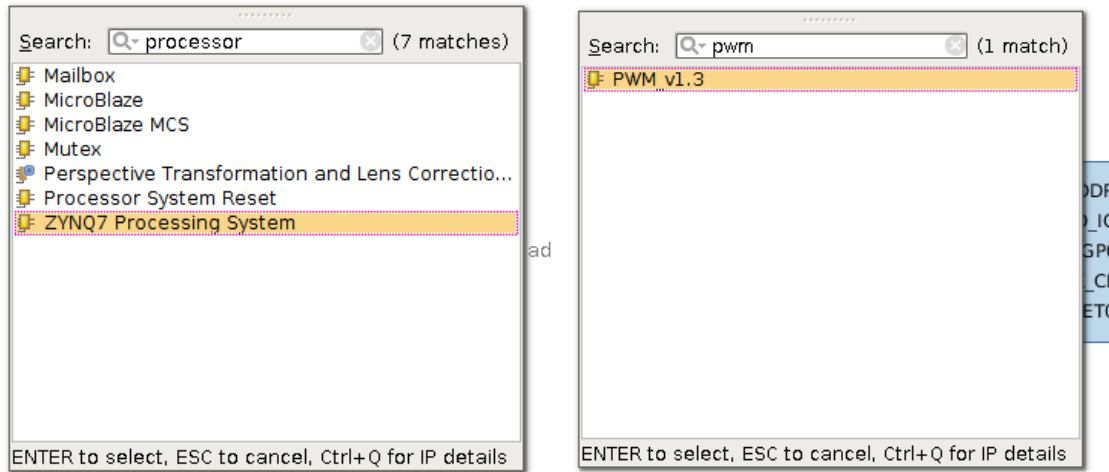



Figure 8: IP Blocks for Master and Slave

cessor 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?

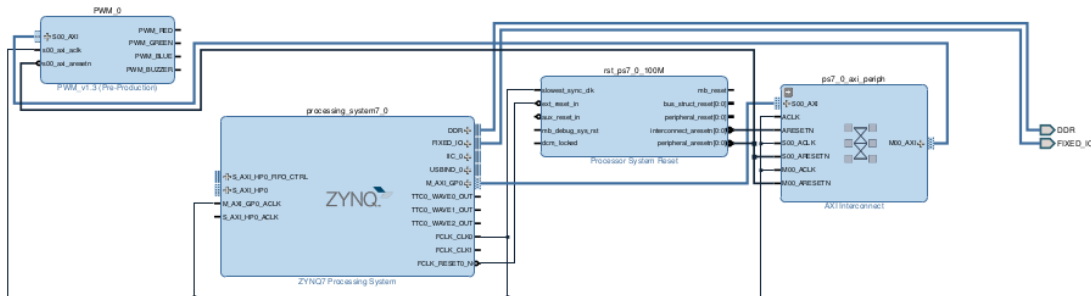


Figure 9: Block Diagram of our System

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* 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:

Check Table 1 for a better understanding of the code.

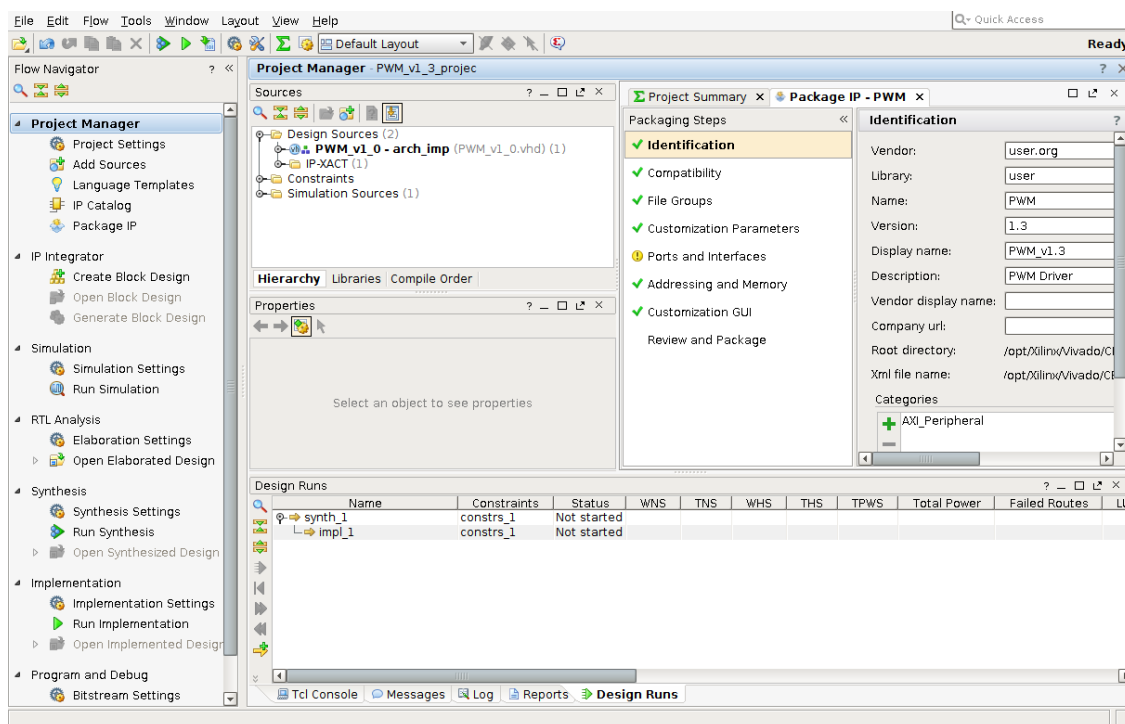


Figure 10: IP Project Structure



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: HP0_ACLK**, we have not configured the Processing System yet.

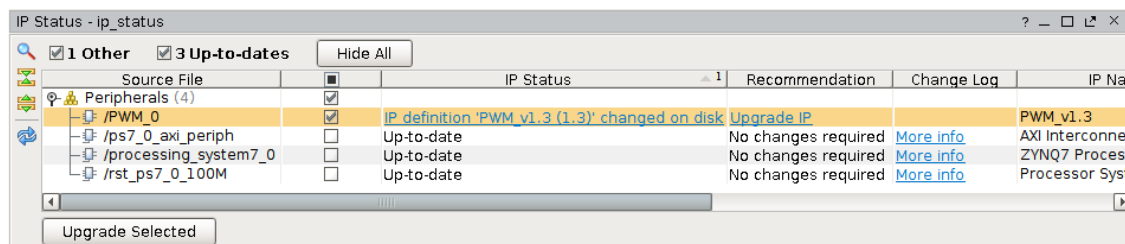



Figure 11: Upgrade IP Window

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/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** 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.



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

12. Let's write some C code

! Get stuck? Ask your assistant for some help.

TO BE ADDED AFTER the talk

3 IMPLEMENTATION USING LINUX OS ON FPGAs

Write the theory needed to run an OS on SoC

Present advantages

3.1 Booting LINUX with System on Chip

3.2 Interfacing the accelerometer + webserver

READING LIST