Tutorial for Implementation of Quadruped Robot's Motion Control on SoC FPGA

Karakasis Chrysostomos
el13136@outlook.com

June 5, 2019

Contents

Chapter 1

Introduction

In this thesis, the Ubuntu 16.04.6 LTS Operating System was utilized, in addition to the following software tools:

- Vivado 2017.4, a software suite for the Hardware Design of the FPGA
- Xilinx Software Development Kit (XSDK) 2017.4, an Integrated Design Environment for the Software Design of the ARM processor
- Minicom, a program for the emulation of the SDK terminal
- Matlab, a computing environment for the visualization of the recorded measurements

In the next chapter, detailed guidelines are included, according to which all necessary programs can be installed and prepared for future use.

Consequently, each one of the four software tools will be analyzed, accompanied by explanatory instructions and step-by-step screenshots.

As depicted in the overall operational diagram (Fig.1.1), initially the user has to design the desired hardware architecture of the platform through Vivado. This includes both the programming of the FPGA, as well as the configuration of the communication between the ARM Processor and the FPGA. Next, via SDK, the user programs the processor and creates the appropriate file, which will be then transferred to the SD Card. Once the SD Card has been inserted to the Zybo Development Board, the execution of the previously configured hardware and software implementations begin, where the interfacing with the user is achieved via Minicom. After the termination of the experiment, all recorded measurements are stored in the SD Card, which can be extracted and inserted in a PC for post processing through Matlab.

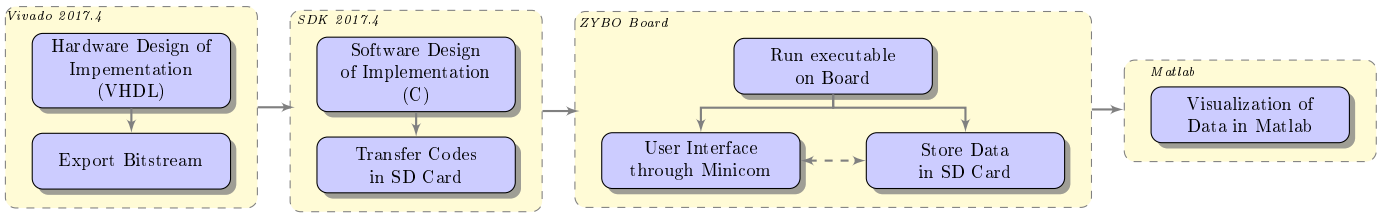


Figure 1.1: Overall Operational Diagram.

Chapter 2

Installing Vivado and Digilent Board Files

In this chapter, an available on line tutorial is quoted from the website of Digilent Company. The actual link is given here*, while the corresponding pdf file is also attached along with this report†, in case the web address is modified in the future. Apart for the installation of the Vivado Design Suite, the Digilent Board Files have to be installed, as they include the corresponding file for the currently employed Zybo Development Board.

Although the cited tutorial refers to the latest version (2018.3), the same procedure can be followed for any version. In our case, the Vivado 2017.4 Design Suite version was installed for Linux and it is also recommended for future users. If the user wishes to download a more updated version, the included source codes may present compatibility errors.

2.1 Install Minicom for SDK

In case the user selects to operate in a Windows Operating System, the Minicom is not necessary. However, due to incompatibility issues with Linux, in order to control and monitor the SDK terminal, Minicom has to be installed. A detailed installation guide can be found here ‡ , while the corresponding pdf file is also attached along with this report §, in case the web address is modified in the future. The same settings were applied in the context of this thesis, as instructed in the given installation guide.

*https://reference.digilentinc.com/vivado/installing-vivado/start
†Installing Vivado and Digilent Board Files.pdf
‡http://processors.wiki.ti.com/index.php/Setting_up_Minicom_in_Ubuntu
§Setting up Minicom in Ubuntu - Texas Instruments Wiki.pdf

Chapter 3

Explanation of Vivado and VHDL Codes

You are highly recommended to experiment with Vivado and VHDL language through several tutorials that exist online by Xilinx, such as *The Zynq Book Tutorials* and *Vivado Design Suite User Guide - Getting Started*, which are included in the **diploma_thesis_codes** folder.

3.1 Creating a new project

Once the installation of the Vivado Design Suite has been completed, we can open the Vivado 2017.4 application and start the design of our implementation. The main start window of Vivado is depicted in Fig.3.1, where we can create a new project or open a recent one.

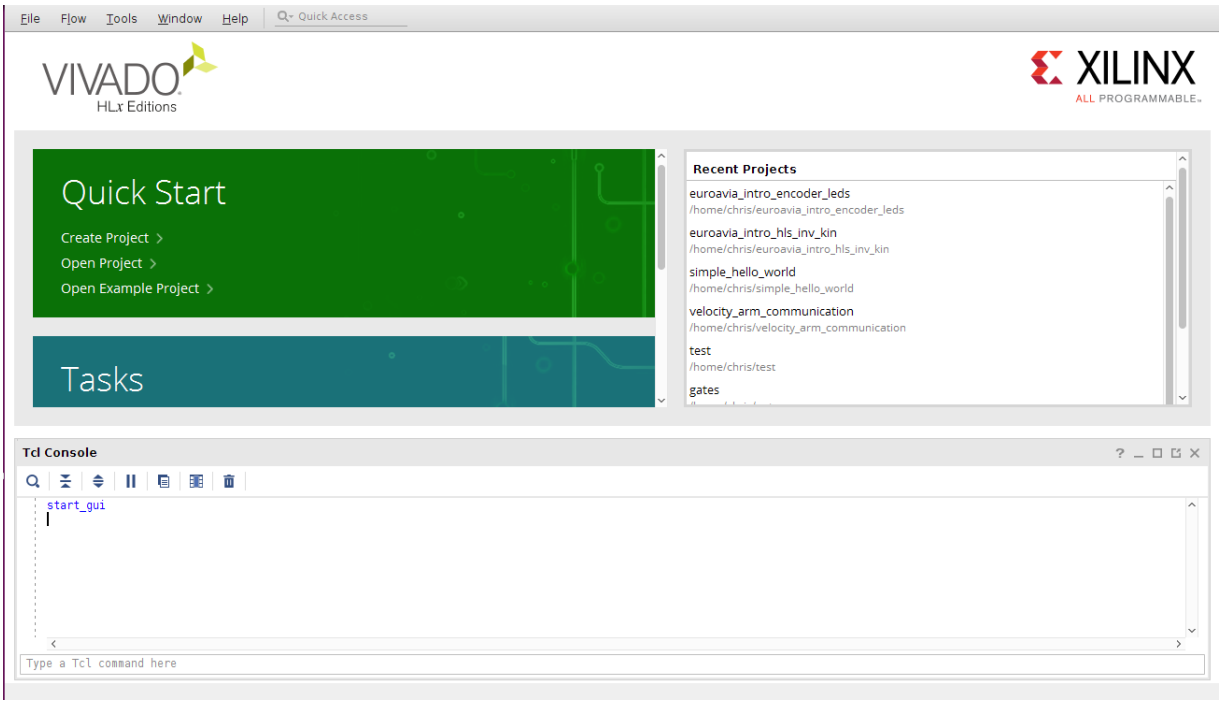


Figure 3.1: The main start window of Vivado.

First we select to create a new project and press "Next" (Fig.3.2).

Figure 3.2: Create a new project.

Next, we have to choose a name and specify the location of the new project. In this case, we will name the project "laelaps_four_legs_tutorial" (Fig.3.2).

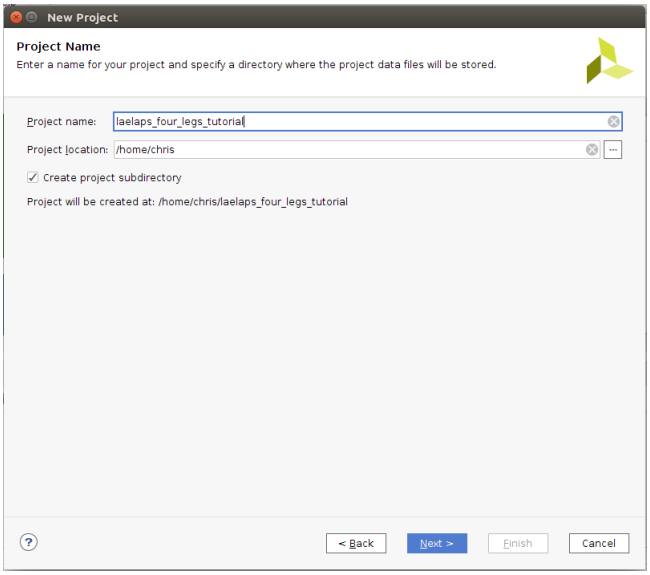


Figure 3.3: Name and location of the new project.

Most of the times, the type of project to create will be "RTL Project", unless the designer wishes to select another option, for example to modify an already created project (Fig.3.4).

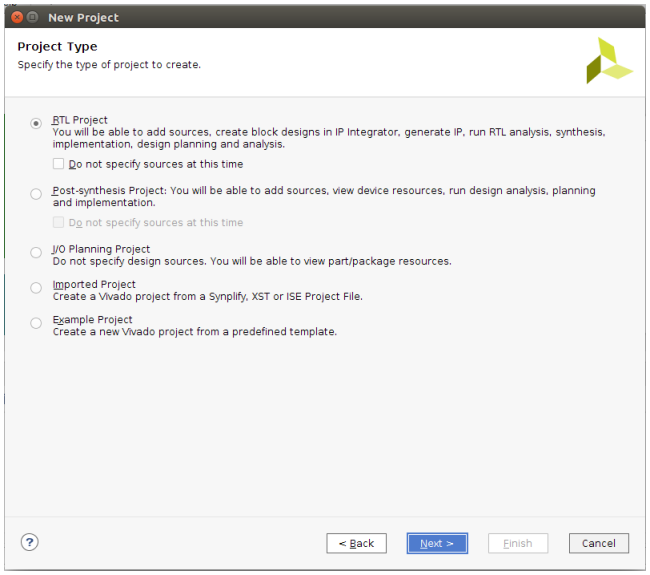


Figure 3.4: Selection of Project Type.

Consequently, in the next window pane (Fig.3.5), we have the ability to upload existing source files such as VHDL codes, IP files, etc. At this point we shall not utilize some existing codes, although this feature is quite useful, when we want to reuse some source codes.

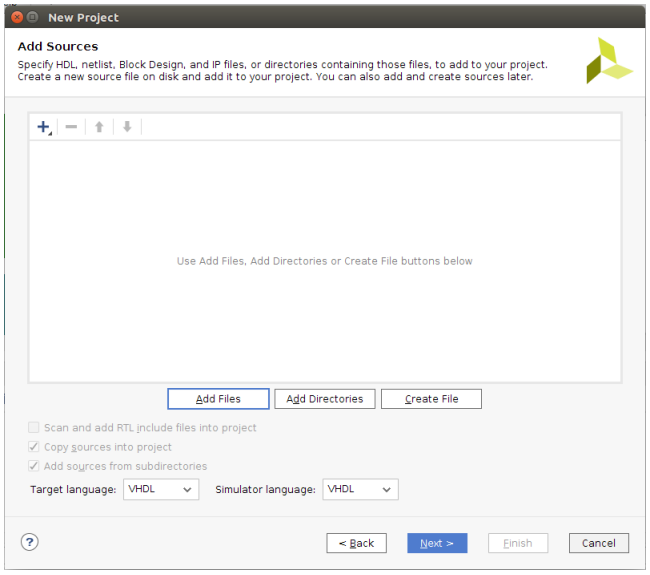


Figure 3.5: Add Sources.

In the "Add Constraints" window (Fig.3.6), we will upload a constraints file, which holds important information regarding the physical ports of the Zybo Development Board such as switches, buttons, LEDS and of course the Pmods. Specifically, through this file it is possible to modify the corresponding name of each port and hence refer to it and utilize it in our design. The file is named "Zybo-Master.xdc" and it is located within the "diploma_thesis_codes.zip" file. Namely, the actual path is: "*diploma_thesis_codes/simple_hello_world/simple_hello_world.srcs/constrs_1/imports/src*". Once it has been selected, press "OK" and then "Next".

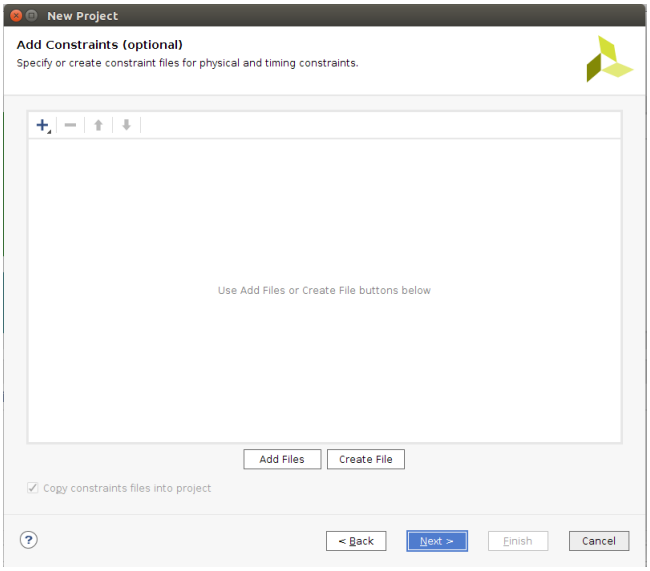


Figure 3.6: Add Constraints.

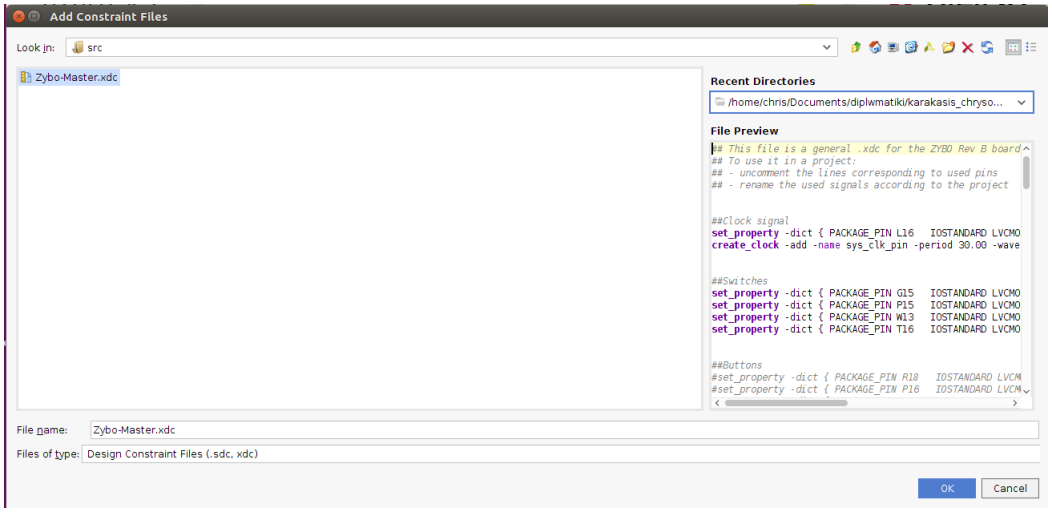


Figure 3.7: Location and Name of the Constraints file.

A crucial step for the creation of a new project is the determination of the board device on which we intend to execute our design (Fig.3.8). In the context of this thesis the Zynq Developement Board (Zybo) was utilized. First, we select "Boards" and then search for the term "zybo". Once the appropriate Display Name is available, we select it and press "Next". If the "Zybo" or "Zybo Z7-10" options are not available, there is probably something wrong with the installation of the Digilent Board Files, therefore refer to Chapter 2.

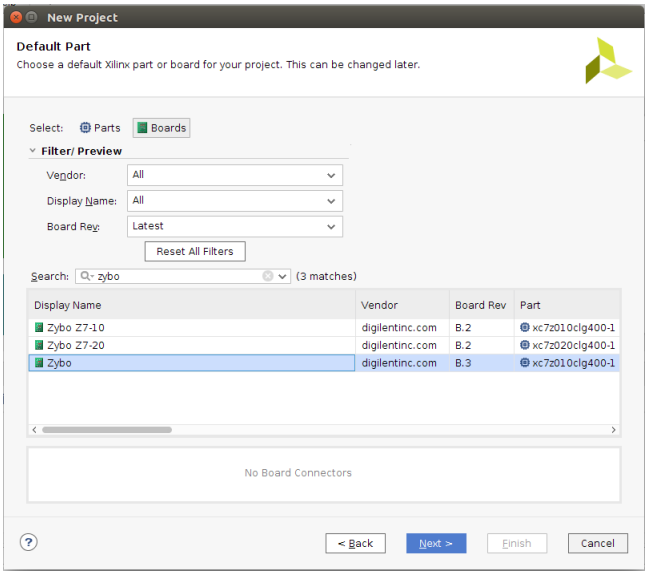


Figure 3.8: Selection of Default Part.

Finally, the settings for the new project are summarized in one window, where we can inspect them for any errors (Fig.3.9). If everything is in order we press "Finish" and otherwise we return to the previous stages via the "Back Button".

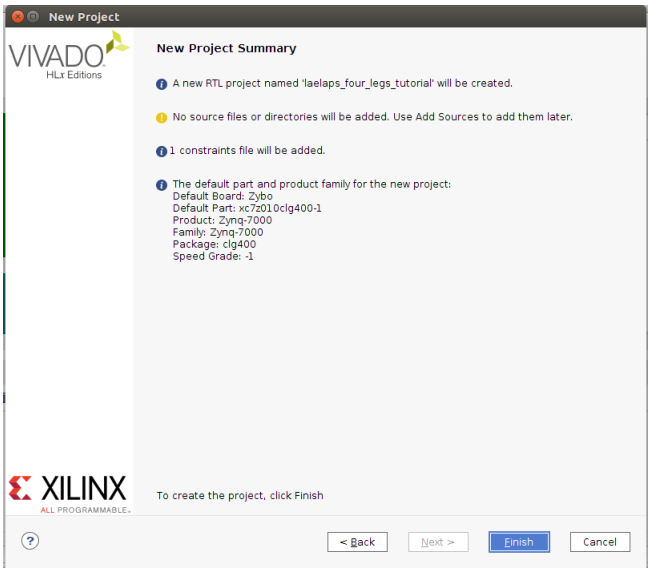


Figure 3.9: New Project Summary.

After several seconds the Vivado Project Manager will open, where the programming of our implementation will take place (Fig.3.10).

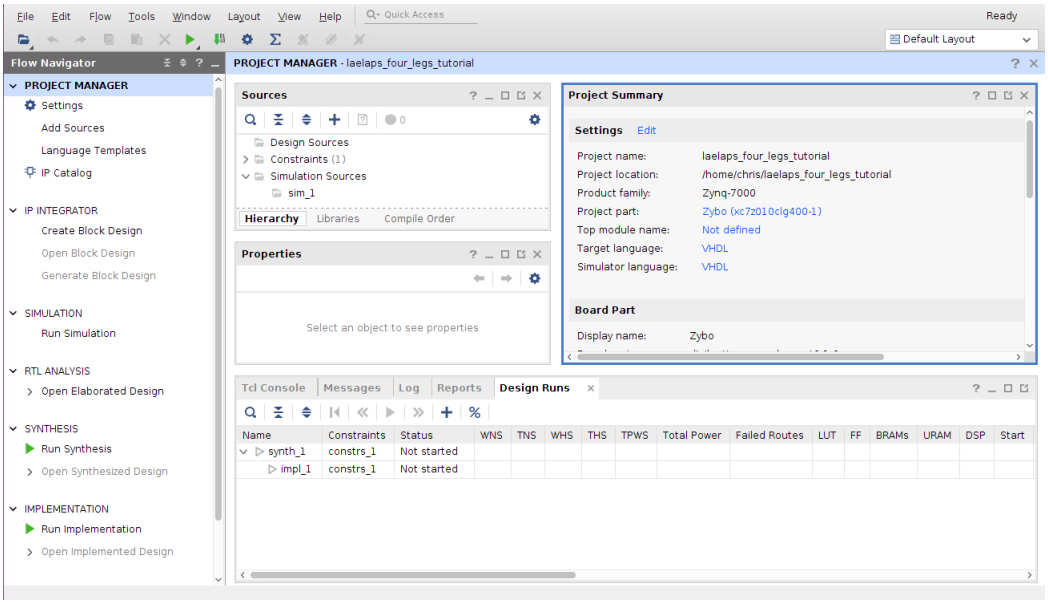


Figure 3.10: Project Manager.

The first step of our implementation is the creation of the *laelaps_four_legs_qei_vel_pwm_1.0* Custom IP, which is responsible for the processing of the encoders' signals, the determination of their positions and velocities, the creation of PWM signals, as well as the communication with the ARM Processor via the AXI4-Lite Interface. In the context of this tutorial, we will create a duplicate of the Custom IP that was realized during the thesis. In order to do so, we select "Tools" and then click on the "Create and Package New IP..." option (Fig.3.11).

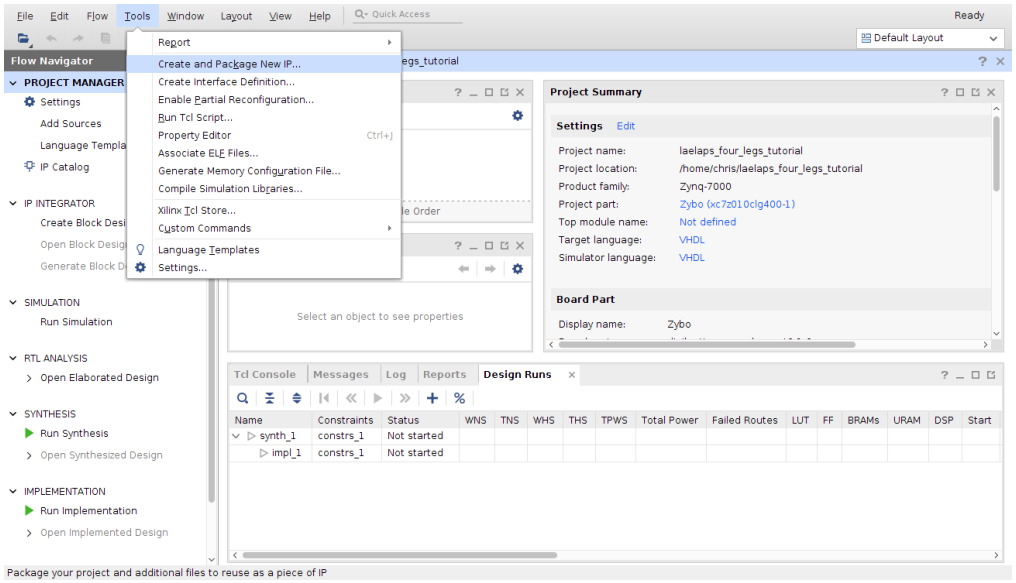


Figure 3.11: Create and Package New IP.

This will open up a new window (Fig.3.12), which informs us about the abilities of this feature. We select "Next" and move on to the Fig.3.13, where we select the "Create a new AXI4 peripheral" option and click "Next".

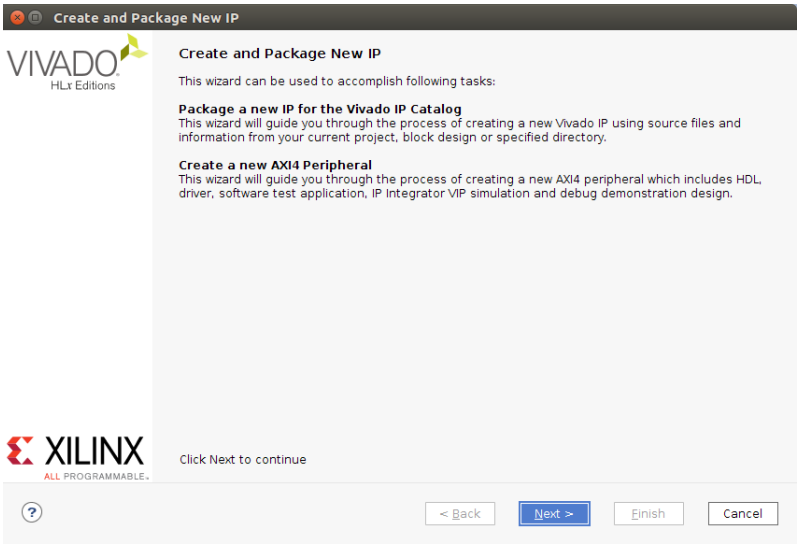


Figure 3.12: Create and Package New IP First Window.

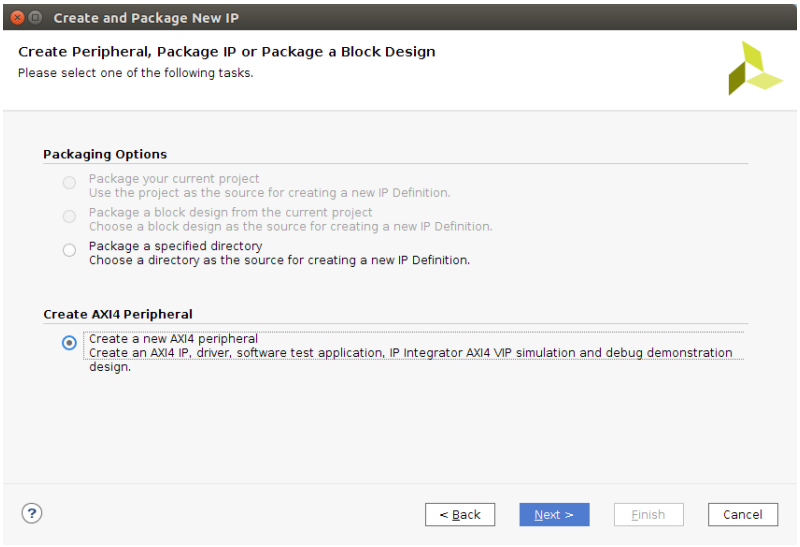


Figure 3.13: Create and Package New IP Second Window.

In Fig.3.14) we define the name of the new custom IP (*laelaps_four_legs_duplicate_ip*) and specify the location, where it will be stored later. Normally, all IPs are stored inside the *ip_repo* folder. If it has not been automatically created, it is strongly recommended to create it and store there all IPs, either custom made or downloaded. Once we have completed all fields, we click "Next".

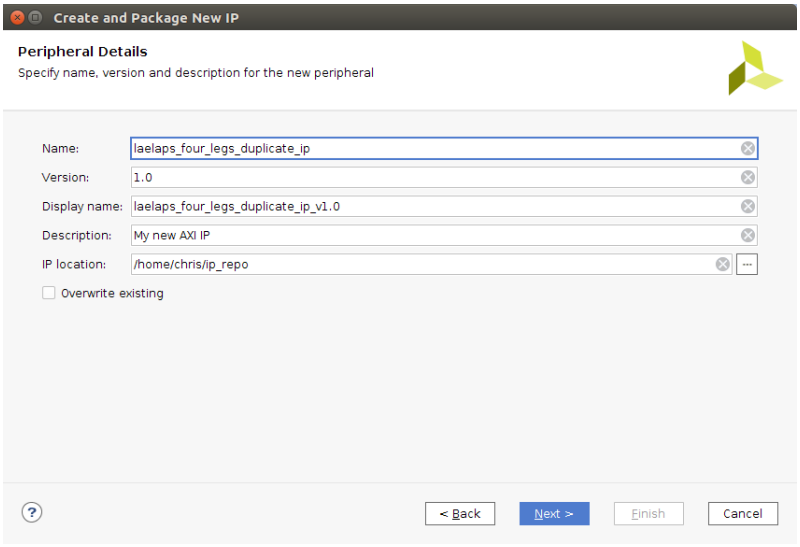


Figure 3.14: New Custom IP Name.

In the "Add Interfaces" window (Fig.3.15), we determine the characteristics of the AXI4 Interface that our Custom IP will support. In this case, an AXI4-Lite Interface will be created, due to the fact that the size of data to be transmitted is rather low. Additionally, for reasons of simplicity, our implementation dedicates three 32-bit slave registers for each joint, one for the motor's velocity, one for the encoder's position and the velocity's sign and one for the corresponding PWM signal. As a result, for four legs and hence 8 joints, 24 32-bit slave registers are required. Once we have completed all fields, we click "Next".

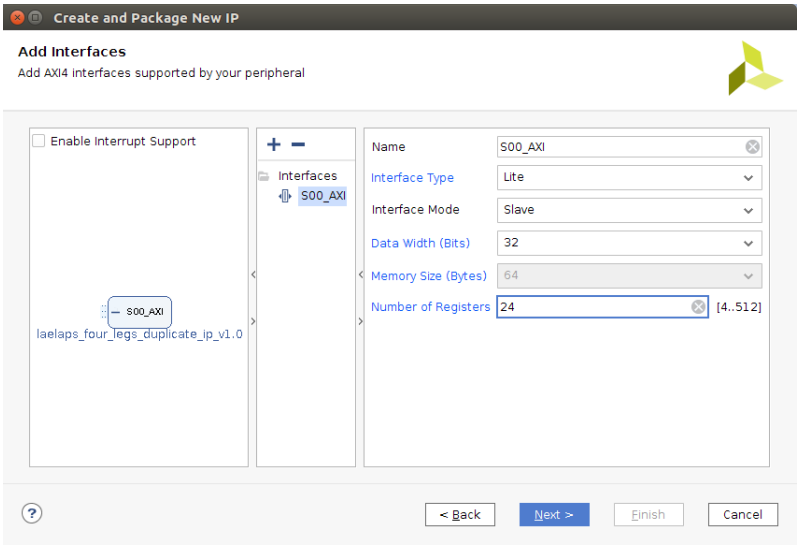


Figure 3.15: New Custom IP - Add Interfaces.

In the last "Create and Package New IP" window (Fig.3.16), our choices are summarized and we select the next step of our design. Since we will modify the IP and insert our design, the "Edit IP" bullet is selected and then we click "Finish".

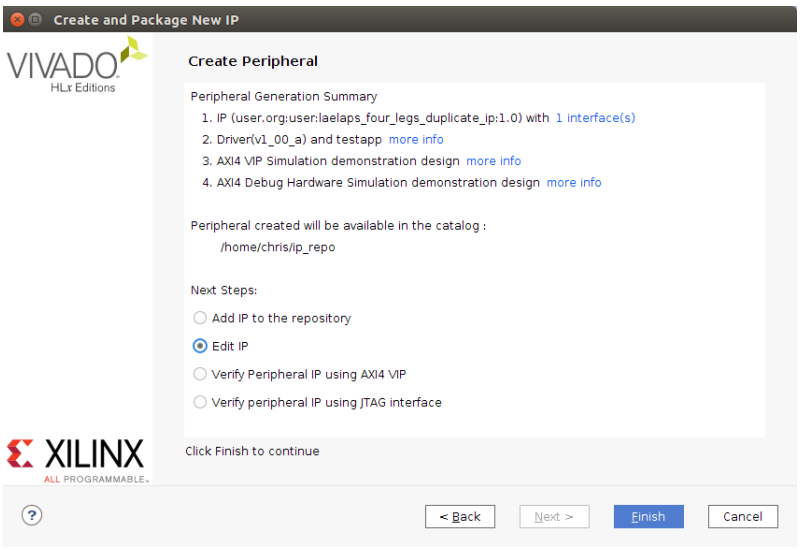


Figure 3.16: New Custom IP - Summary.

After a few seconds, a new Vivado project, named *edit_laelaps_four_legs_duplicate_ip_v1_0*, will open (Fig.3.17), through which the programming of the Custom IP can be achieved.

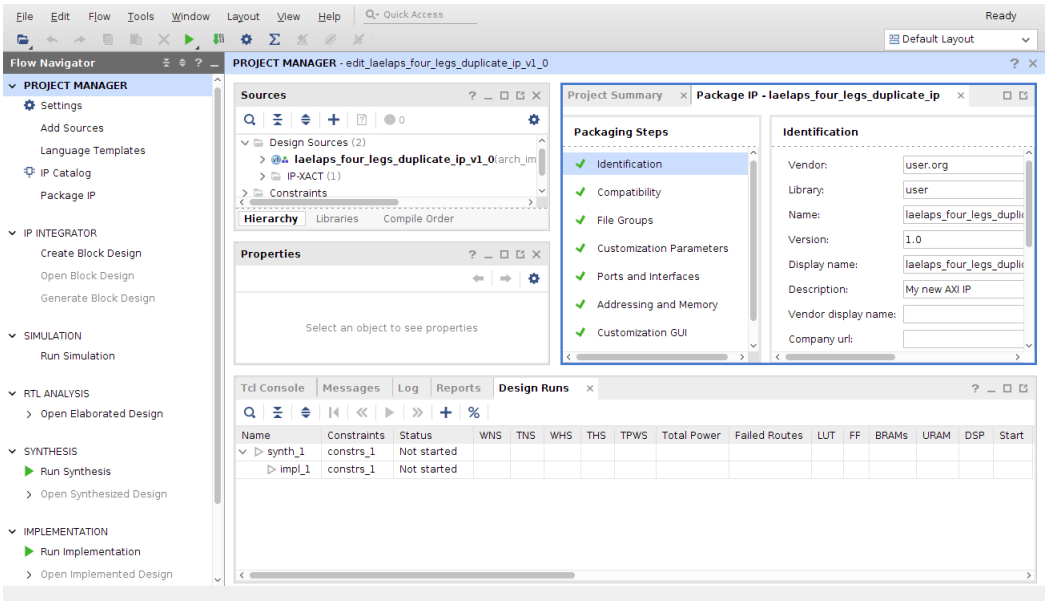


Figure 3.17: Project Manager for New Custom IP.

In the *Sources* pane (Fig.3.18), you should see two HDL source files:

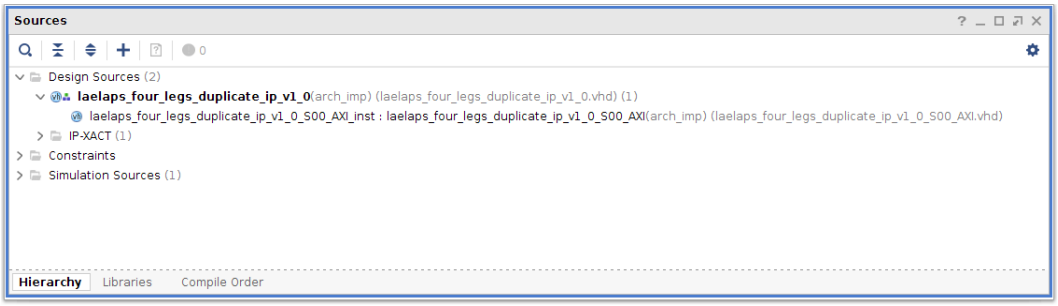


Figure 3.18: Source Pane of New IP.

As we specified our target language as VHDL in Fig.3.5 earlier, the template files have been generated in VHDL. Had we specified Verilog as the target language, Verilog source files would have been created. The two source files are:

- **laelaps_four_legs_duplicate_ip_v1_0** — This file instantiates all AXI-Lite interfaces. In this case, only one interface is present.
- **laelaps_four_legs_duplicate_ip_v1_0_S00_AXI.vhd** — This file contains the AXI4-Lite interface functionality which handles the interactions between the peripheral in the PL and the software running on the PS.

The *IP Packager* pane will also be open in the Workspace:

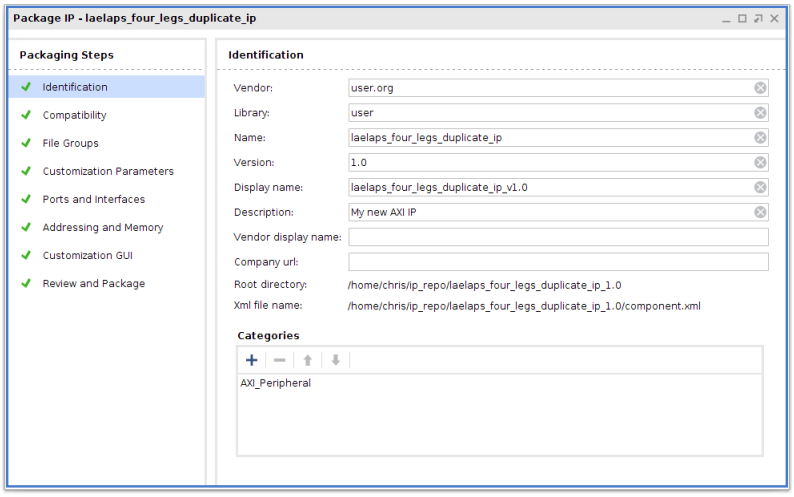


Figure 3.19: IP Packager.

The information that we specified about our peripheral will be visible.

We can now add the functionality to our **laelaps_four_legs_duplicate_ip** peripheral. We will be adding several source HDL files, which will be utilized as components in our peripheral.

In order to do that, we click the + symbol in the *Sources* pane.

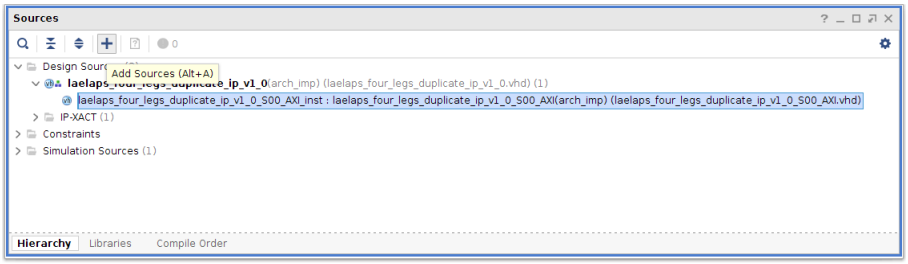


Figure 3.20: Add Sources in the New Custom IP.

A new menu opens up, where we have to select whether we want to add or create constraints, design sources, and simulation sources. In this case, since we wish to add design sources, the second choice is selected and the "Next" button is clicked.

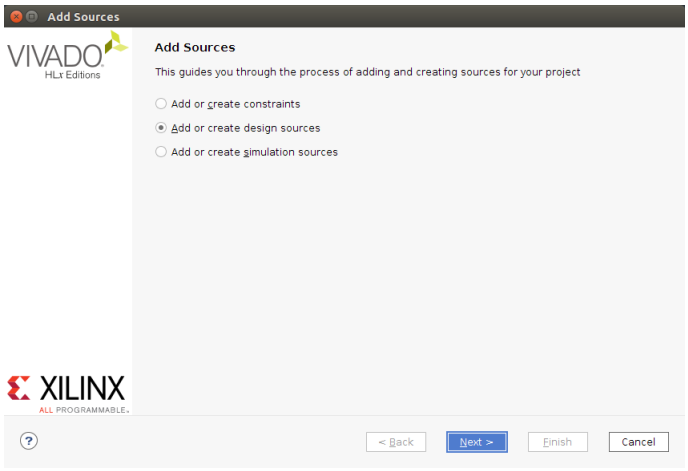


Figure 3.21: Add Sources in the New Custom IP - Second Figure.

Next, we click the "Add Files" button, which enables us to browse through our files and select which ones we want to integrate to our design. Specifically, the following source files will be selected, which can be found here: *diploma_thesis_codes/laelaps_four_legs_qei_vel_pwm_1.0/src*.

- *encoder.vhd*, responsible for the position calculation using the encoders' signals
- *filter.vhd*, responsible for the filtering of the encoders' outputs
- *pwm_freq.vhd*, responsible for the creation of PWM signals
- *vel_qei.vhd*, responsible for the velocity estimation
- *qei.vhd*, the top module which integrates and synchronizes the *encoder*, *filter*, *vel_qei* components

If we wanted to create a new source file, we would choose the "Create File" option, which enables us to determine the file's characteristics such as its name and type.

Once all source files have been selected (Fig.3.22), we click "Finish" and return back to Project Manager window.

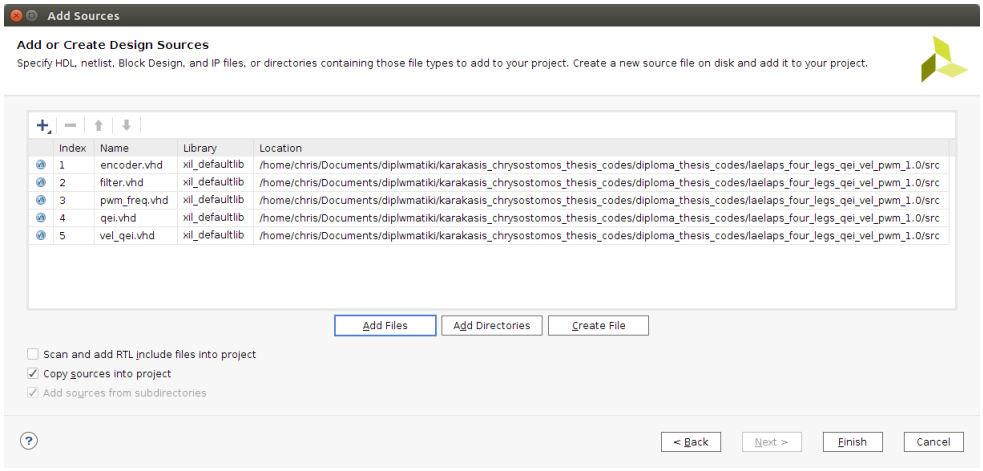


Figure 3.22: Add Sources in the New Custom IP - Add Files.

Now the following state should be visible (Fig.3.23), where the *qei* and *pwm_freq* units have been included. As you can observe, the *filter*, *encoder* and *vel_qei* units are integrated inside the *qei* block. This feature will be discussed promptly, as we explain the functionality of each unit.

Figure 3.23: Add Sources in the New Custom IP - Third.

3.2 Analysis of the Hardware Source Codes

As we mentioned above, the purpose of the Custom IP is to manage the calculation of the position and the velocity for each motor, the creation of PWM signals, as well as the communication with the ARM Processor. The first three tasks are been realized via corresponding hardware source codes written in VHDL.

First of all, the ***filter*** code is responsible for the stabilization of the system's inputs [*]. In particular, the block receives as input a 1-bit signal and if it is stable then the block exports the signal as output. By using a fast clock, the system takes three samples of the input and if all three samples are in accordance with each other, the signal is considered stable. In this application the input signal is one of the three signals generated from the encoders (A, B, Index). Secondly, the ***encoder*** code emulates a Quadrature Encoder Interface (QEI) as described in the Diploma Thesis [†]. Thirdly, the ***vel_qei*** code realizes the velocity estimation unit of our implementation, as described in the Diploma Thesis [‡] .

Now, the ***qei*** block is responsible for the connection, synchronization and management of the previous three units, which it utilizes as components. Initially, it creates a four times slower clock signal than the one it operates with. Therefore, we have two clock signals, one fast and one slow. Hereupon, it receives as inputs the unfiltered signals generated from the encoder (A, B and Index) and supplies them as inputs to three ***filter*** components. Each one of them operates according to the fast clock signal, whose frequency will be specified later, and outputs the stable and filtered encoder's signal. Consequently, the filtered signals are supplied as inputs to the ***encoder*** unit, along with the slow clock signal, while its generic variables are being defined at this point. It is important to note that the values assigned here bypass any other declaration inside the ***encoder*** code. It is quite similar to calling a C function and specifying its arguments. Then the ***encoder*** block outputs the motor's direction and position signals, which are being forwarded as inputs to the ***vel_qei*** block, along with the slow clock signal. Additionally, the position signal is linked with one of the ***qei*** block's outputs. Finally, the velocity's value and sign are being computed and exported as outputs of the ***qei*** block, along with the aforementioned position signal.

Now, due to the fact that the *qei* block utilizes three *filter*, one *encoder* and one *vel_qei* components, we can see those instances entailed under the *qei* unit in the *Sources Pane* in Fig.3.23.

Regarding the ***pwm_freq*** code, as described in the Diploma Thesis[§], it is in charge of the creation of PWM signals for the control of the power supplied to the quadruped's motors.

In total, the ***qei*** and ***pwm_freq*** are the top modules that will also be utilized as components for the ***laelaps_four_legs_duplicate_ip_v1_0_S00_AXI*** entity, which will additionally configure the communication between the FPGA and the ARM Processor.

All corresponding source codes along with helpful comments can be found inside the *laelaps_four_legs_duplicate_ip* IP through this link:

diploma_thesis_codes/laelaps_four_legs_duplicate_ip_1.0/src

[*]Refer to Section 4.1 of the Diploma Thesis (Page 49)
[†]Refer to Section 4.2 of the Diploma Thesis (Page 50)
[‡]Refer to Section 4.3 of the Diploma Thesis (Page 52 and especially Page 54)
[§]Refer to Section 4.5 of the Diploma Thesis (Page 57)

3.3 Configuration of the Custom Made IP

3.3.1 *laelaps_four_legs_duplicate_ip_v1_0_S00_AXI*

For the configuration of the ***laelaps_four_legs_duplicate_ip_v1_0_S00_AXI*** source file, we basically modify the code in specific regions destined for the User, such as Lines 8 and 19 in Fig.3.24.

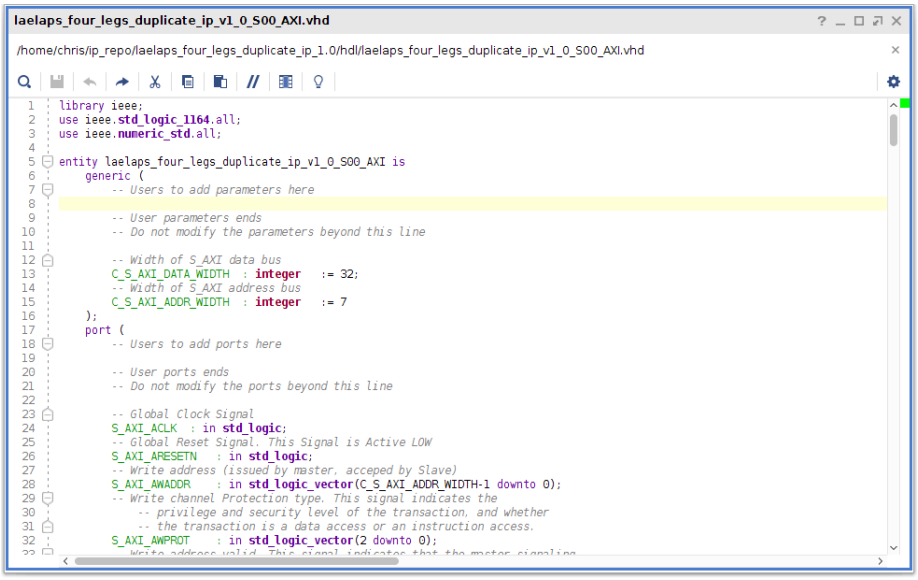


Figure 3.24: Regions where the User is allowed to insert new code.

Initially, we specify the generic variables of our IP (Fig.3.25) and add the necessary ports (Fig.3.26).

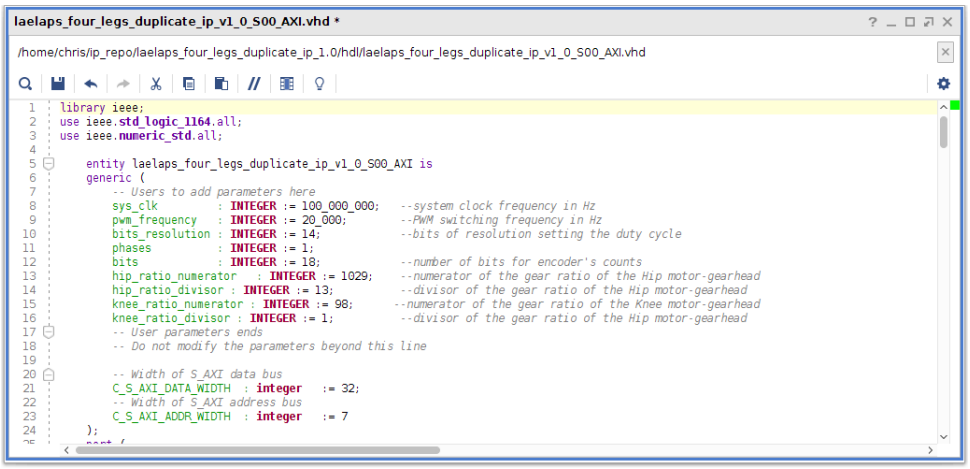


Figure 3.25: Adding the necessary Generic Variables.

The ports correspond to the Pmod Ports through which the reading of the encoders' signals and the transmission of the PWM signals will be achieved. The users can name those ports anyway they desire, but in this case we named them after some specific keywords that existed in the constraints file we added earlier (Fig.3.7) that will be elaborated in a later section (3.3.5). In addition, two sets of ports are included that represent the Zybo Development Board's switches and LEDs.

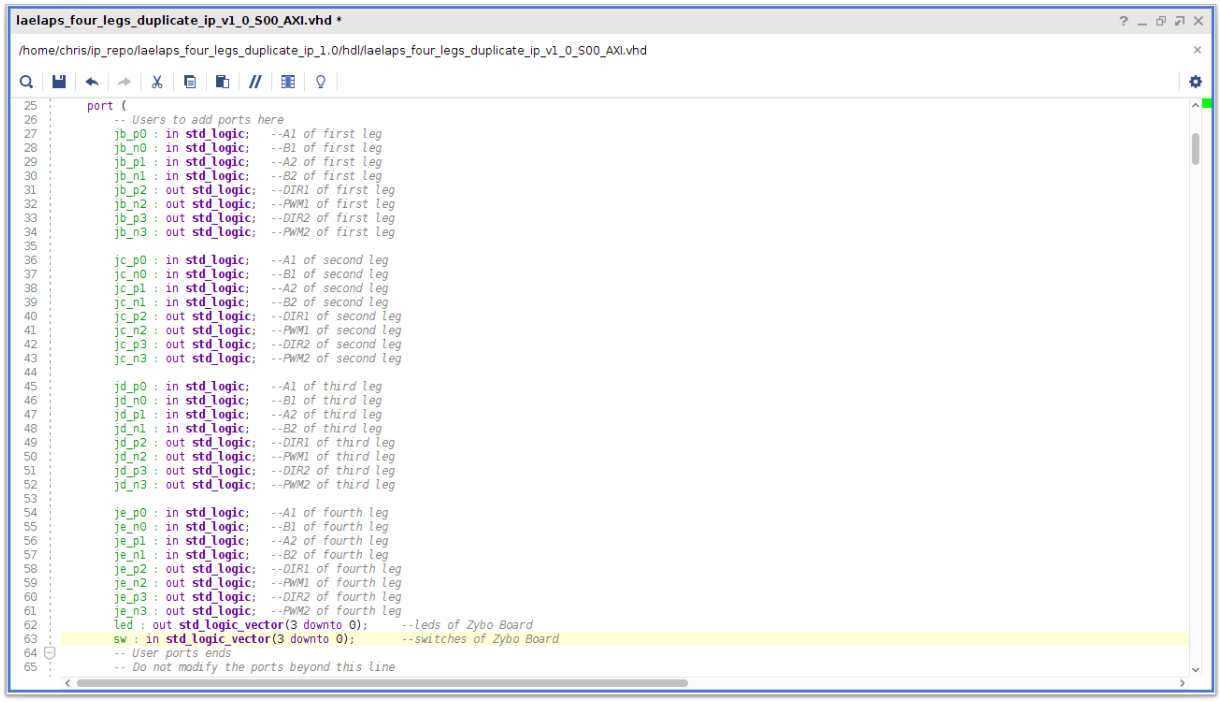


Figure 3.26: Adding the necessary User Ports.

Afterwards, we specify any signals we want to utilize in our design, between the Lines 151-153 (Fig.3.27).

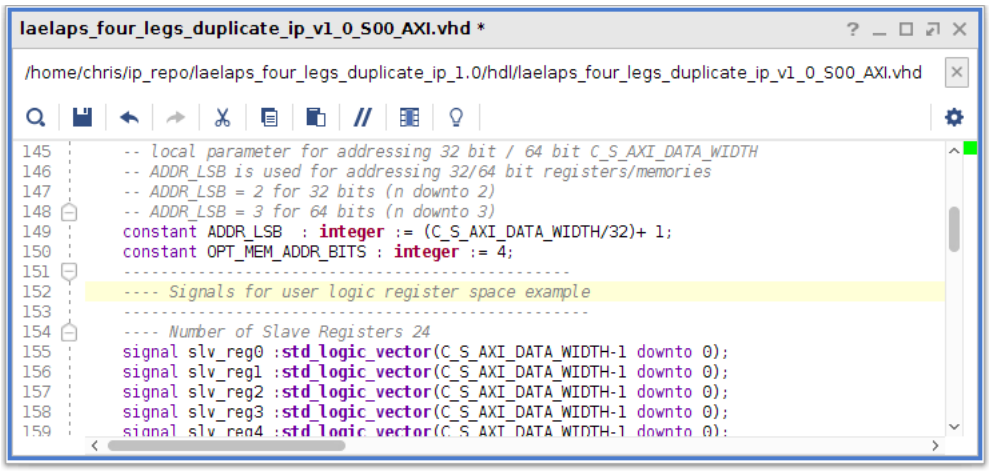


Figure 3.27: Adding the necessary User Signals.

Specifically, for each leg we define two sets of three signals for each joint (one for the position, one for the velocity's value and one for its sign) and two signals for the PWM of each joint's motor (Fig.3.28). In total eight signals for each leg.

Figure 3.28: User Signals for the Fours Legs

Moreover, some auxiliary signals are defined, which will serve as temporary variables (Fig.3.29).

Figure 3.29: User Auxiliary Signals.

Subsequently, we declare the components that we plan on utilizing as they are have been defined in the corresponding source files (Fig.3.30). In this case, we will include the ***qei*** and ***pwm_freq*** files, which were explained previously.

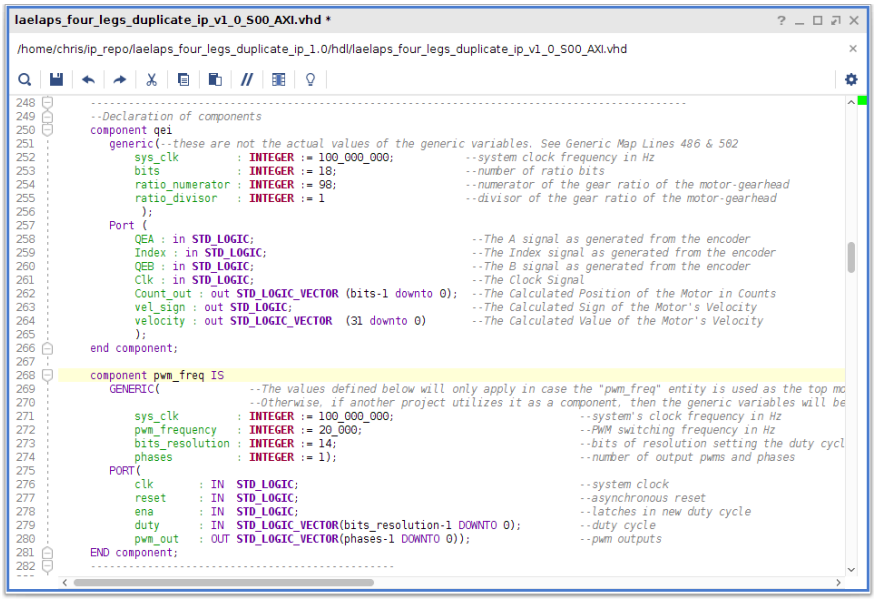


Figure 3.30: Declaration of Components.

A large part of the remaining source code must remain unmodified, as it is vital to the internal functionality of the AXI4-Lite Interface. In fact, the next modification occurs after Line 737, where we either supply the slave registers with the signals we wish to send to the ARM Processor, or read data from them that were transmitted there by the Processor. However, the values of those signals will be set later in the code, together with the utilization of the slave registers designated for reading.

In order to explain how the slave registers work, we will examine the signals and registers assigned for the management of the first leg, since the same logic applies for the those destined for the rest of the legs (Fig.3.31).

As we had configured our IP in the beginning, 24 32-bit slave registers exist in total, each recognized with a corresponding binary expression. The "0000" code represents the first slave register, the "0001" code represents the second one, and so on.

As a result, in lines 750-752 the first slave register is configured, which in this case will be a sending register and transmit data from the FPGA to the ARM Processor. The first *bits* bits are set with the Position in counts of the first encoder of the first leg, where the generic variable *bits* is equal to 18 in our implementation. The 19th bit is set with the sign of the velocity for the first leg's first motor, while the remaining bits are not utilized and hence set to zero.

The second slave register ("0001") is also a sending register and is set with the value of the velocity for the first leg's first motor. As a remainder, in our implementation the velocity signal is a 32-bit signal and hence we have to employ all the bits of this slave register. This is the reason why we chose to send the velocity's sign through the previous slave register.

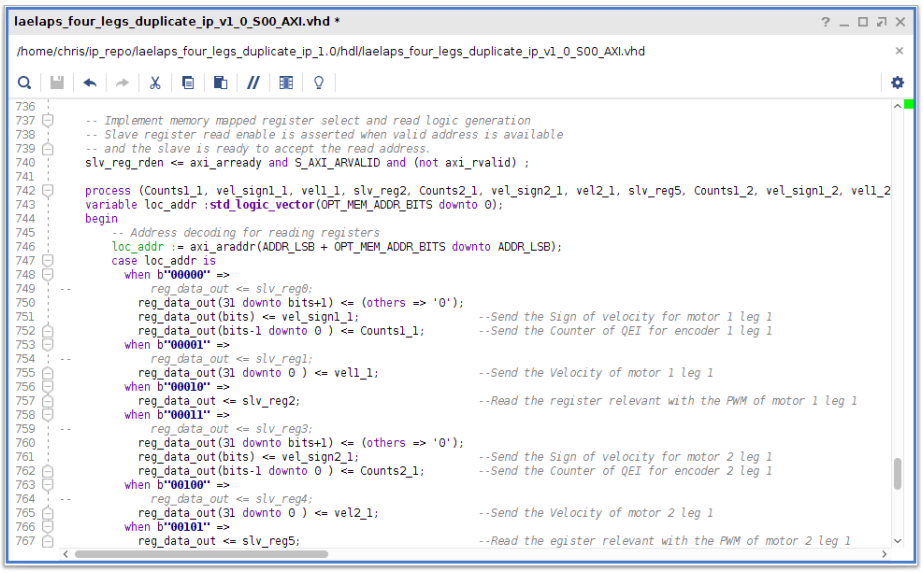


Figure 3.31: Configuration of Slave Registers of AXI4-Lite Interface.

The third slave register ("0010") is a reading register, which transfers information assigned in the Processor's domain. Basically, this signal is relevant to the creation of the PWM signal for the first leg's first motor. Some of its bits will determine the enable and reset signals and the Duty Cycle of a ***pwm_freq*** component, while one of its bits stipulates the Direction Signal of the Motor.

Likewise, the same logic applies for the next three slave registers, which are designated for the first leg's second motor, while the overall procedure is repeated for the remaining legs. In summary, the signals supplied to the sending registers will be computed via a fitting number of ***qei*** components, while the reading registers will determine the inputs of corresponding ***pwm_freq*** components.

The call of the ***qei*** and ***pwm_freq*** components occurs after Line 852, where the user logic can be added (Fig.3.32).

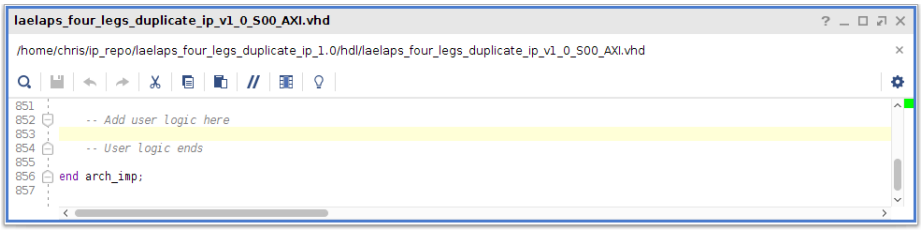


Figure 3.32: Inclusion of the user logic.

Specifically, 8 ***qei*** components are deployed, one for each joint of the quadruped. Once their generic variables are configured inside the *generic map*, the previously defined signals, which correspond to the Pmod Ports, are supplied as the inputs QEA and QEB, while the Index input is set to zero and hence disabled. The S_AXI_ACLK is the clock signal of our system, whose frequency can be specified later. Eventually, the three outputs of each component is linked with the intermediate signals that feed the sending slave registers (Fig.3.33).

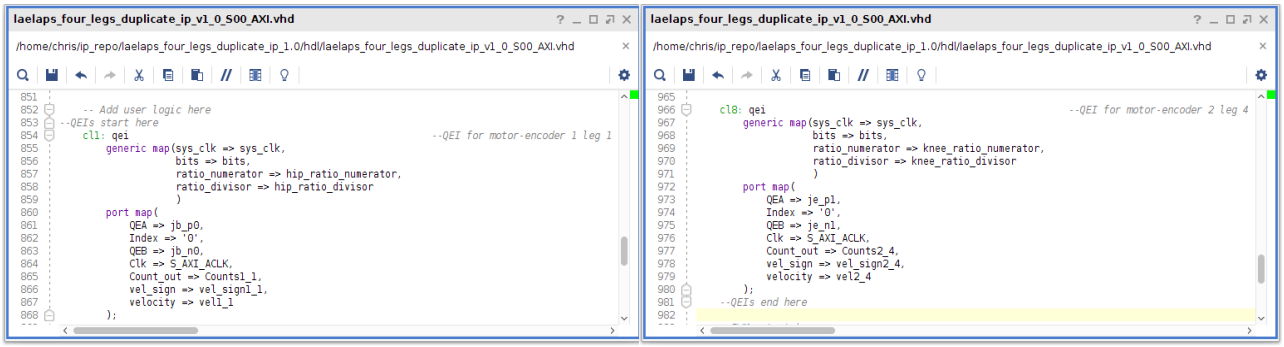


Figure 3.33: The first and last *qei* components

Ultimately, 8 ***pwm_freq*** components are employed, each responsible for the creation of each motor's corresponding PWM signal. At first, their generic variables are configured, while they receive as inputs the fitting bits of the reading slave registers that we discussed above. As outputs, we have the PWM signals, which together with the the third bits of the respective reading slave registers (Direction Signals) are redirected to the Pmod ports of the system that serve as outputs and will connect to the motors' drivers (Fig.3.34).

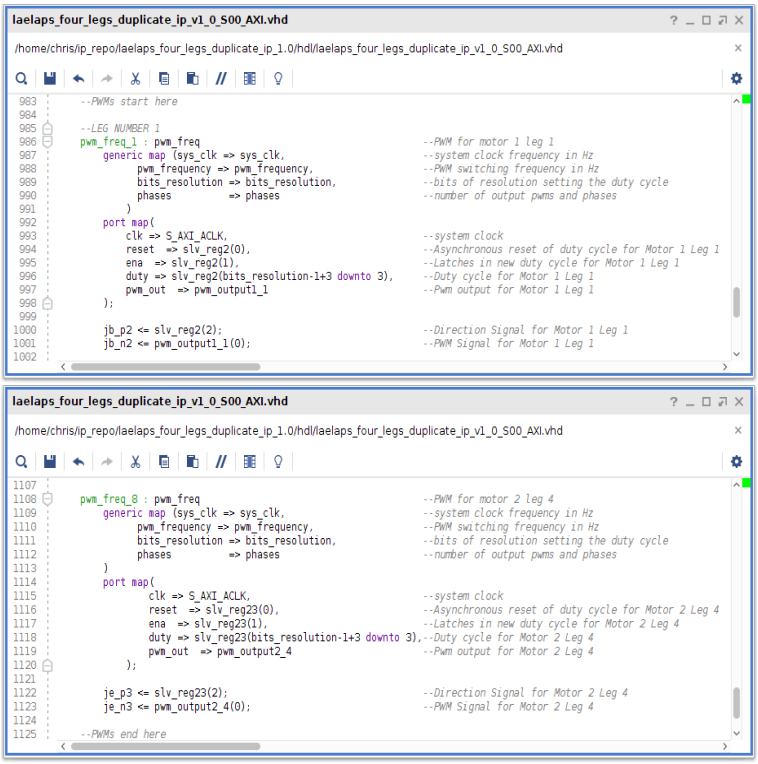


Figure 3.34: The first and last *pwm_freq* components

The last modifications inside the ***laelaps_four_legs_duplicate_ip_v1_0_S00_AXI*** source file will be now elaborated, although they are not vital to the functionality of the implementation. For supervision purposes, we chose to redirect the PWM and Direction signals to the LEDs. However, due to the fact that the Zybo Board is equipped with only four LEDs, a mechanism is required to select which signals we wish to see at a given time to the LEDs. Therefore, a simple multiplexer was constructed, which according to the switches state, controls which signals are displayed. If the first switch is on and the rest off, the LEDs display the PWM and Direction signals of the first leg. Likewise, if the second switch is on and the rest off, the LEDs display the PWM and Direction signals of the second leg, and so on. The first two LEDs show the PWM and Direction signals of the leg's first motor, while the last two represent the PWM and Direction signals of the leg's second motor (Fig.3.35).

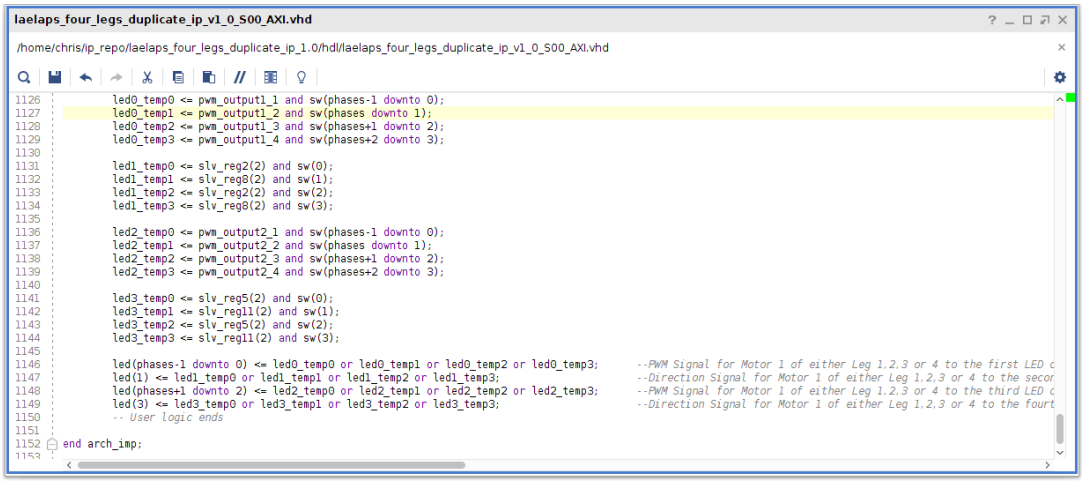


Figure 3.35: Switches-LEDs Multiplexer.

At this point, we save our changes in this source file and return to the *Sources* Pane, where we can observe that the ***qei*** and ***pwm_freq*** units are not independent, but rather they are now located under the *laelaps_four_legs_duplicate_ip_v1_0_S00_AXI* project. In fact, eight qei and eight pwm_freq components are depicted, in correspondence to our changes in this source file (Fig.3.36).

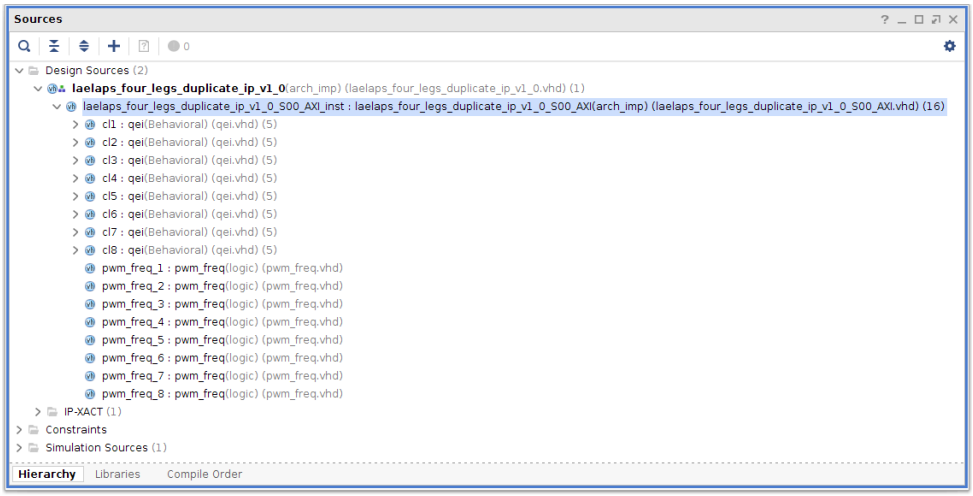


Figure 3.36: Updated Source Pane.

3.3.2 *laelaps_four_legs_duplicate_ip_v1_0*

Last but no least, this source file serves as the top module our custom IP, which instantiates the AXI4-Lite Interface and defines all generic variables we have utilized so far.

Similarly to the previous source file, first we include this module's generic variables (Fig.3.37) and ports (Fig.3.38), which are exactly the same with the previous source file. However, since this is the top module, the values set in the generic variables now will broadcast to all the components we have described. Therefore, by changing the values here, we can control the generic variables in the whole implementation.

The system's clock frequency is set to 83333333 Hz, due to the fact that this frequency was found suitable to our design for reasons that will be explained later. The motors currently in use in Laelaps II and their drivers operate with PWM signals around the frequency of 20kHz and hence it is an acceptable value. The bits of resolution for the Duty Cycle were proved to be sufficient, although higher values could also be used. Since our PWM signals are required to have only one phase, the *phases* variable is set to "1". The 18 bits of precision for the encoders' counts were chosen in order for our system to support the current gear ratios of the motors, as explained in section 4.2 of the Diploma Thesis. On the other hand, the last four generic variables correspond to the current gear ratios, whose values are mentioned in subsection 3.4.4 of the Diploma Thesis. As a result, if the motors change in the future, the only thing that is required to be modified is these four variables.

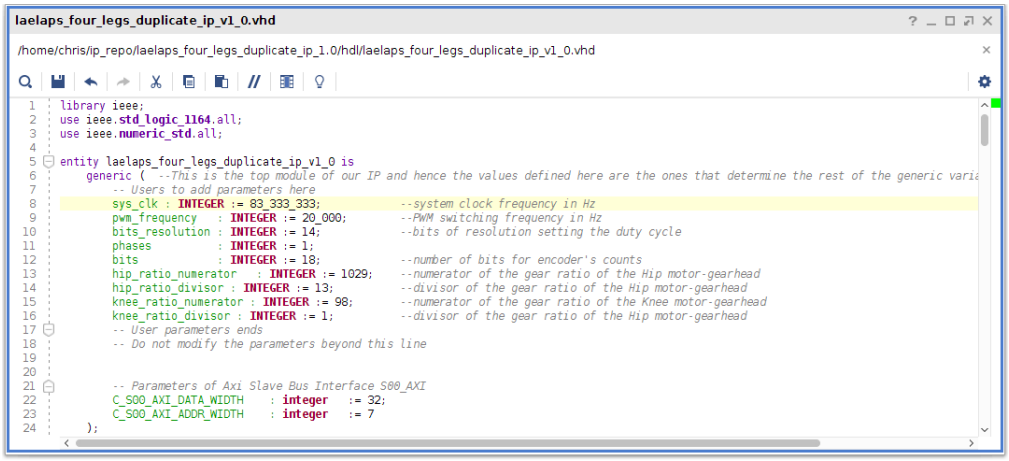


Figure 3.37: Top Module's Generic Variables.

Following that, we update the declaration of the *laelaps_four_legs_duplicate_ip_v1_0_S00_AXI* component, by adding its generic variables (Fig.3.39) and ports (Fig.3.40).

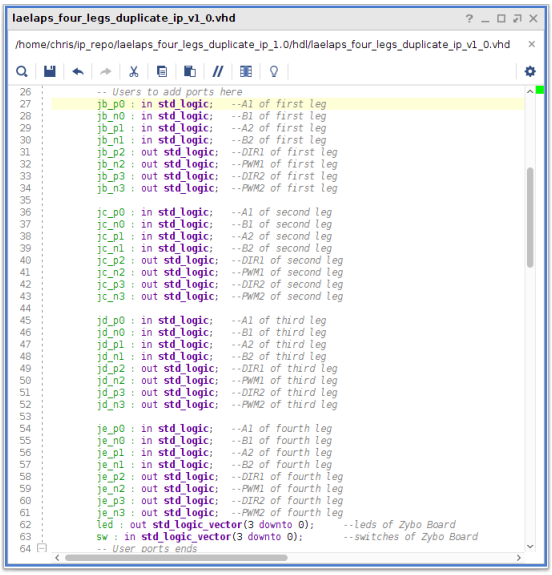


Figure 3.38: Top Module's Ports.

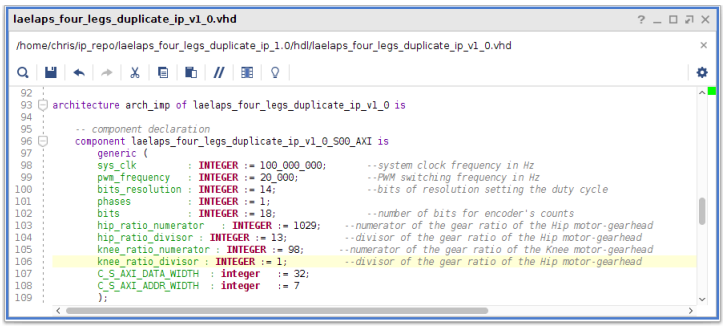


Figure 3.39: Top Module's Component Declaration - Generic Variables.

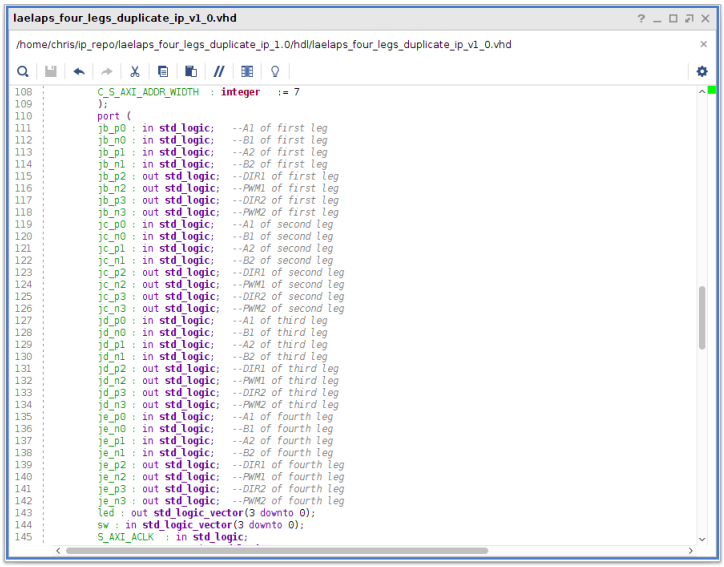


Figure 3.40: Top Module's Component Declaration - Ports.

Lastly, as with every component, the generic and port map fields must be completed, where we set the values of the component's generic variables (Fig.3.41) and ports (Fig.3.42).

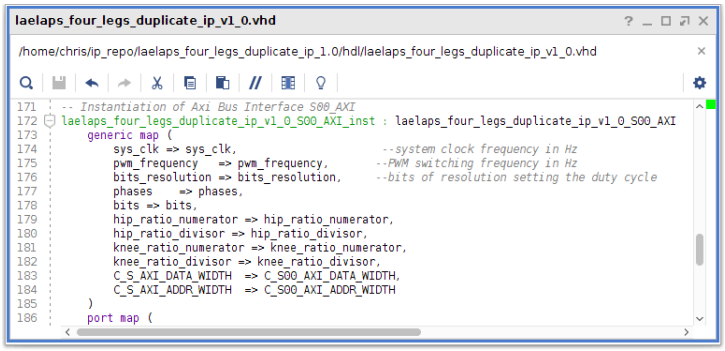


Figure 3.41: Top Module's Component Generic Map.

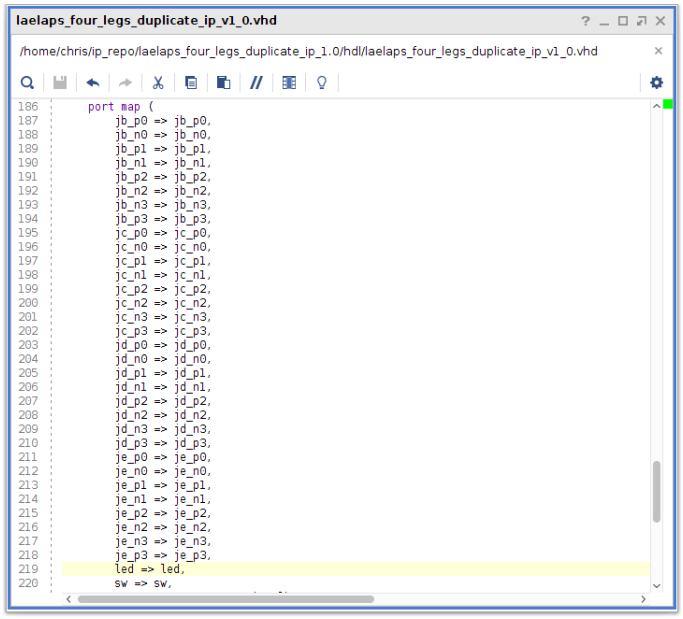


Figure 3.42: Top Module's Component Port Map.

With this, all modifications are now complete and we are ready to package our IP.

3.3.3 Packaging the IP core

Now click on *Package IP* in the Flow Navigator (Under the PROJECT MANAGER tab) and you should see the Package IP tab (Fig.3.43). First we click in the *File Groups* tab, where we select the **Merge changes from File Groups Wizard** option (Fig.3.44).

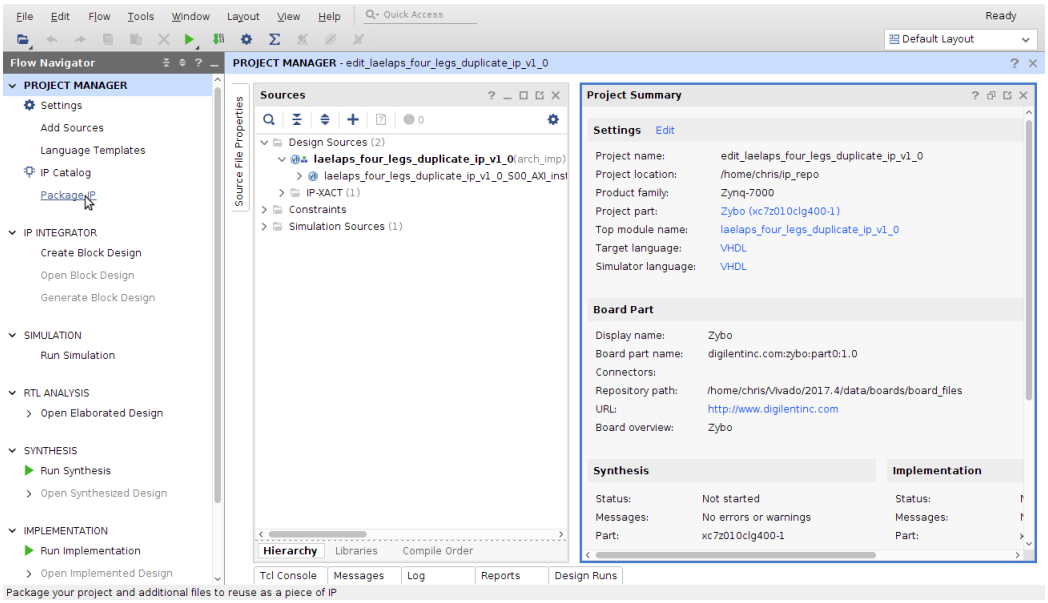


Figure 3.43: Click on Package IP.

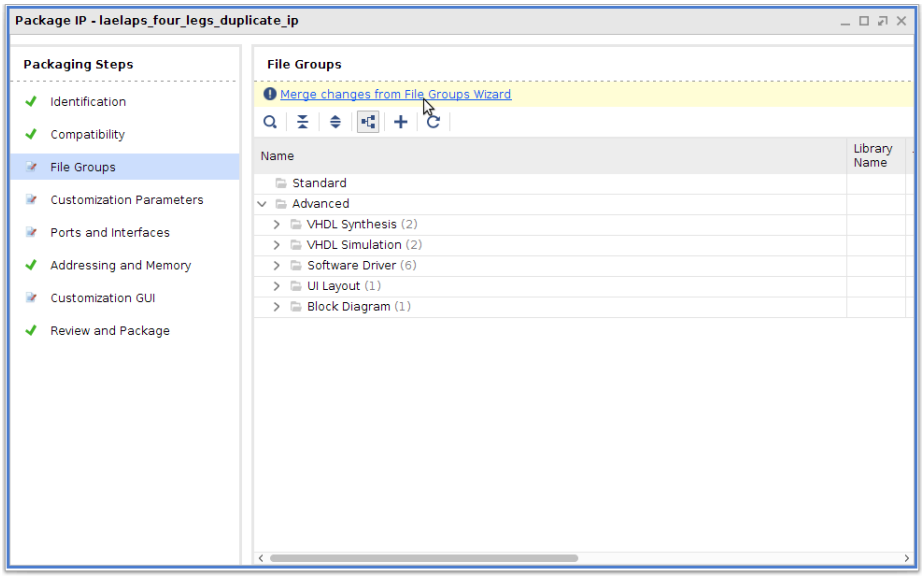


Figure 3.44: Merge changes from File Groups Wizard.

The same procedure is repeated for the *Customization Parameters*, *Ports and Interfaces* and *Customization GUI* tabs, where we select the corresponding **Merge changes from ... Wizard** option.

Finally, inside the *Review and Package* tab, we click the **Re-Package IP** option, which will finalize our Custom IP.

In the end, if warning messages appear inside the *File Groups* tab, relevant with the directory of the source files (Fig.3.45), then you can follow some steps mentioned later to correct this issue.

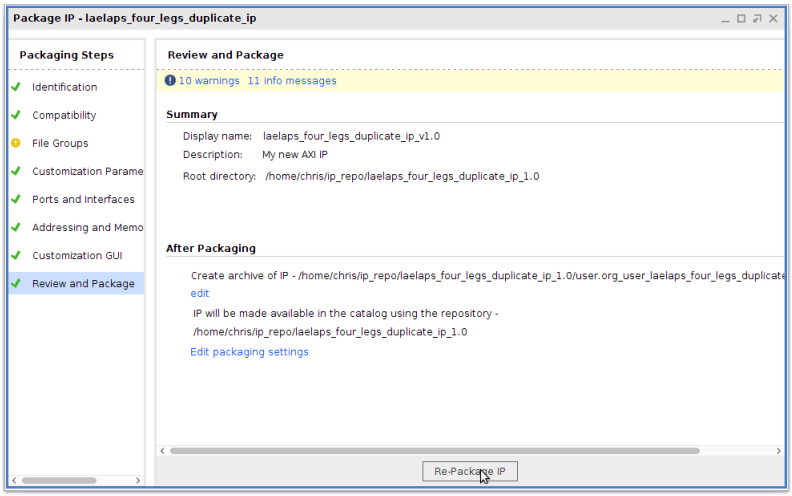


Figure 3.45: Warning Messages.

We close the ***edit_laelaps_four_legs_duplicate_ip_v1_0*** project and return back to the original ***laelaps_four_legs_tutorial*** project. Inside the **Flow Navigator**, under the **IP INTEGRATOR** category, we select the **Create Block Design** option. This will open a Block Design Diagram, where we can add several IPs and useful blocks (Fig.3.46).

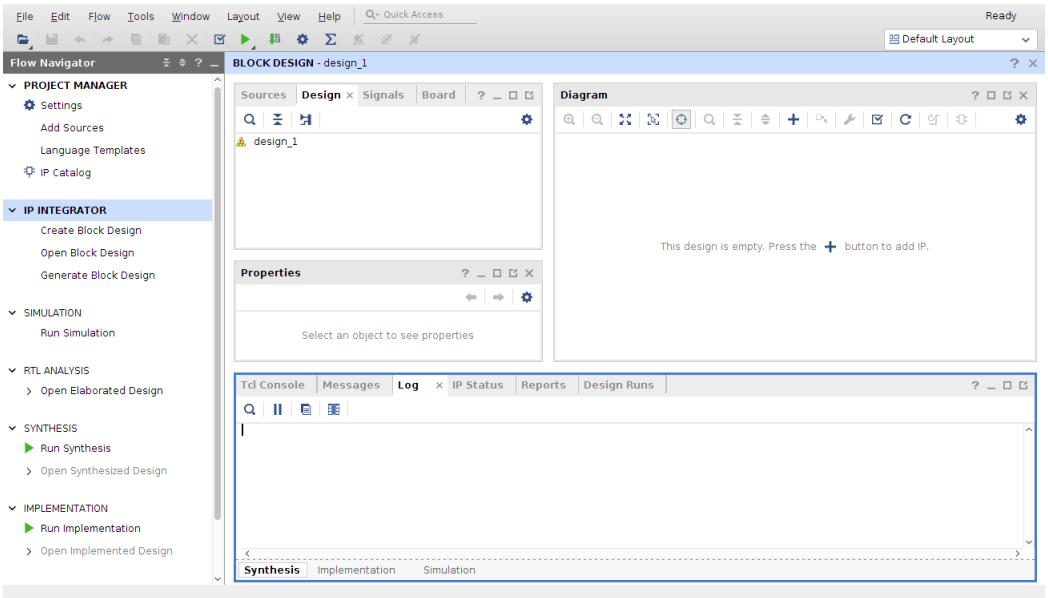


Figure 3.46: Block Design.

By default, numerous blocks and IPs are available, but sometimes the ones we have created are not visible. For this reason, we will add them to the current library. Again, through the **Flow Navigator**, under **PROJECT MANAGER**, we select **Settings** and then once the new window open, we click in the **Repository** option under the **IP** category (Fig.3.47). If our custom made IP (*laelaps_four_legs_duplicate_ip_1.0 in this case*) is not visible, then we + button and browse in our file system to locate the *ip_repo* folder or the folder where our custom IP is located (Fig.3.48). Once we click *Apply* and *OK*, our library will be updated and hence be able to find our custom IP.

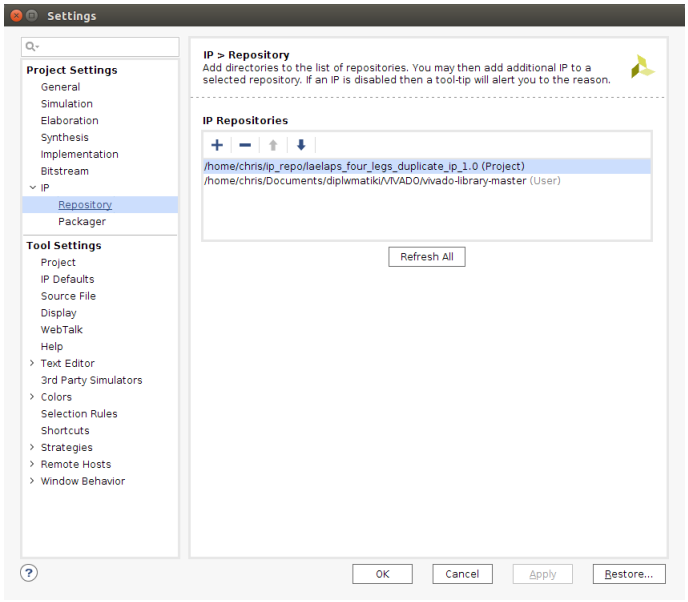


Figure 3.47: Settings.

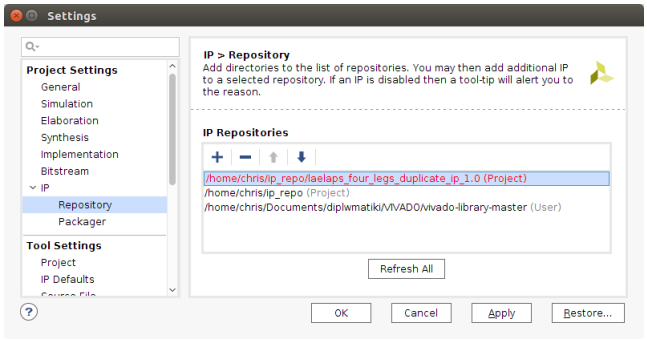


Figure 3.48: Ip_repo inclusion.

To ensure that, we return to our Block Design and click the **Add IP** button (Fig.3.49). A search engine will appear, where we can search by name the IP or block we wish to include in our Block Design. After searching and selecting our Custom IP, we should be able to see its architecture and physical ports (Fig.3.50).

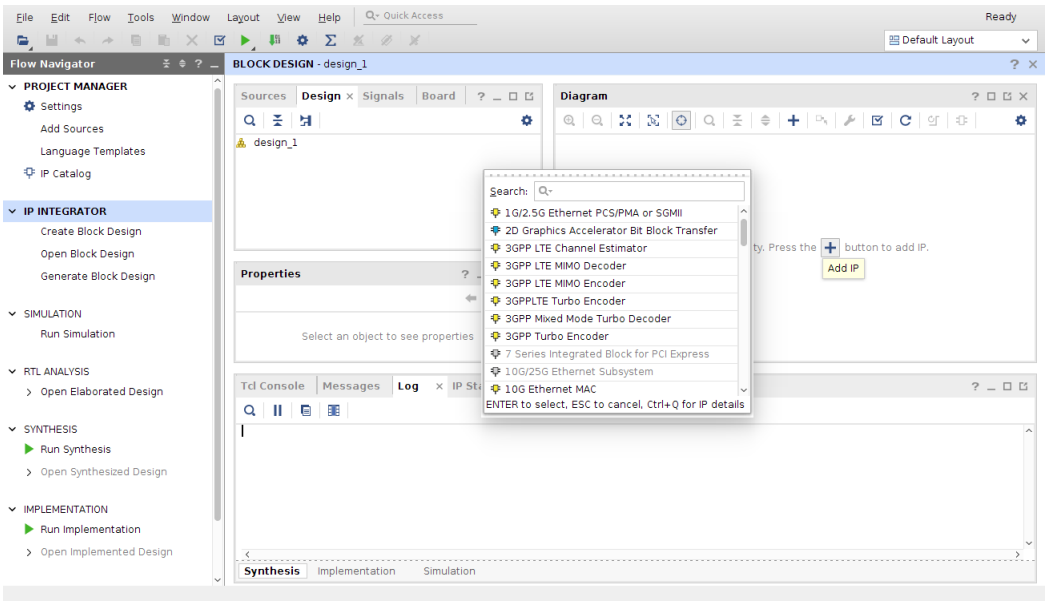


Figure 3.49: Add IP optin.

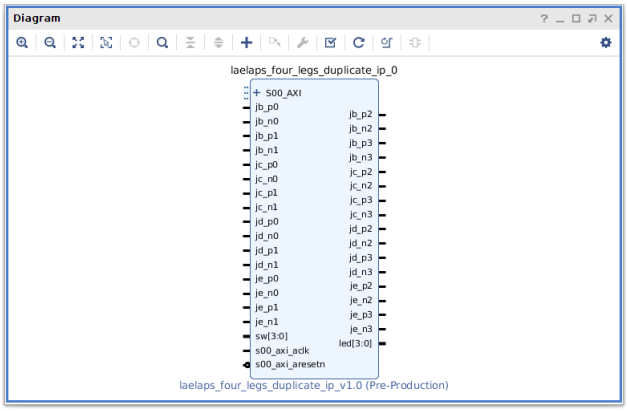


Figure 3.50: Custom IP architecture and physical ports.

Now, if during the creation of the IP, some warnings appeared relevant to the directories of the source codes, you can follow the following steps. First, right click the Custom IP in the Block Design and select **Edit in IP Packager** option. This will open a new project, where modification can take place on our Custom IP (Fig.3.51).

We return to the *Sources* pane and select and delete the *qei* and *pwm_freq* components of the IP, along with their internal components. In the next window, we select the "*Also delete project local files/directories from disk*" option and click "OK" (Fig.3.52).

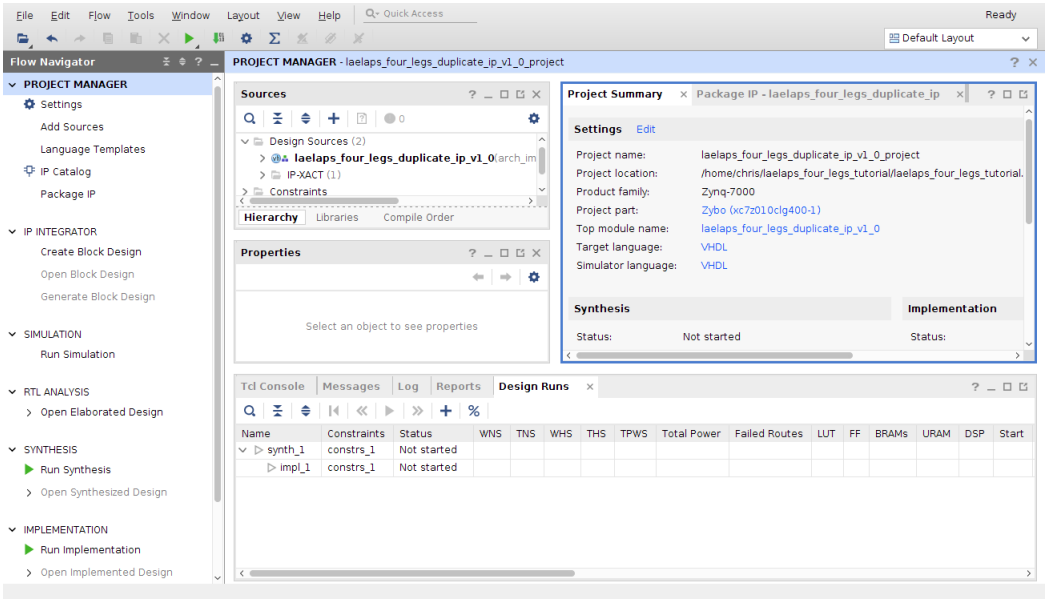


Figure 3.51: Custom IP Project.

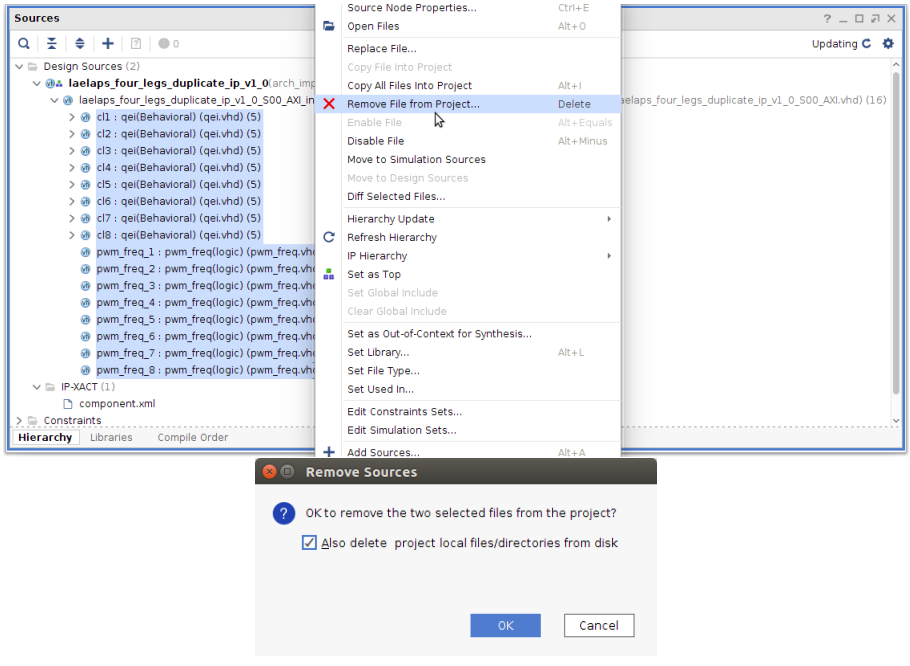


Figure 3.52: Delete the IP's components.

After the deletion of the component source files, the following state should be visible (Fig.3.53), which will be corrected by clicking the **Add Sources** "+" symbol. There, we select the "Add or create design sources" option and click Next, where we have the option to browse inside our folders and specifically inside the *src* folder, inside the ***laelaps_four_legs_duplicate_ip_1.0*** folder. In case that folder does not exist, we create it and inside it we copy all the source files we created from their original location. Once we have selected all source files, we click "OK" and proceed with the "Finish" button (Fig.3.54).

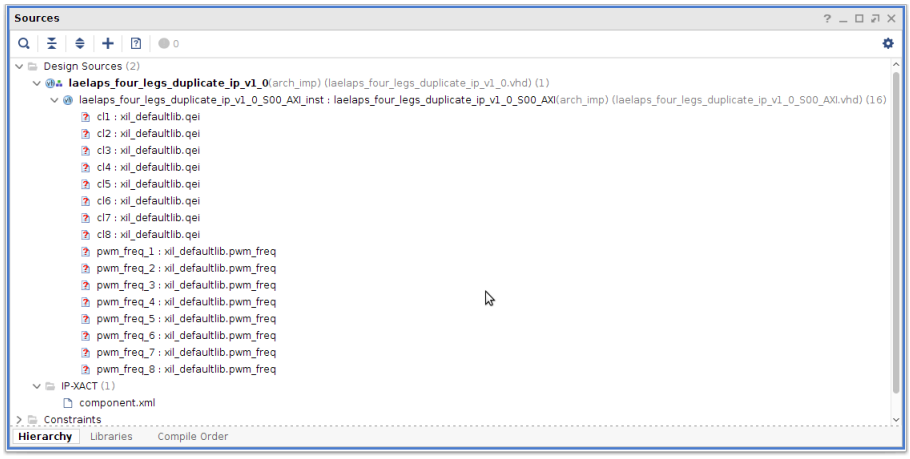


Figure 3.53: State after the deletion.

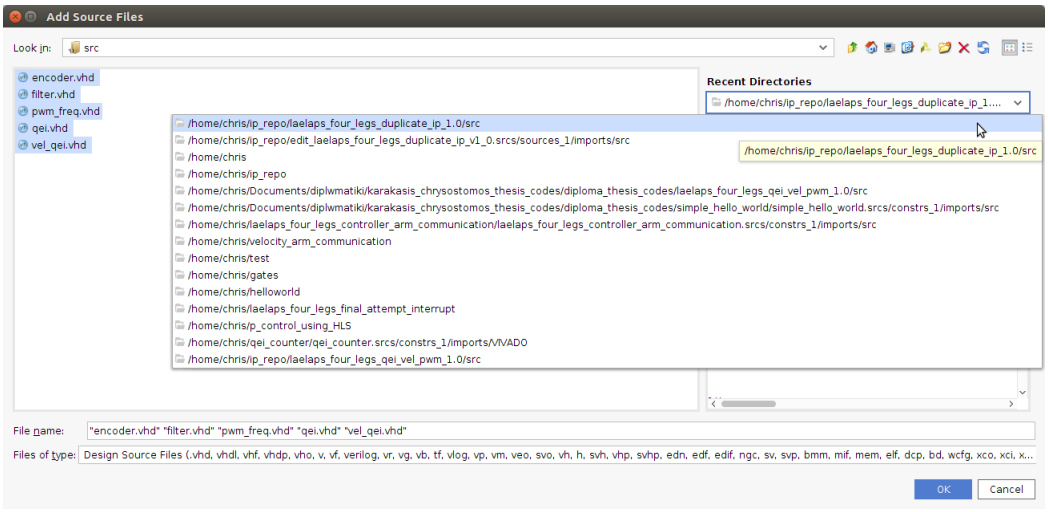


Figure 3.54: Selection of the new source files.

After that we left click the **component.xml** file inside the *Source* Pane (Fig.3.55), which will open the **Package IP Manager**. Once again, we click the **Merge changer from File Groups Wizard** option and then all warnings should disappear.

Finally, inside the *Review and Package* tab, we click the **Re-Package IP** option, which will finalize our Custom IP (Fig.3.56). It is strongly recommended to create an archive of the IP, in the *After Packaging* section, which will create a corresponding zip file of our IP that can be easily transferred to other devices and computers.

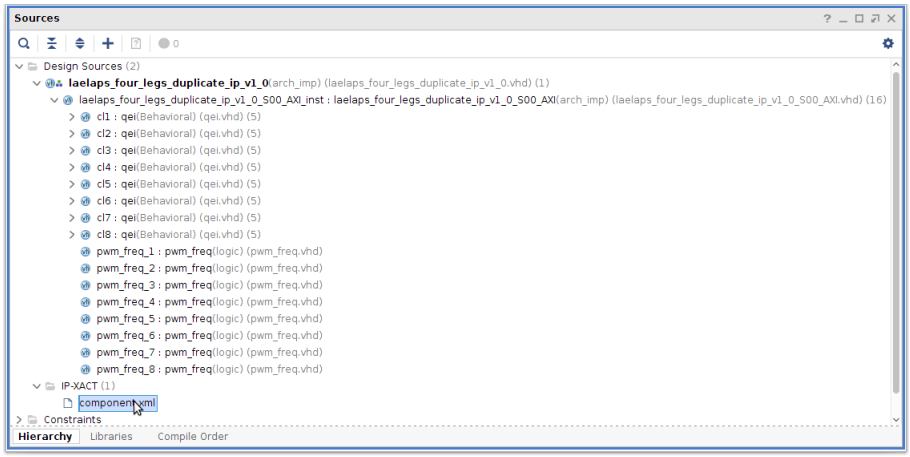


Figure 3.55: Left click component.xml

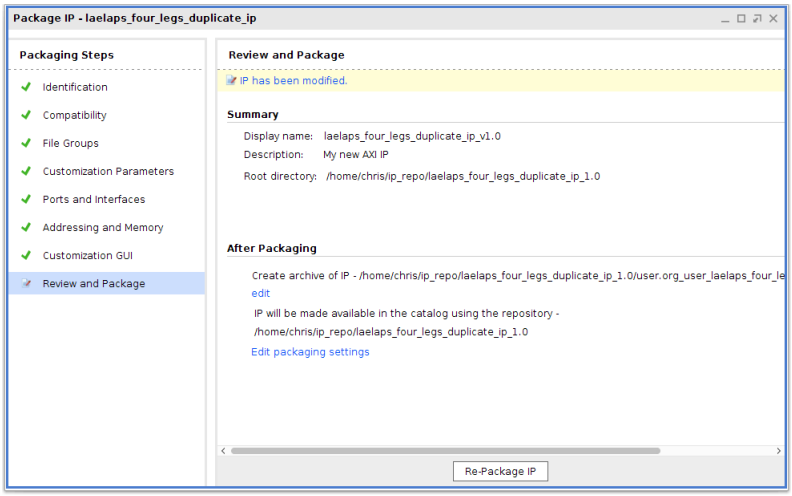


Figure 3.56: Re-Package IP.

3.3.4 Configuring the Block Design

Since our Custom IP has been completed, we are now ready to insert the remaining blocks of our implementation. Initially, we include the *ZYNQ7 Processing System*, which represents the ARM Processor of the Zybo Development Board (Fig.3.57). On the top of the Block Design Diagram, the Designer Assistance will propose two options of action, from which we select to execute the **Run Block Automation** (Fig.3.58). A new window will pop up, which explains the details of this actions. We do not change the default options and select "OK" (Fig.3.59). Following that, we select the **Run Connection Automation** option and once again proceed with the default settings (Fig.3.63). Eventually, we result with the the architecture depicted in Fig.3.61. As you can see, the *Processor System Reset* and *AXI Interconnect* blocks have been automatically created, while several connections have been made. These settings are vital to our implementation as they ensure the valid functionality of the components and their communication via the AXI Interconnect.

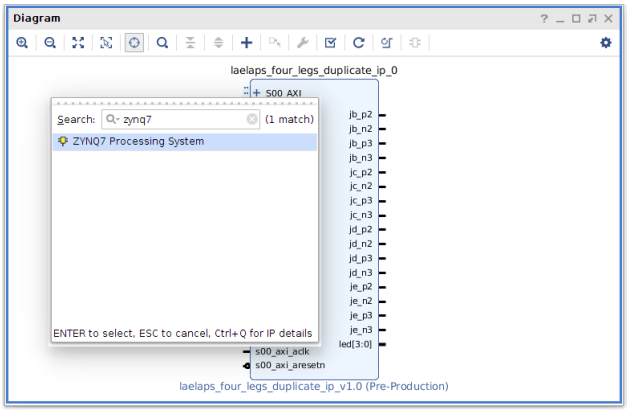


Figure 3.57: ZYNQ7 Processing System.

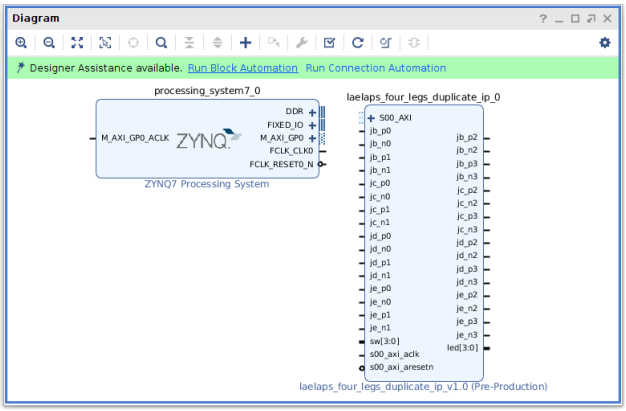


Figure 3.58: Run Block Automation.

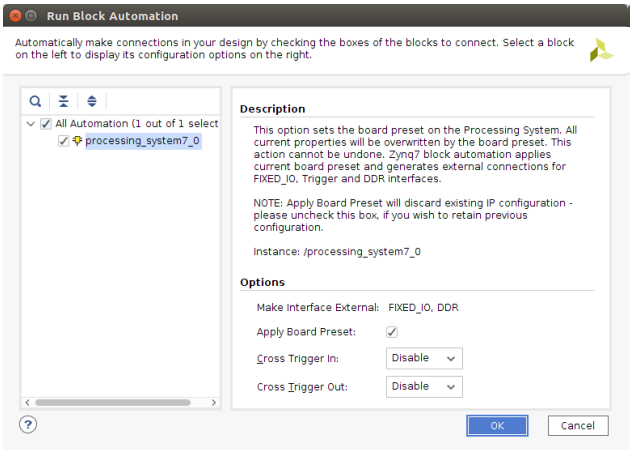


Figure 3.59: Details of Run Block Automation.

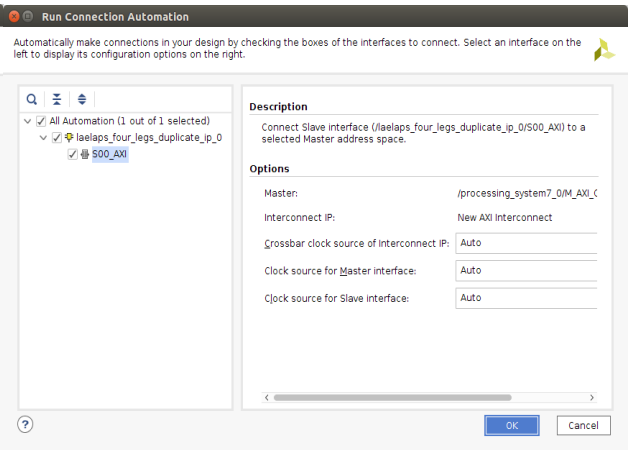


Figure 3.60: Details of Run Connection Automation.

The only thing that remains is to connect our Custom IP's inputs and outputs to their respective external ports. This is achieved by right clicking our IP and selecting the *Make External* option (Fig.3.62).

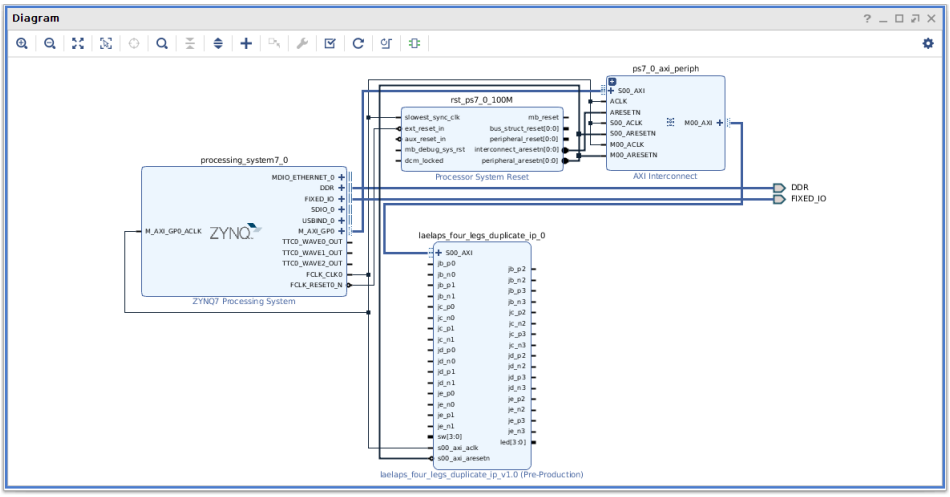


Figure 3.61: Resulting Architecture.

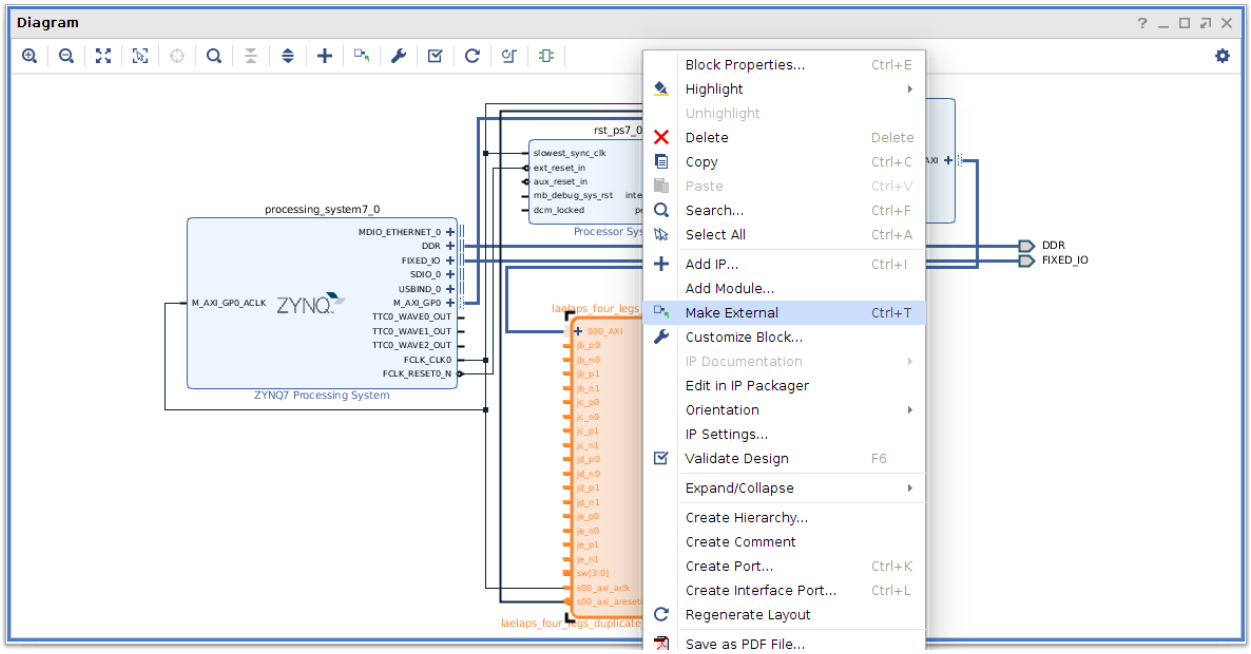


Figure 3.62: Make External Option.

Finally, we end up with the finalized architecture of our implementation, whose Layout and Routing can be optimized with the following shortcut buttons (Fig.3.64 and Fig.3.65).

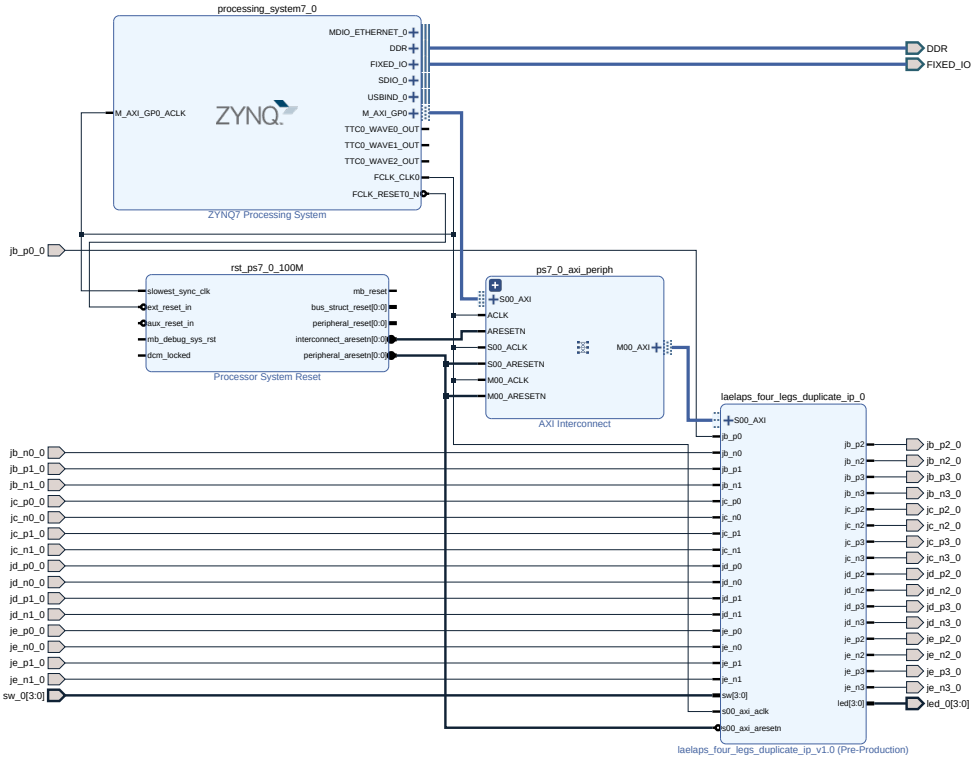


Figure 3.63: Details of Run Connection Automation.

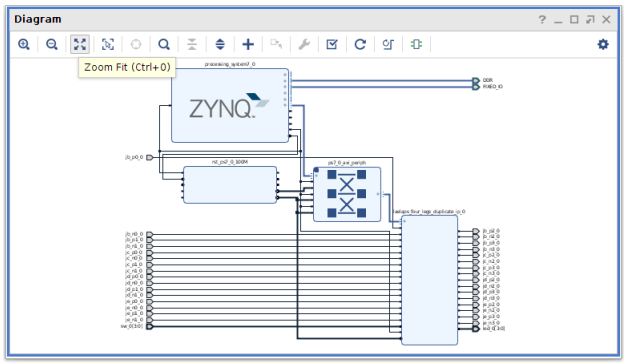


Figure 3.64: Zoom Fit Shortcut.

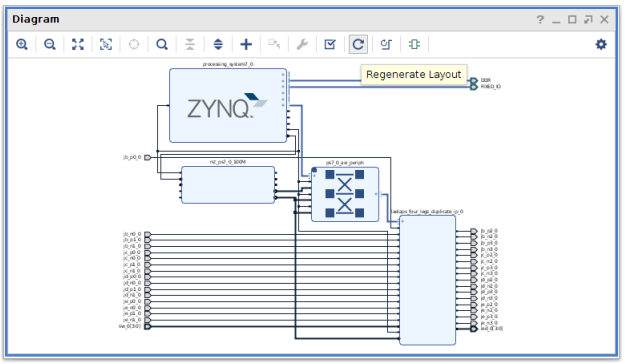


Figure 3.65: Regenerate Layout.

3.3.5 Constraints

As we had mentioned earlier, the names of the external and physical ports are configured in the constraints file *Zybo-Master.xdc*, which can be accessed through the *Sources* Pane. In that file, all physical ports of the Zybo Development Board are included and we simply have to uncomment the code lines, which correspond to the ports we wish to utilize. In this case, the switches, the LEDs and the Pmod Ports. As you can observe, for every port inside the **get_ports** field, a special codename exists, through which we can refer to that port. As a result, it is important to ensure that these codenames match with the ones inside the Block Design Diagram (Fig.3.66, 3.67).

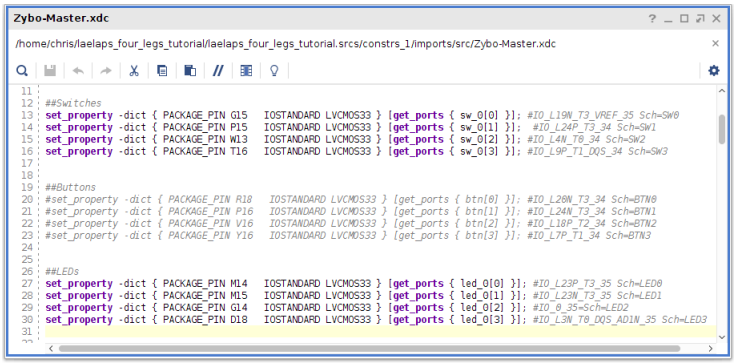


Figure 3.66: Code Lines of Constraints File relevant with the Switches and LEDs.

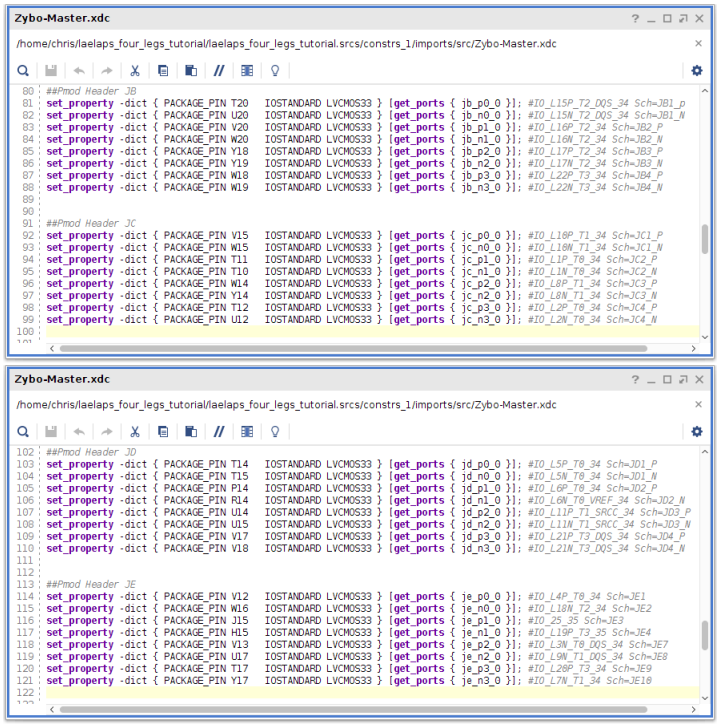


Figure 3.67: Code Lines of Constraints File relevant with the Pmod Ports.

3.3.6 Configuring the Operating Frequency of the FPGA

A critical aspect of our hardware design is its operating frequency. Specifically, the FPGA's operating frequency determines the period of the overall's system's clock signal.

One way to configure this frequency is through the **Block Design Diagram**, where we double click the *ZYNQ7 Processing System* Block. A **Re-customize IP** window will open, with a *Clock Configuration* section visible, under the the *Page Navigator* category (Fig.3.68). In fact, the system's clock is the *FCLK_ CLK0* and it is located under the *PL Fabric Clocks*. By modifying the Requested Frequency entry, we can determine the system's frequency, which can receive selected values between 0.1MHz and 250MHz. The users do not have to know these specific values, but they can simply insert their desired value and the system will find the nearest available value, which will be visible in the *Actual Frequency* entry.

We should note that choosing the maximum available frequency is not always the optimal solution. First of all, sometimes our designs cannot support all available values. In those cases, our system cannot respond to the given frequency and therefore a lower frequency should be adopted. On the other hand, sometimes our system can operate at higher frequencies than the one we have selected. However, although selecting a higher value would accelerate our processing speed, it would also increase the system's power consumption, a feature that is not always preferable.

A useful indicator for the designer is the *Worst Negative Slack (WNS)* and *Total Negative Slack (TNS)* metrics, which are created alongside with the Timing Report of the Project's Summary. This report is generated automatically during the generation of the Bitstream file, whose process will be explained in the next paragraph. If those metrics have negative values, then this indicates that our system cannot support the current frequency and hence a lower value should be selected, before re-generating the Bitstream File. Conversely, if they have positive values, the operating frequency can be increased.

For the laelaps_four_legs_tutorial, the maximum available frequency was found to be around 83 MHz (83.333336 MHz) and hence we selected this value, before generating the Bitstream File.

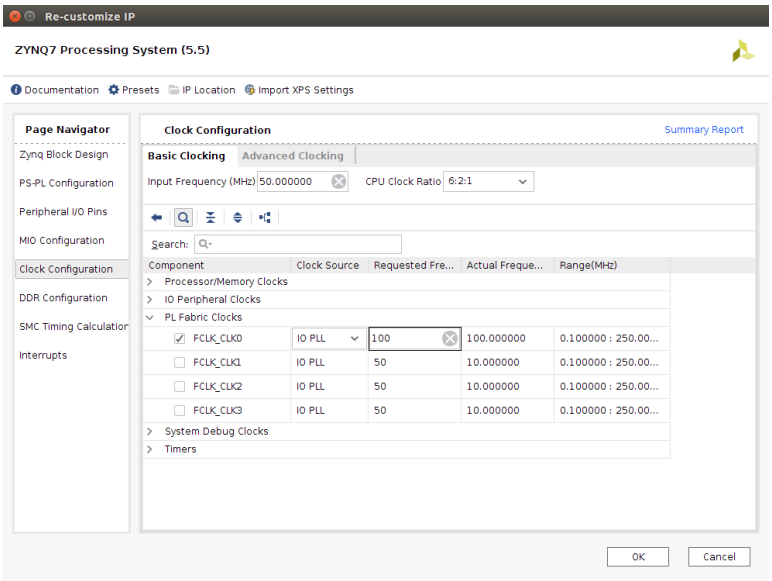


Figure 3.68: Configuration of the FPGA's Operating Frequency.

3.3.7 Generate Bitstream File

The last part of the Hardware Design of our implementation is to create a Bitstream file. To do that, we have to first create the HDL Wrapper of our design, by right clicking our design inside the *Design Sources* category of the *Sources* Pane (Fig.3.69). After we select the **Vivado manage wrapper and auto-update** option, we are ready to Generate our Bitstream, via the appropriate command in **Flow Navigator** pane, under the **PROGRAM AND DEBUG** category. Since there are no implementation results available, we hit "OK", in the pop up window to launch synthesis and implementation as well. In the **Launch Runs** window, we choose to *Launch runs on local host*, where the *Number of jobs*, corresponds to the number of processors we want to employ for this action. Due to the fact that the generation of the Bitstream usually takes around 15 to 30 minutes, we recommend to utilize as many processors as possible. You can observe the procedure of the generation, by clicking on the PROJECT MANAGER under the **Flow Navigator** Pane. In Fig.3.70 the Project Summary is observed, during the generation of the Bitstream File.

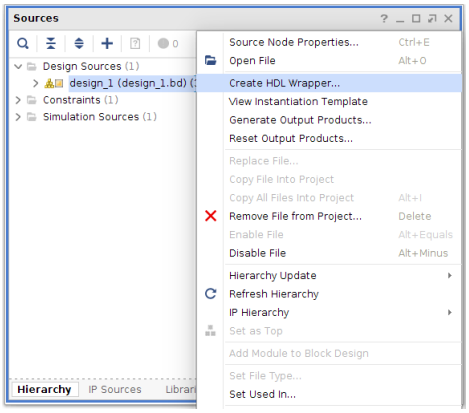


Figure 3.69: Create HDL Wrapper.

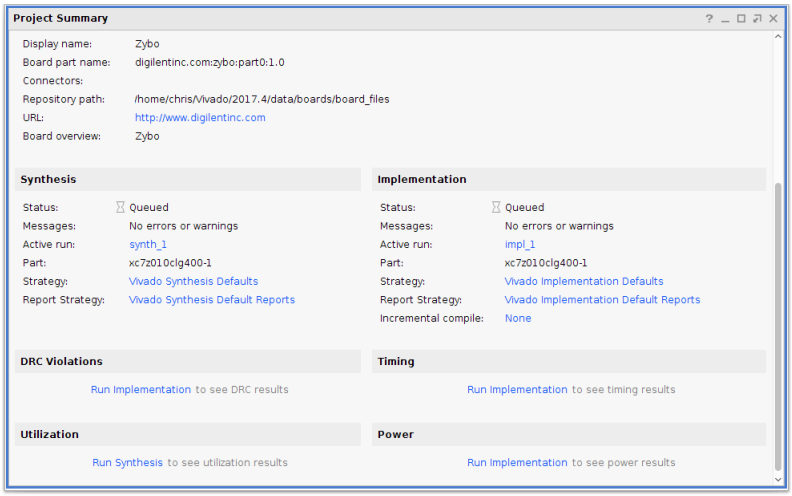


Figure 3.70: Project Summary.

Once the generation has been concluded, the Project summary informs us about various important aspects of our design (Fig.3.71). First of all, the Status of the **Synthesis** and **Implementation** sections, reassures us that no syntax errors exist in our implementation. Secondly, any critical and normal warnings notify us regarding some characteristics of our design that may prove to be troublesome or not. In Fig.3.71 the critical warning is not relevant with our design and therefore we will not delve into it. The **Timing Report** shows the values of the slack metrics, which in the 83 MHz Project Summary are nonnegative and rather small. This indicates that the given frequency is suitable, while in the 100 MHz Project Summary (Fig.3.72) the slack metrics are negative and hence indicate the necessity to decrease the frequency. The **Utilization Report** depicts the overall resource consumption of the project, which is elaborated in the Diploma Thesis [¶]. Lastly, the Power Report shows estimates regarding the On-chip Power and Temperature, although they are not accurate and therefore we do not rely on them.

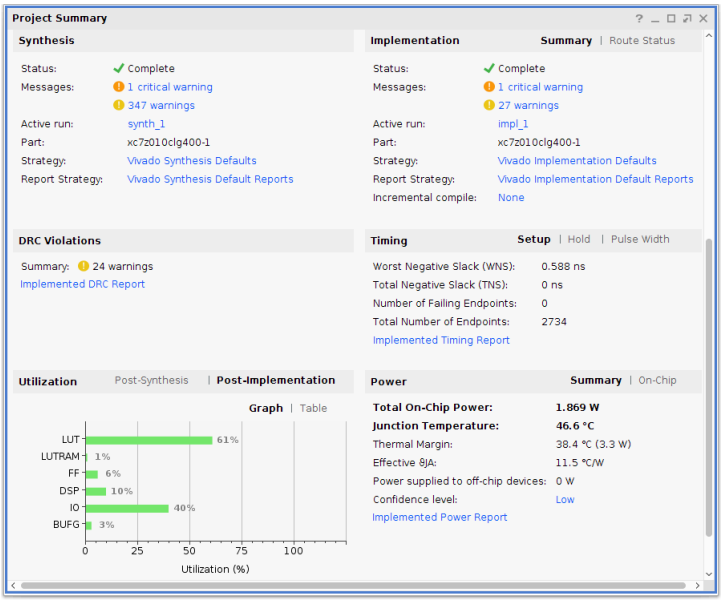


Figure 3.71: Finalized Project Summary.

Figure 3.72: Project Summary with Negative Slack for 100 MHz Operating Frequency.

For more information, the following Programmable Logic Tutorials are strongly recommended: Getting Started with Zynq,Creating IP in HDL and Creating a Custom IP core using the IP Integrator.

[¶]Page 64 in Chapter 4.

Chapter 4

Software Partition in SDK

As soon as the hardware development of our implementation on Vivado has been concluded, we can initiate the software development that will be executed to the ARM Processor of the Zybo Development Board. For this purpose an Integrated Design Environment for creating embedded applications is provided by Xilinx, the Xilinx Software Development Kit (XSDK).

4.1 Export the hardware design to SDK

After the generation of the Bitstream File, we export our design to SDK, in order to link the software and hardware partitions. In Vivado, from the File menu, select *Export* → *Export Hardware* (Fig.4.1). In the window that appears, tick *Include bitstream* and click *OK* (Fig.4.2). Again from the File menu, select *Launch SDK*. In the window that appears, use the settings of Fig.4.3 and click *OK*.

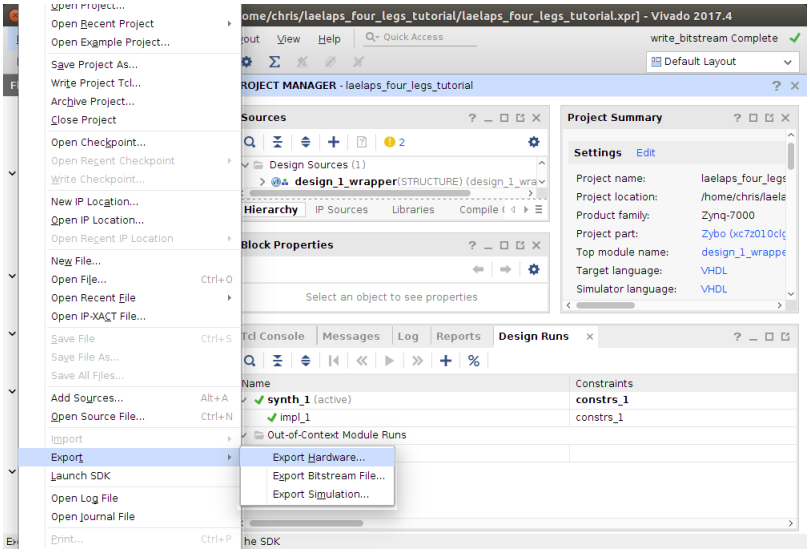


Figure 4.1: Export Hardware.

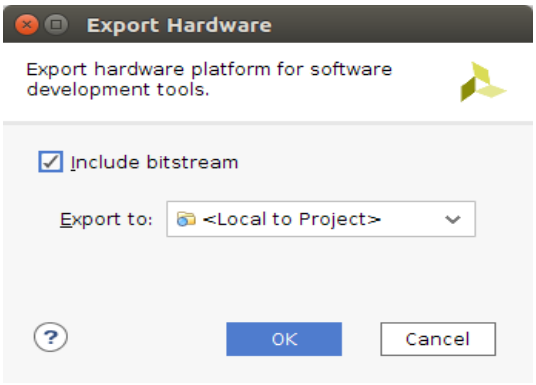


Figure 4.2: Export Hardware - Include Bitstream.

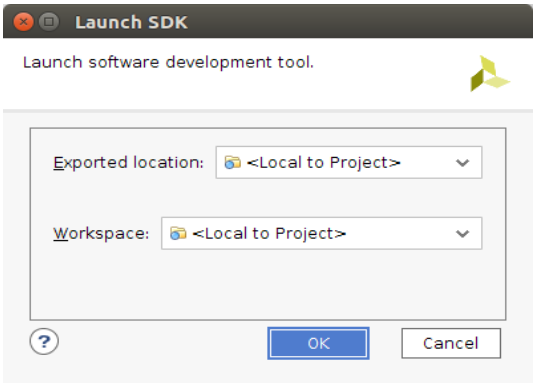


Figure 4.3: Launch SDK Settings.

At this point, the SDK loads and a hardware platform specification will be created for your design. You should be able to see the hardware specification in the Project Explorer of SDK as shown in the image below (Fig.4.4). You are now ready to create a software application to run on the PS.

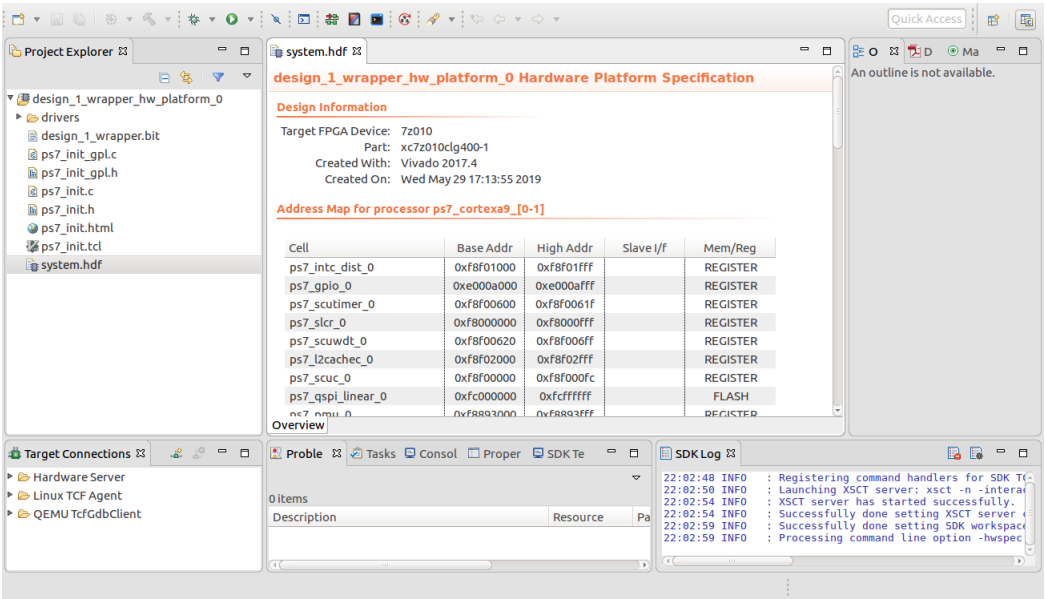


Figure 4.4: SDK Home Screen.

4.2 Preliminary Settings for the SDK

From the File menu, select *New* → *Application Project*. A New Project window will appear, through which we will create a FSBL project file, which is required in order to make our implementation bootable from a SD Card (Fig.4.5). Once a name has been chosen for the project, *laelaps_four_legs_tutorial_fsbl* in this case, we click *Next*. In the **Templates** dialog box, select the *Zynq-7000 AP SoC FSBL* template (Fig.4.6) and then click *Finish*. As a result, the application project will be created inside the *Project Explorer* pane, along with the corresponding board support package (bsp file). For more information, the following Tutorial is strongly recommended: Creating a New Zynq FSBL Application Project .

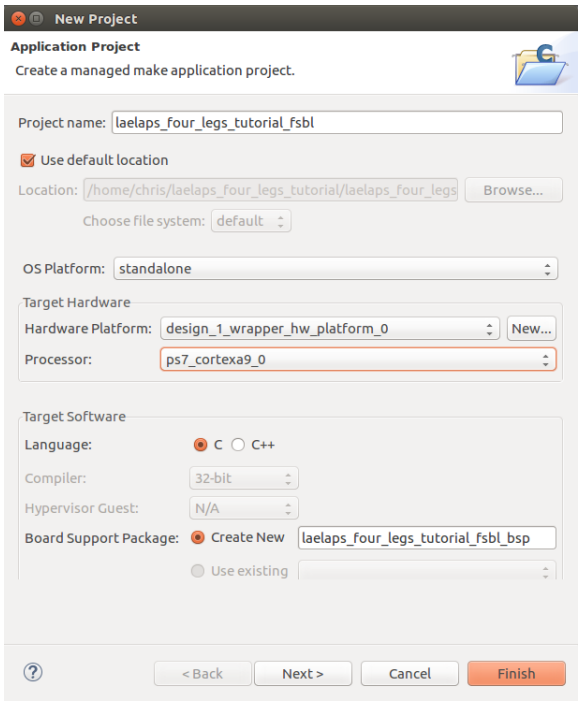


Figure 4.5: FSBL Project File.

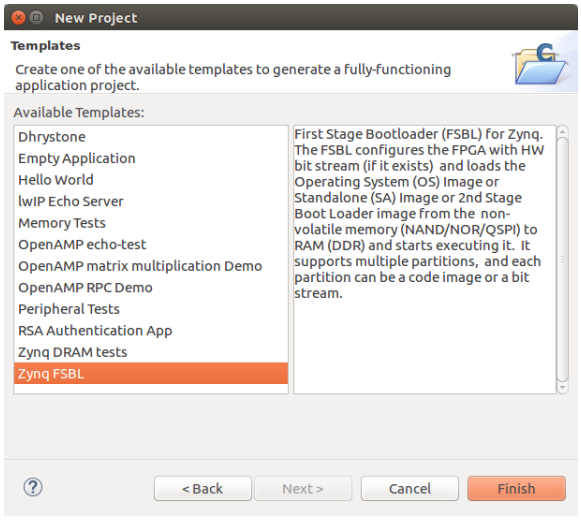


Figure 4.6: Template for Zynq FSBL.

Consequently, again from the File menu, select *New* → *Application Project*. A New Project window will appear, through which we will create our software application file that will be responsible for the programming of the ARM Processor (Fig.4.7). Once a name has been chosen for the project, *laelaps_four_legs_tutorial* in this case, we click *Next*. In the **Templates** dialog box, select the *Hello World* template (Fig.4.8) and then click *Finish*. As a result, the software application project will be created inside the *Project Explorer* pane, along with the corresponding board support package (bsp file).

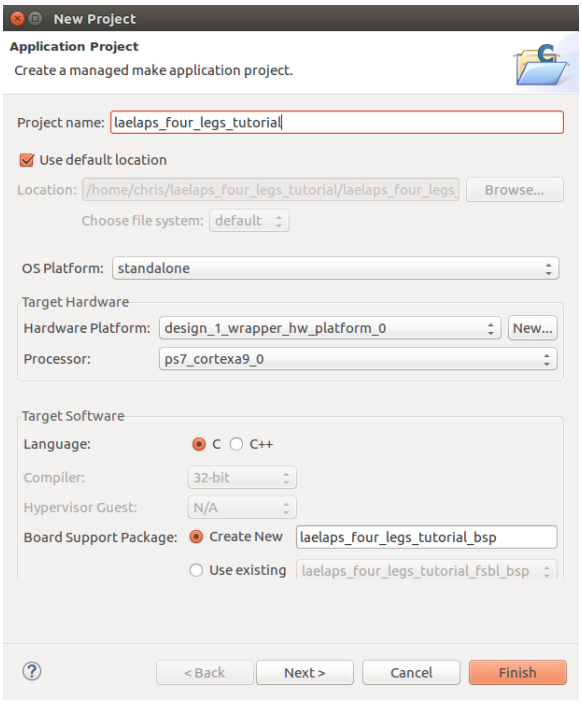


Figure 4.7: Software Application Project File.

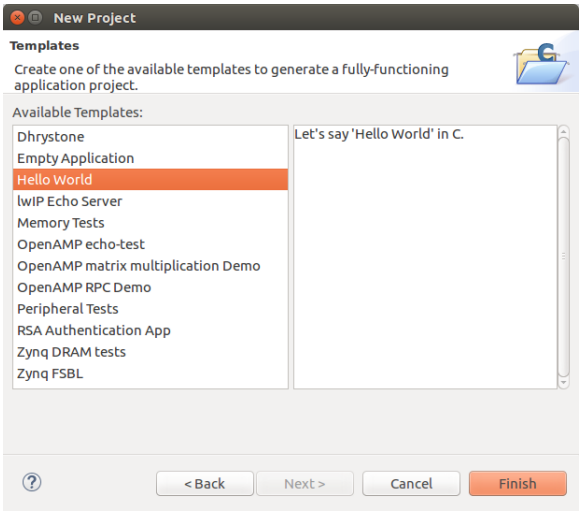


Figure 4.8: Template for Software Application.

Now, from the Project Explorer pane we navigate inside the *laelaps_four_legs_tutorial* folder and open the **helloworld.c** file, by double clicking it (Fig.4.9). This is the source file that will be modified in order to execute our implementation.

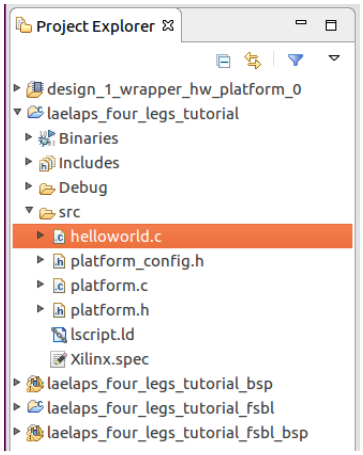


Figure 4.9: The *laelaps_four_legs_tutorial* folder.

4.3 Modify the Software Application

By default, the **helloworld.c** source file is a basic application that simply prints a "Hello World" message (Fig.4.10). Based on this example, the user can experiment and eventually formulate his/her desired application in C or C++ language.

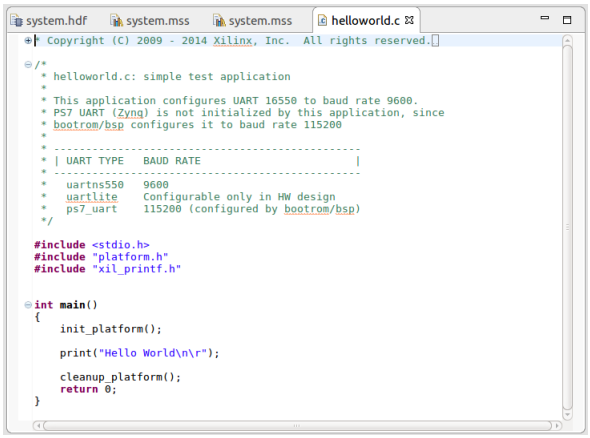


Figure 4.10: The helloworld.c initial code.

As a remainder, in our implementation the processor is responsible for the execution of both the High and Low level controllers, as well as the retention of the necessary data for post processing using Matlab. However, although the software segment of the implementation has been explained in the context of the Diploma Thesis *, some parts have been altered and hence it is required to analyze them.

First of all, an Interrupt Service Routine (ISR) has been included, which provides increased temporal precision and enables the interfacing with the user via a terminal. Specifically, we can determine how frequently an interrupt occurs, which causes the execution of the Low-level controller that has been placed inside the corresponding interrupt handler. On the other hand, the main function emulates the High-level controller, while handling the interfacing with the user. Finally, after the duration of the locomotion experiment has finished, all recorded data are transferred to the same SD card, where our implementation boots from.

For the installation of the ISR, we followed the examples described in the **Zynq Workshop for Beginners** and especially the 8th exercise, whose solution is available here. The complete pdf file, along with useful code templates and solutions to its exercises are available here, while the pdf and the 8th exercise's code are also included inside the "diploma_thesis_code.zip" file.

4.3.1 Main Function

Based on the 8th exercise solution of the **Zynq Workshop for Beginners**, we only modified certain parts to introduce the desired performance. Once all necessary declarations have been made, an infinite while loop initiates, where the interrupts are enabled and the frequency of the interrupts is configured (Fig.4.11). Currently the frequency has been set to 1kHz, since it has been proven suitable for our application in the experiments so far.

*Chapter 5 of the Diploma Thesis.

```
// Load the timer with a value that represents one second of real time
// HINT: The SCU Timer is clocked at half the frequency of the CPU.
XScuTimer_LoadTimer(&my_Timer, XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ*pow(10,-3) / 2);
//XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ / 2 -> 1Hz
//XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ*pow(10,-3) / 2 -> 1kHz
```

Figure 4.11: Interrupt Frequency Configuration.

The *XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ* constant represents the frequency of the CPU that is equal to 650MHz. Since the SCU Timer is clocked at half that frequency, if we load the *XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ / 2* value to the timer, every second (1Hz frequency) an interrupt will occur. According to that logic, we configured the frequency to 1kHz.

Next, the interface with the user begins, by printing a welcome message and a request for the state machine code. Inside a while loop, which checks whether a flag variable is equal to '0', the system continuously awaits for a user input from the terminal via a non-blocking function. When the code "1" is given, an interrupt counter is reset, according to which we can measure the number of interrupts that have occurred and hence the time that has intervened since the pressing of the "1" code. Simultaneously, a counter variable is also reset to zero, which designates where the next experiment measurement should be temporarily stored inside the global *array_print* array. Lastly, the flag variable is set to '1' and another message is printed to the terminal, informing the user that the trotting experiment has started.

Following, the system awaits again inside a while loop, which checks whether the flag variable is equal to '1', for a user input from the terminal via a non-blocking function. In case the user types "0", the experiment is terminated, the flag variable is set to '2', and a corresponding message is printed. Otherwise, the while loop continues.

Afterwards, a series of if statements take place that help to deduce the current state of the flag variable. If the flag variable is equal to '2', it indicates that the user commanded the termination of the program and hence the control gains are set to zero. Furthermore, a delay of 0.5 sec is inserted to ensure that at least one more interrupt will occur, which will set all duty cycles to zero, in order to disable the PWM signals sent to the motors. Moreover, the system breaks from the infinite while loop and once all interrupts have been disabled, the main function exits. If the flag variable is equal to '3', it indicates that the experiment was terminated due to the expiration of the experiment's duration and hence the storing of the recorded data begins. A corresponding message is printed, the interrupts are disabled and the *store_data_in_sd* function is called. This function reads and prints to the SD Card, all data stored inside the global *array_print* array, where all the experiment measurements are temporarily saved. Once the printing has concluded (usually takes around 20 seconds), the system exits.

4.3.2 array_print function

This function was created based on the examples found here and here. In order to utilize the f_printf function to print our data to the SD Card file, certain adjustments are required for the SDK program. Inside Xilinx SDK, open your Board Support Package (system.mss file) an select Modify this BSP's Settings (Fig.4.12). Enable the xilffs library and then under Overview, you can select xilffs (Fig.4.13). As instructed here, in order for the f_printf function to be available *read_only* must be equal to 0, while *use_strfunc* must be equal to either 1 or 2.(Fig.4.14)

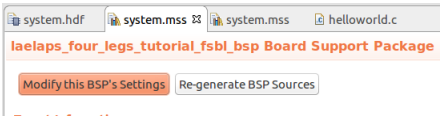


Figure 4.12: Modify this BSP's Settings

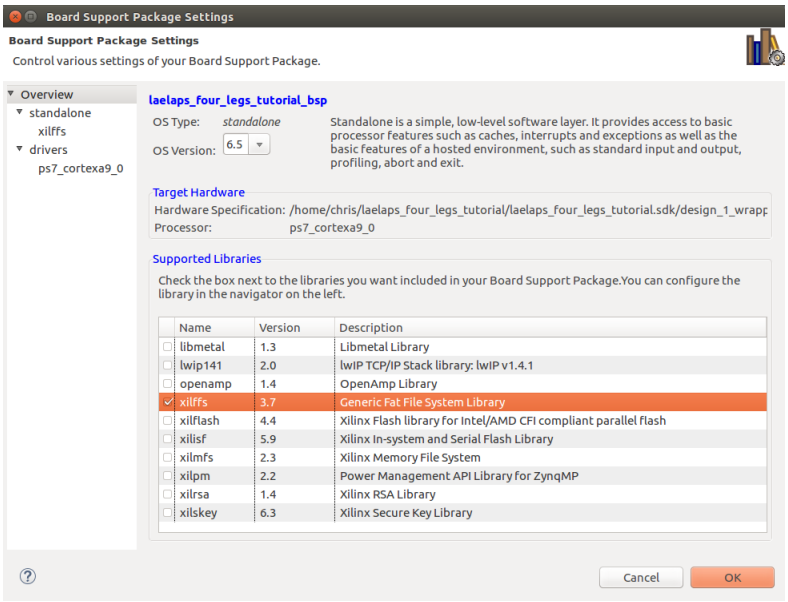


Figure 4.13: Enable xilffs library.

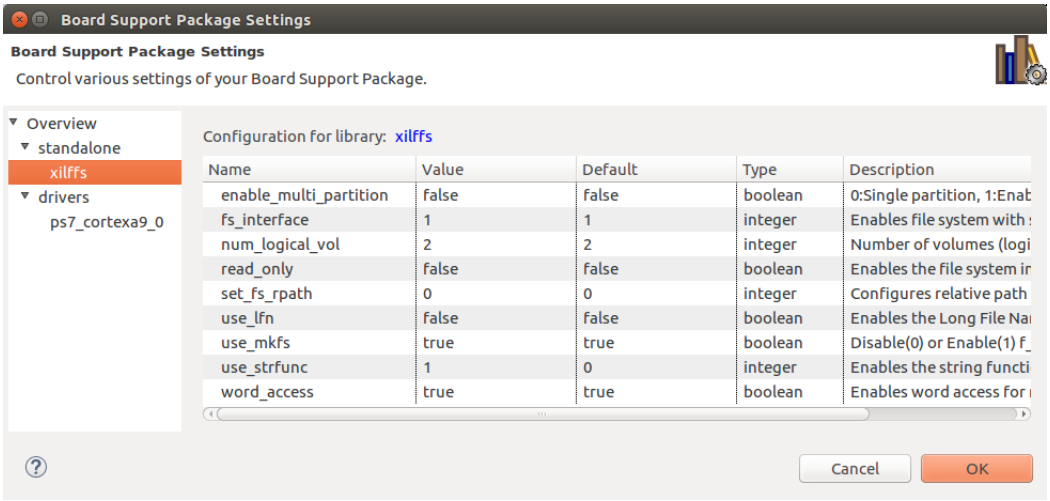


Figure 4.14: Configuration for library xilffs.

4.3.3 Interrupt Handler

Every time an interrupt happens, the program control transits to the *my_ timer_ interrupt_ handler* function, which represents its interrupt handler. After the declaration and definition of all necessary variables for the Low-level controller, certain default commands pertinent with the functionality of the interrupt are included, while the global interrupt counter is increased. Based on that counter, we can measure time, given that every 1ms an interrupt takes place. For example, if the interrupt counter is equal to 5000, then 5 seconds have intervened since the reset of the interrupt counter, which happens after the user has pressed "1" in the main function. Therefore, we can keep track of time since the beginning of the trotting experiment and later deduce whether it has exceeded the desired duration.

Subsequently, several if statements help us find out at which state our system is. If the global flag variable *flag* is equal to '0' and we have not exceeded the first *lerp_ t_ interval* seconds of the experiment, then the user has not yet pushed the "1" button and hence we simply execute control to zero point. The ellipse parameters are set to zero, while the control gains steadily reach their final values through a linear interpolation of *lerp_ t_ interval* seconds.

If the *flag* is equal to '1' and the first *lerp_ t_ interval* seconds of the experiment have not passed, the ellipse parameters start to increase progressively to their final values through linear interpolation of *lerp_ t_ interval* seconds.

If the *flag* is equal to '1' and less than *lerp_ t_ interval* seconds remain for the experiment to finish, the ellipse parameters start to decrease progressively to their initial values through linear interpolation of *lerp_ t_ interval* seconds.

If the *flag* is equal to '2', than the user has ordered the termination of the experiment and the control gains are set to zero, in order to disable the PWM outputs to the motors.

Ultimately, if the duration of the experiment has exceeded its defined limit and the *flag* variable is '1', then the *flag* is set to '3' and the Control gains are set to zero, in order to stop the trotting experiment and initiate the storage of the recorded data.

Next, the Low-level controller is executed, utilizing the parameters that were defined inside the if statements previously. As described in the 5th chapter of the Diploma Thesis, first the inverse kinematics and the trajectory planning are executed, and afterwards the PD-Controller is applied. For the communication with the FPGA, the following pointer is utilized, which is linked with the memory address of the first slave register of the AXI4-Lite Interface (Fig.4.15). The memory addresses dedicated to the AXI4-Lite Interface are specified in the system.hdl file of our project (Fig.4.16).

```
//Pointer to the Memory Address of the first slave register
Xuint32 *baseaddr_p = (Xuint32 *)XPAR_LAELAPS_FOUR_LEGS_DUPLICATE_IP_0_S00_AXI_BASEADDR;
//*(baseaddr_p+1) refers to the second slave register and so on
```

Figure 4.15: Pointer to the first slave register's memory address.

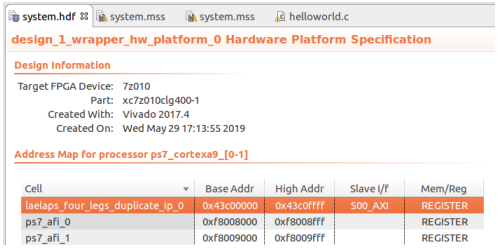


Figure 4.16: Memory Address specified in the system.hdf file.

4.4 Libraries Installation

Our final source file utilizes a lot of mathematical functions that are included normally in the "math.h" C library. In order to have access to that library, we have to carry out the following steps.

From the Project menu, select Properties (Fig.4.17). In the next window, under *C/C++ Build* click on *Settings* and then under *ARM v7 gcc linker* click on *Libraries*. Afterwards, click on the Add button (Fig.4.19), which allows us to include the "m" library we want (Fig.4.18). After this, there should no problem using any mathematical functions.

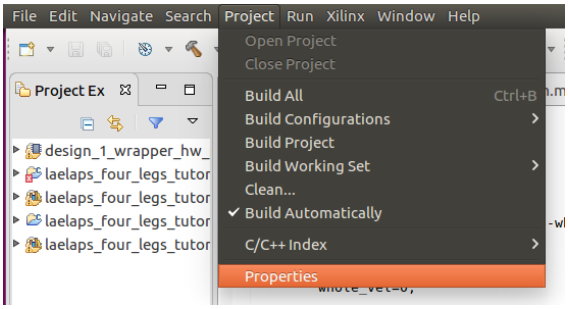


Figure 4.17: Project → Properties.

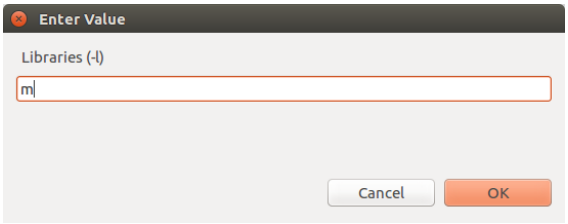


Figure 4.18: Add m Library.

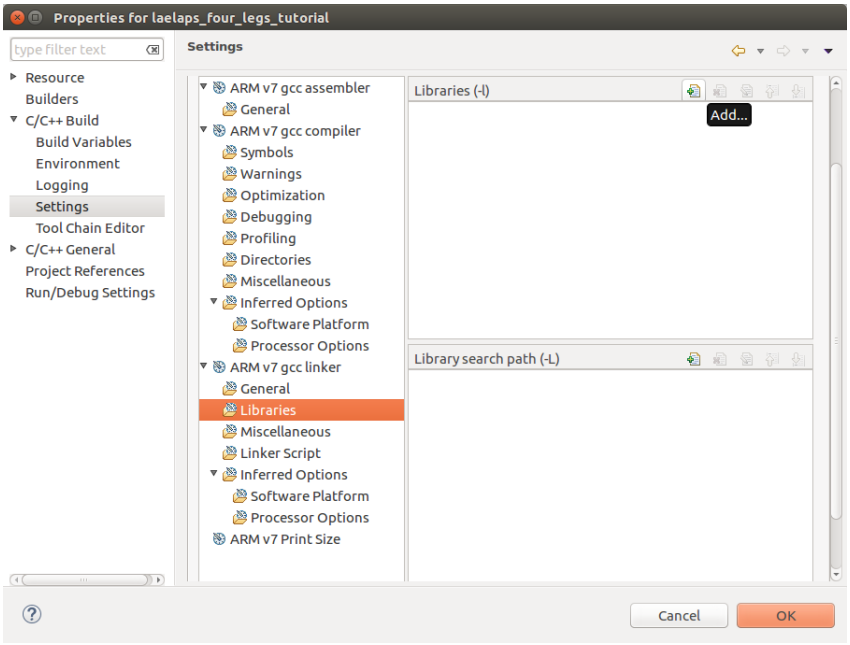


Figure 4.19: Add Library.

4.5 Load executables to the SD Card

When the editing of the software partition has also been completed, we can transfer our source files to the SD Card that will later be mounted into the Zybo Development Board. Specifically, a BOOT.bin will be created, which will entail both the software and hardware segments of our implementation. The programming of the device can also be realized through a USB cable, however this method was proved to be defective.

First, we simply select the *laelaps_four_legs_tutorial_fsbl* file inside the **Project Explorer** pane, and then from the *Xilinx* menu at the top of the screen, we click the *Create Boot Image* option (Fig.4.20). A new window will appear with several default settings that do not require modification. The only thing that is necessary is to add the file path of our software application in the *Boot image partitions* section. Of course, the *Output path* can be modified to any convenient location for the user, as this is where the BOOT.bin file will be stored afterwards. Thankfully, as you can see, the Bitsteam and fsbl files are already included and hence we simply have to click on the *Add* button and browse to the .elf file of our software application project, *laelaps_four_legs_tutorial.elf* in this case (Fig.4.23). Ultimately, the following status should be visible (Fig.4.24), where all that remains is to click the *Create Image* button. If we now browse to the location specified in the *Output path*, we should find the BOOT.bin file, which afterwards has to be transferred to a FAT-32 formatted SD Card. On the same SD Card, a .txt file will be created after the execution of the implementation. The name of this file can be configured in the initial declaration of the C code (Fig.4.21). Naturally, if any modifications take place either on the hardware or software partition, the whole Create Boot Image procedure has to be repeated, and the new BOOT.bin file has to be retransferred to the SD Card.

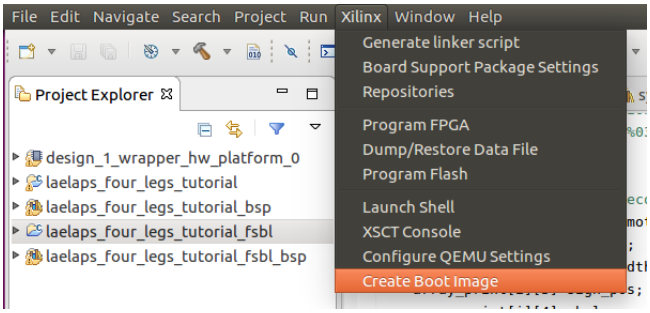


Figure 4.20: Create Boot Image.

```
//the name of the .txt file, where the recorded data will be stored
static char FileName[32] = "0:/reoutput.txt";
```

Figure 4.21: Configuration of the .txt file's name.

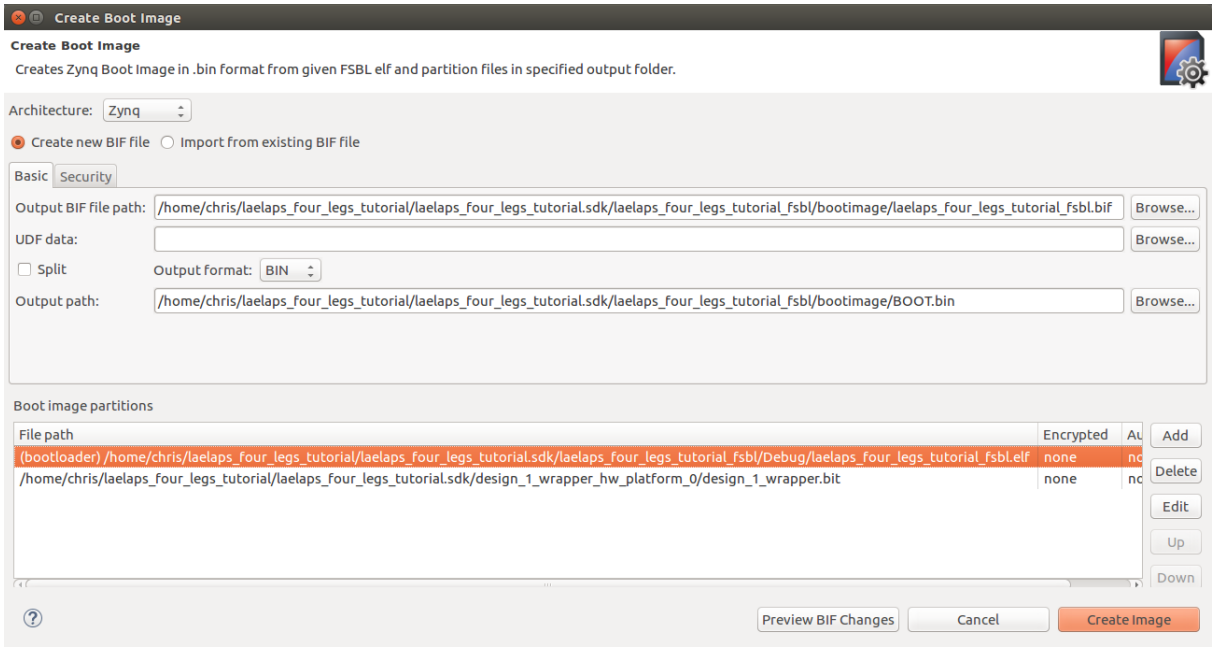


Figure 4.22: Create Boot Image Settings.

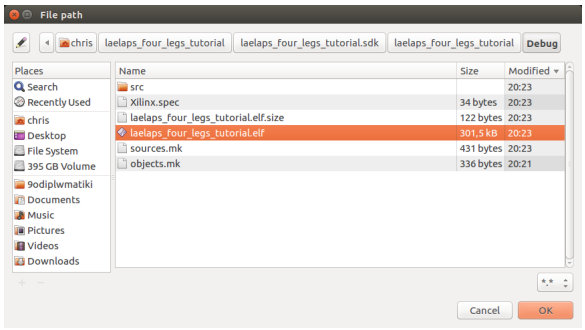


Figure 4.23: Location of the Software Application Project .elf file.

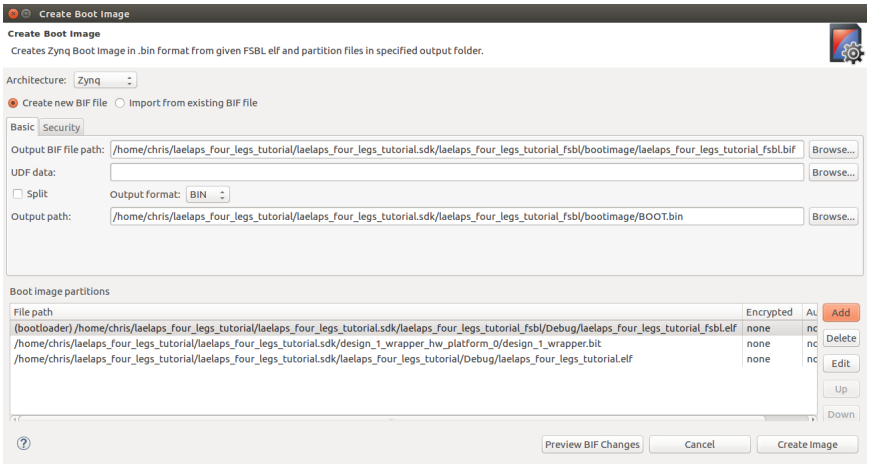


Figure 4.24: Final status of the Create Boot Image window.

The SD Card containing the BOOT.bin file should be inserted in the *microSD connector (Reverse side)* (Port No.25 of Fig.4.25) and the *Programming Mode Jumper* (No.21 of Fig.4.25) should be set to the two pin headers on the left. Moreover, the device should be connected to a PC via a USB cable (Port No.3 of Fig.4.25). As soon as the Board is powered on, the execution of the source codes begins. If we want to reset the Processor, we can simply press the Processor Reset Pushbutton (Port No.15 of Fig.4.25), and for a complete reset, we can press the *Logic configuration reset Pushbutton* (Port No.16 of Fig.4.25).

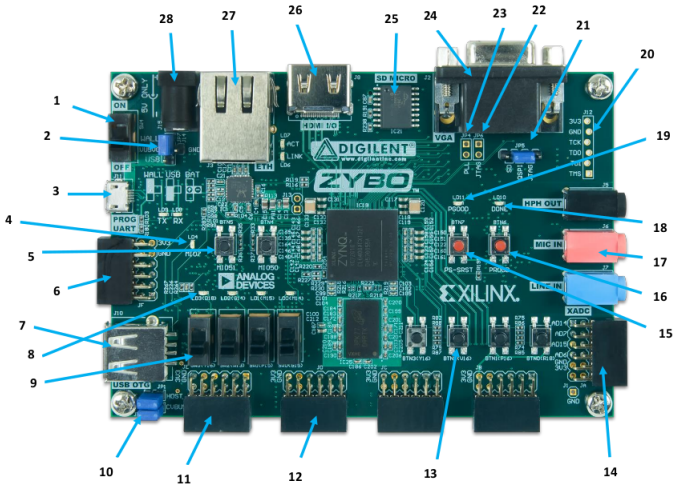
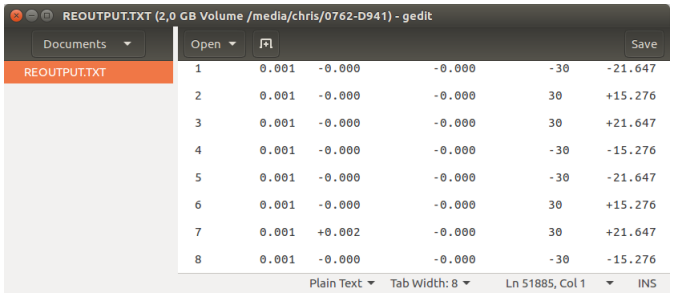


Figure 4.25: ZYBO Device Diagram.

Additionally, the encoders' of the motors and the PWM Ethernet cables should be connected to the Secondary Boards. Each socket has been numbered with its respective code that is created in the software partition and hence the REOUTPUT.TXT file. As a result, the user can connect each motor's signals wherever he/she wants to, but has to invert the angles that correspond to right legs' signals (Refer to the last lines of the *my_timer_interrupt_handler* function in the Software partition) and update the Matlab code for the visualization (Refer to the next chapter). As you can see, the .txt file entails useful information regarding each joint of the robot, recorded at several moments of time. In fact, each line consist of the respective joint's codename (1-8), the specific time instant of the recording (0.001 refers to 1ms), the angle (degrees) and velocity of the joint (rad/s), the Duty Cycle of the PWM signal that will be sent to the joint's motor (+/- for Positive/Negative Direction), and the desired angle of the joint (degrees).


REOUTPUT.TXT (2,0 GB Volume /media/chris/0762-D941) - gedit

Documents | Open | Save

REOUTPUT.TXT

1	0.001	-0.000	-0.000	-30	-21.647
2	0.001	-0.000	-0.000	30	+15.276
3	0.001	-0.000	-0.000	30	+21.647
4	0.001	-0.000	-0.000	-30	-15.276
5	0.001	-0.000	-0.000	-30	-21.647
6	0.001	-0.000	-0.000	30	+15.276
7	0.001	+0.002	-0.000	30	+21.647
8	0.001	-0.000	-0.000	-30	-15.276

Plain Text | Tab Width: 8 | Ln 51885, Col 1 | INS

Figure 4.26: Form of the REOUTPUT.TXT file.

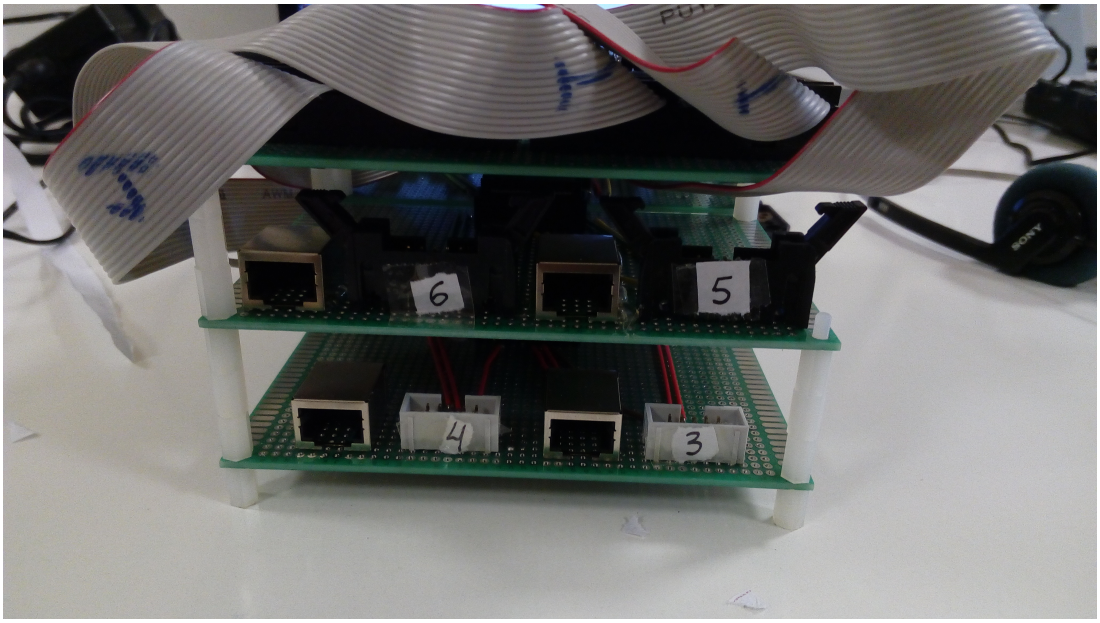


Figure 4.27: Sockets 3-4-5-6 of the Secondary Boards.

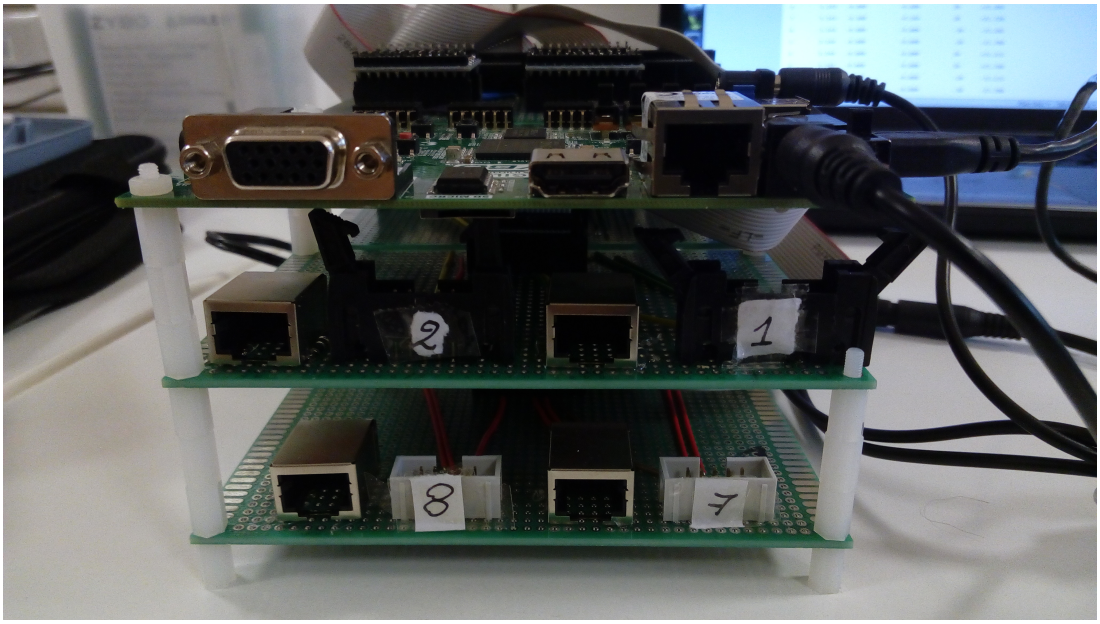


Figure 4.28: Sockets 1-2-7-8 of the Secondary Boards.

Chapter 5

Execution of the Implementation

As mentioned above, the device is connected with a PC through a USB Cable, which is required for the interfacing with the user. Specifically, the user can insert commands and observe any messages printed from the processor. This feature is achieved through serial communication via the serial port. In order to monitor that communication the Minicom program is utilized (Section 2.1). In Fig.5.1 we can observe the Start Menu of Minicom through which we can configure the settings of the serial communication.

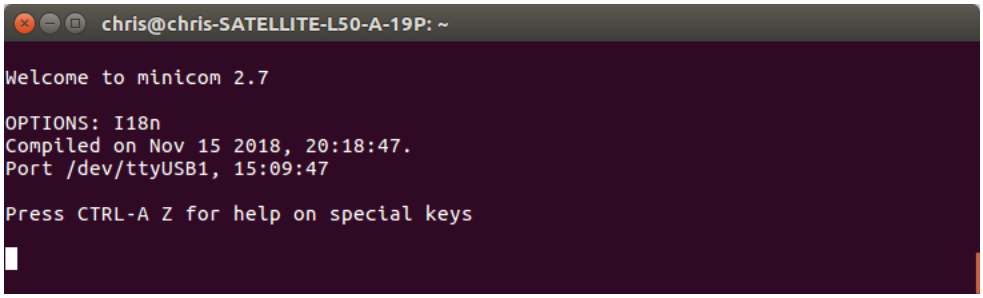


Figure 5.1: Start Menu of Minicom.

As soon as the device has been powered on, we can observe messages printed out in the terminal, while the users can insert their commands when it is required, as programmed in the software application. In the following figures the output of the system is illustrated for two different executions: in both of them, the system is commanded by the user to initiate the trotting experiment and in the first one (Fig.5.2) it is terminated after the expiration of its duration, while in the second one (Fig.5.3), the user commands its termination.

Once the execution of the implementation has been concluded, we can remove the SD Card and insert in our PC. Inside it, apart from the BOOT.bin file, we should find a .txt file (REOUTPUT.TXT in our case Fig.4.21), which will entail all recorded measurements from the experiment (Fig.5.4). In order to visualize these data, a post-processing phase takes place in Matlab.

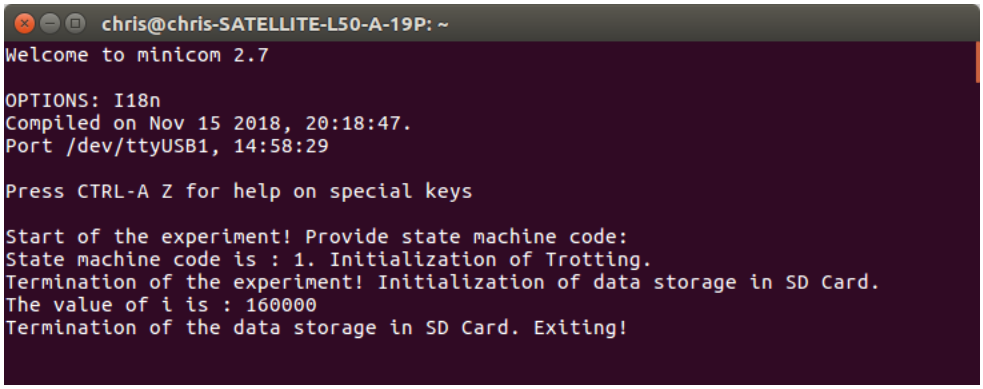


Figure 5.2: Printed Messages After Execution - Termination due to expiration of experiment's duration.

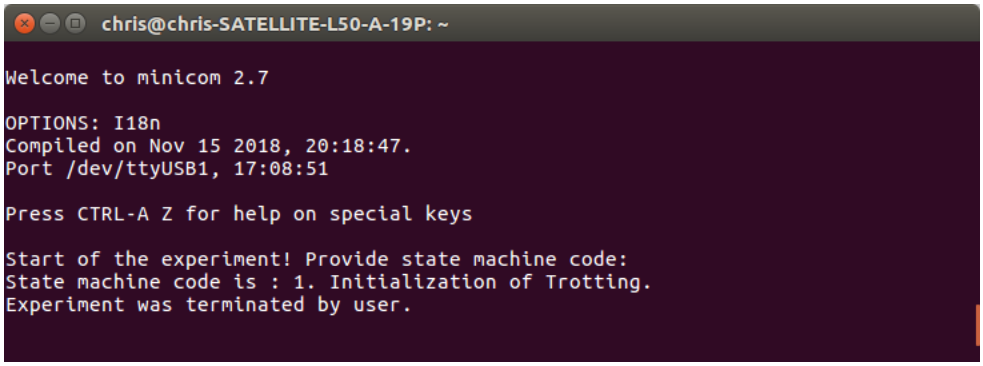


Figure 5.3: Printed Messages After Execution - Termination by user.

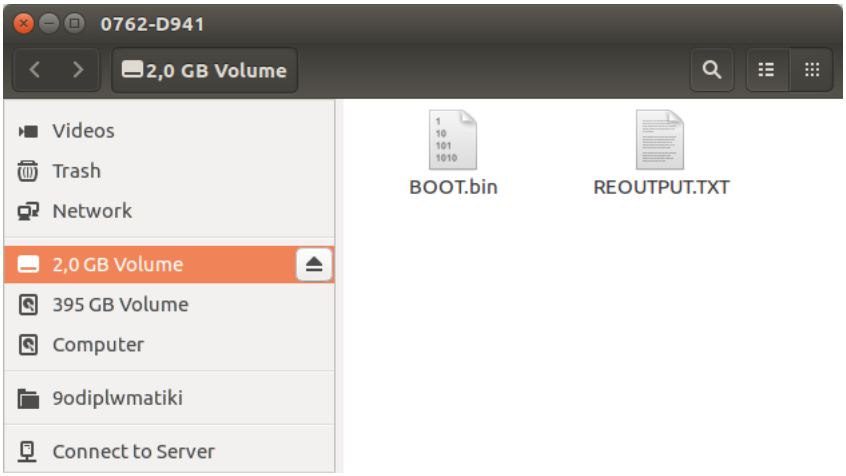


Figure 5.4: SD Card folder after execution.

Chapter 6

Matlab Codes

The .txt file that is created after the execution (REOUTPUT.TXT), is given as input to the *velocity_ estimation_ four_ legs_ inv_ kin.m* Matlab code, which creates four figures: the response of each leg's end effector in the steady state, the response of each joint's angle, the response of each joint's velocity, and the response of each joint's PWM Command. Through them, we can validate our experiment and observe its outcome.

According to where each motor signals were connected the Matlab code has to be updated, in order to ensure that the titles of the figures refer to the right joints. For example, in Fig.6.1 the lines with codenames 1 and 2 of the REOUTPUT.txt file correspond to the Hinder Right leg, the signals with codenames 3 and 4 of the .txt file correspond to the Hinder Left leg and so on. That way we can ensure that the produced figures represent the correct joints.

```
 8 -    A7=A(find(A(:,1)==1),2:6); %these are for the HR leg
 9 -    A8=A(find(A(:,1)==2),2:6);
10 -    A5=A(find(A(:,1)==3),2:6); %these are for the HL leg
11 -    A6=A(find(A(:,1)==4),2:6);
12 -    A3=A(find(A(:,1)==5),2:6); %these are for the FR leg
13 -    A4=A(find(A(:,1)==6),2:6);
14 -    A1=A(find(A(:,1)==7),2:6); %these are for the FL leg
15 -    A2=A(find(A(:,1)==8),2:6);
```

Figure 6.1: Match-up of figures' titles and connected motor signals.

The *velocity_ estimation_ four_ legs_ inv_ kin.m* is available inside the *diploma_ thesis_ codes* folder as well.