Preface

Thanks so much for Mr. Joseph Attard for writing this amazing tutorial book for the Z-turn Board. Joseph is a Senior lecturer II working at Malta College for Arts Science and Technology on the island of Malta. He became an electronics enthusiast since a very early stage which culminated in a BEng Hons degree in Electronic Systems from Portsmouth University UK (2010) and a Master degree in MicroElectronic and MicroSystems from the University of Malta (2016). He has gained great experience from building various embedded systems using both PIC microcontrollers and Xilinx FPGAs. He makes sure to keep up with technology and always finds time to learn new microprocessor/FPGA systems. His passion led him to Xilinx Zynq 7 System-on-Chip (SoC), and after an extensive research on available Computer-on-Module Boards, he settled for MYIR's Z-turn Board which is one of the best cost per available-peripheral Computer-on-Module boards, available on the market. Joseph's email address is ***pic18f4455@yahoo.com***

In this book, Joseph shared a lot of content on how to work with the Z-turn board, starting from simply creating a project in Vivado to flash an LED, continuing to Detecting Switch inputs, all the way to interfacing the Zynq 7 System on Chip to multiple analogue sensors through multiple XADC channels. All the above-mentioned interfacing is done from both the ARM Cortex A9, commonly known as the Processing System and the Artix 7 FPGA, commonly known as Programmable Logic, both residing within the Zynq 7 SoC.

Being a lecturer, Joseph did not simply show the steps of how to achieve the goals one is set to achieve when using the Z-turn Board, but also pointed out hidden procedures, one has to undergo, while implementing these steps. He makes sure to explain the reasons why, one has to go through the required hidden procedures, and this added material, makes it really helpful for beginners and established engineers alike, to quickly get used to the Z-turn board.

This book includes a lot of C code and VHDL code written by the author himself. This is accompanied by lots of comments and explanations from where the C functions are derived in SDK which for the novice engineer would be quite difficult to understand. All VHDL code is originally written by the author and one has to have a good understanding of how VHDL works to obtain the full benefit of this book.

The Z-turn board is one of the best off-the-shelf Computer on Module Boards available in the market today. It has a vast array of peripherals ranging from very high-speed interface connectors, HDMI, USB etc. It is advisable by the author to invest a little bit more in MYIR's Z-turn Cape IO board which could be connected directly underneath the Z-turn board. This Cape IO board offers better IO capabilities for those users who would like to interface the Z-turn Board to external peripherals such as LEDs, switches, motors, sensors, etc.

Given the vast amount of peripherals present on the Z-turn board and the amount of computer

power present on the Zynq 7, one cannot ignore the potential one can achieve, in areas such as Machine Learning, Machine Vision and AI. By writing this book, Mr. Joseph Attard and MYIR are hoping to make the Z-turn board, the preferred choice for both novice engineers and experienced engineers alike, in their endeavor to learn how to work with Xilinx Zynq 7 SoC.

This work will not stop here and MYIR encourage more and more players to join in sharing knowledge with the general public for a better future.

Catalogue

Getting Started

The first point of reference to start working with the Z-turn board is a Youtube video that shows the link from where to download the board-support-files and how to install them correctly. This is given below:

https://www.youtube.com/watch?v=VDYoweTZtfU

This video will introduce the Github link below, from where to download the board-support-files.

https://github.com/q3k/zturn-stuff

Then the Github site will take you to the following wiki page:

https://reference.digilentinc.com/reference/software/vivado/board-files?redirect=1

The above is the wiki page that shows how to install the board files for Vivado. The following sections show how to install the board support files for Vivado.

Installing the Board-Support-Files

The board files are used by Vivado. These consist of XML files used by Vivado to recognize various development boards. The board files were downloaded from Github link stated above.

The board file folder must be copied to the location shown by figure 1 below:

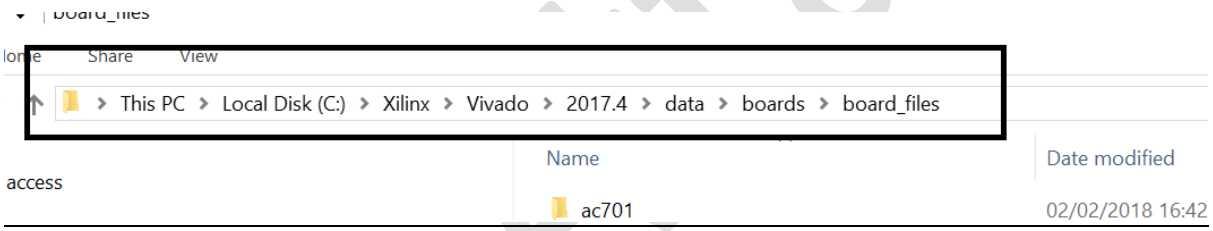


Figure 1: Location where to copy the Board support Files

Figure 2 below shows the copied folder in the ***board_files*** directory within the ***Xilinx*** Directory. The folder must be copied ***as is***! That is, ***<u>do not remove or add any files</u>*** to the copied folder!

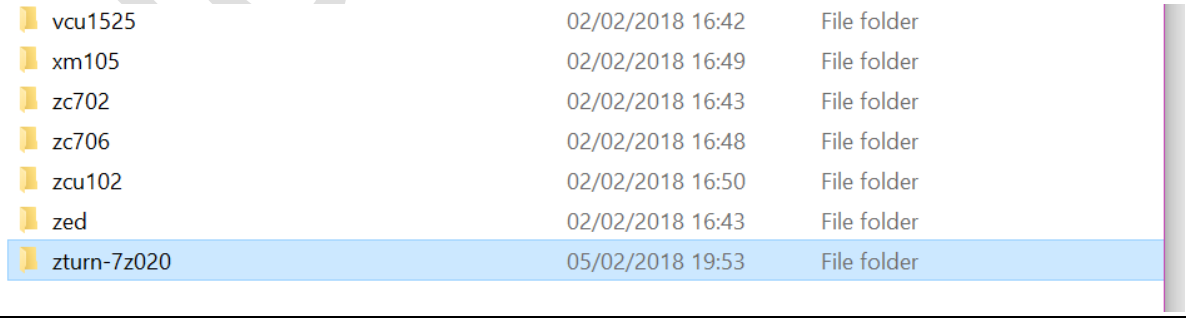


Figure 2: Z-turn folder seen with the other board support files within Xilinx Directory

Figure 3: Github folder contents

Figure 3 above shows the contents of one of the folders downloaded from Github. It is advisable to copy both folders into the Xilinx directory as they are in the folder indicated by figure 2.

The following section shows how to create a project for the Z-turn board in Vivado. The project will only include a simple VHDL module, therefore only the ***Programmable Logic*** part of the ***Zynq 7 System-on-Chip*** will be used. The following sections show the full procedure, right up to programming the ***Zynq 7***.

<u>How to create a project in Vivado</u>

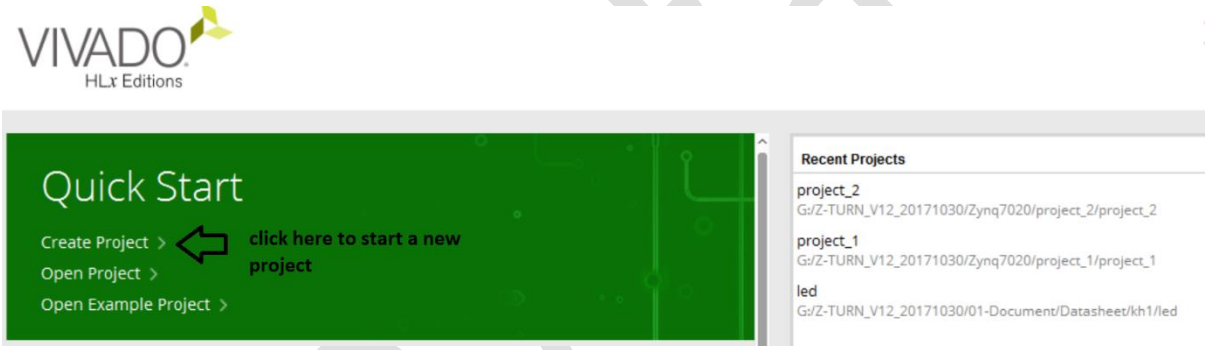


Figure 4: Start page in Vivado

Figure 4 above is self-explanatory, all the user has to do is to left click on *Create Project* link.

A window pops up, click next

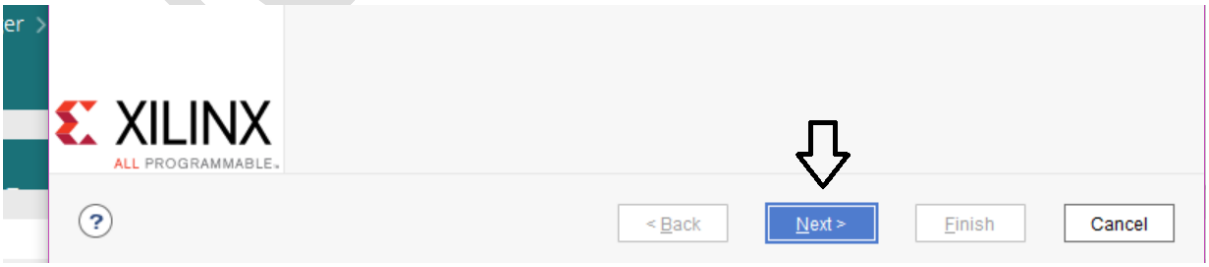


Figure 5: Pop Up Window 1

For the first pop-up window, click on *Next*.

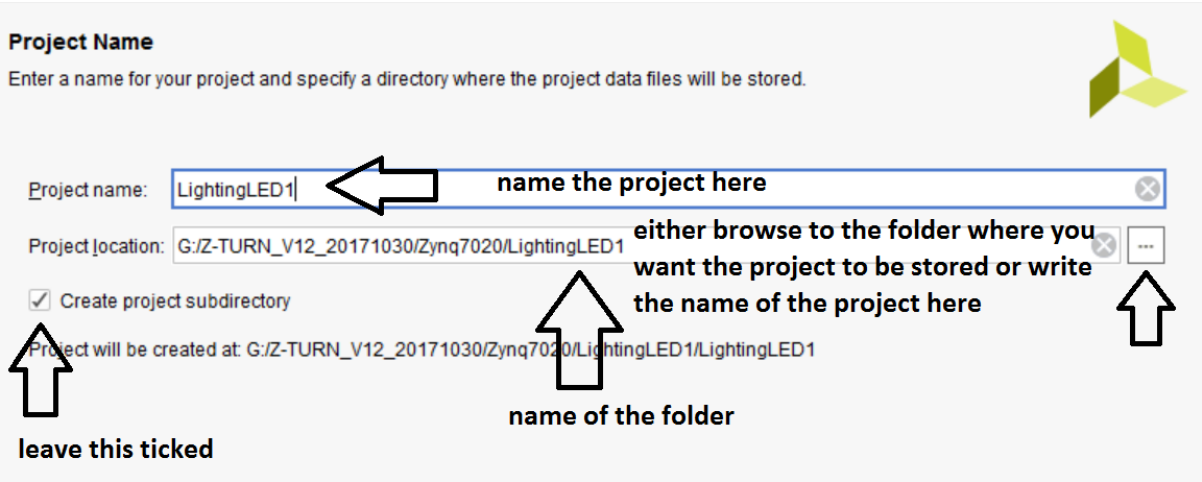


Figure 6: Name the Project Window

Figure 6 shows where to write the name of the new project, and what one needs to do, to store in the desired location within the PC.

Click on ***NEXT*** again.

Figure 7: Project Type Window

Since in this project, a VHDL module is going to be created, then one should leave the window shown in Figure 7 as is. Click on ***NEXT*** again.

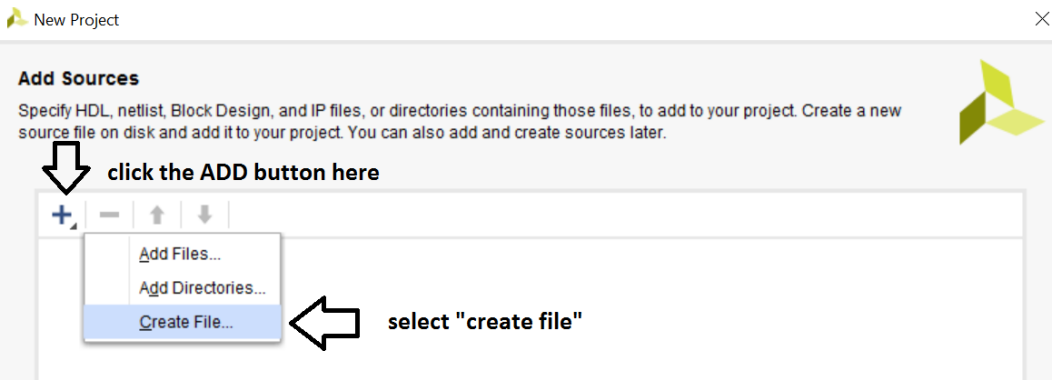


Figure 8: Adding the VHDL source file

In the next window, shown by Figure 8 in the previous page, the user will be asked whether any source files shall be created and/or included in the project. So, one must first click on the ***plus (+)*** sign in the left corner and then select ***create file*** from the drop-down list.

This will lead to another pop-pup window which asks for the name of the module and whether VHDL or Verilog will be used as the preferred language for the module created. Since the author only knows how to program in VHDL, then the file type is going to be VHDL. The file should be given a name that is related to the function of the module. In this case, since the VHDL module is going to light an LED, the name implies the module's function! All this is shown in figure 9 below:

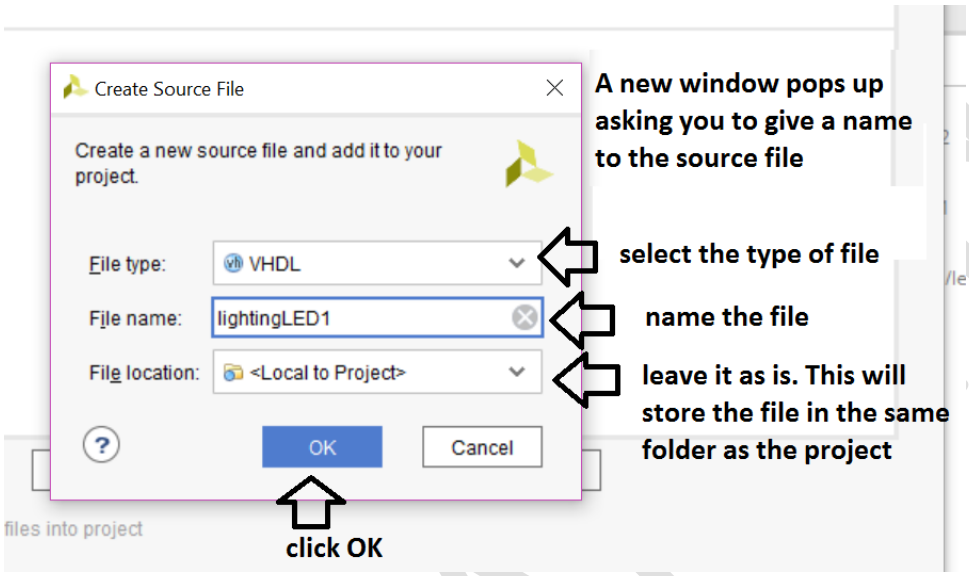


Figure 9: Give a name to the VHDL module

Figure 10 below shows the new VHDL module is part of the project. One could add as many modules as needed. This could become handy if a top-down approach is used to build the system.

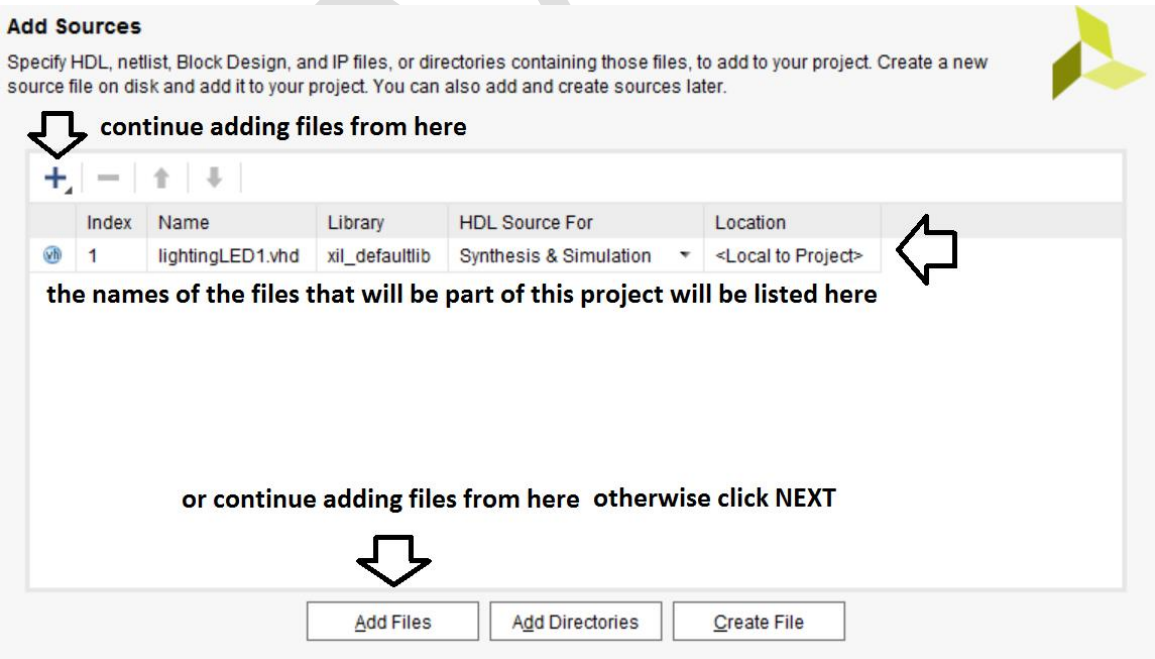


Figure 10: Add source files to the project

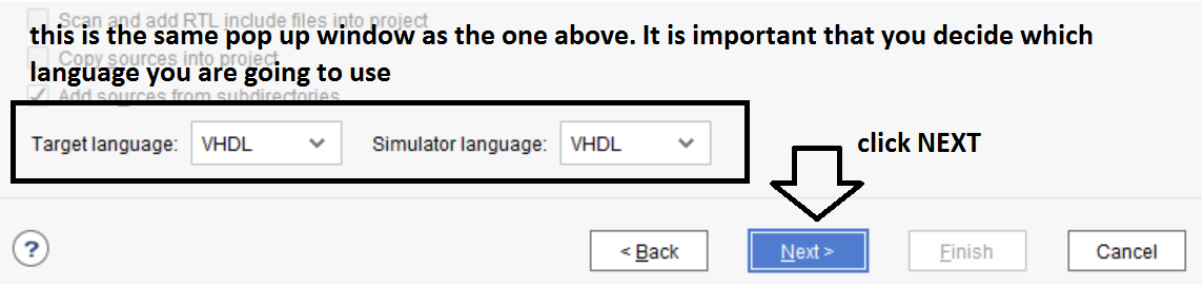


Figure 11: Choosing the language used in the source file

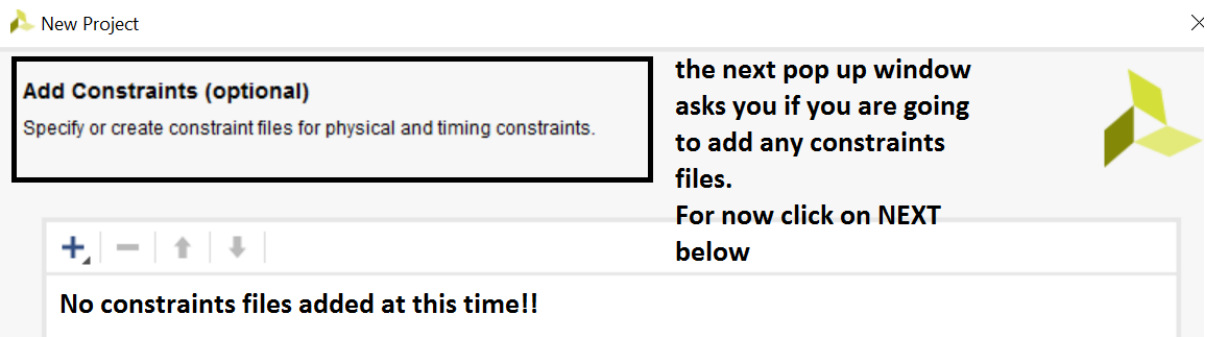


Figure 12: Constraints File

Vivado will then ask whether a constraints file should be added. Since the board support files have been included, one does not need to include a constraints file at this stage, so click ***NEXT*** for this window, without doing any changes.

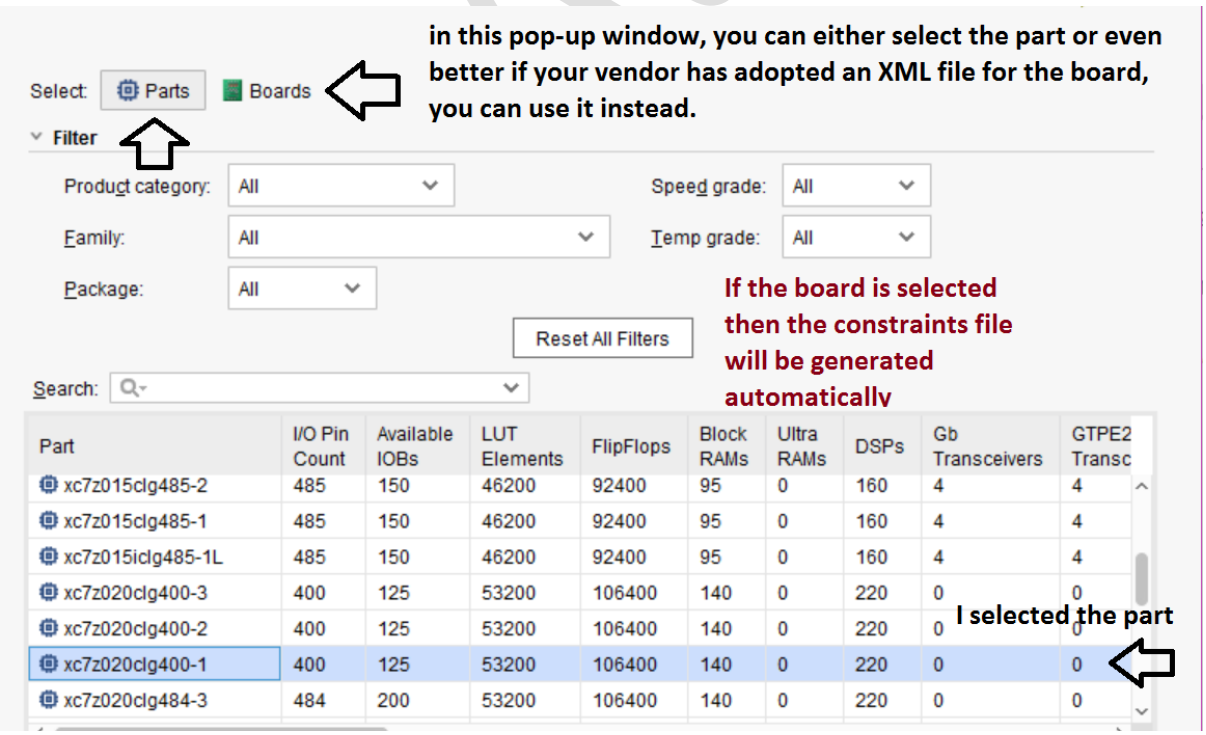


Figure 13: Choosing the Zynq 7 SoC according to part number

In the next pop-up window, one can either choose the Zynq 7 according to the part number resident on the development board, or even better one can choose the board itself by first clicking on the

boards tab and then choose the Z-turn board from the list. This is only possible ***if*** the Board Support files of the relative board are included in the Xilinx folder as described earlier in this chapter.

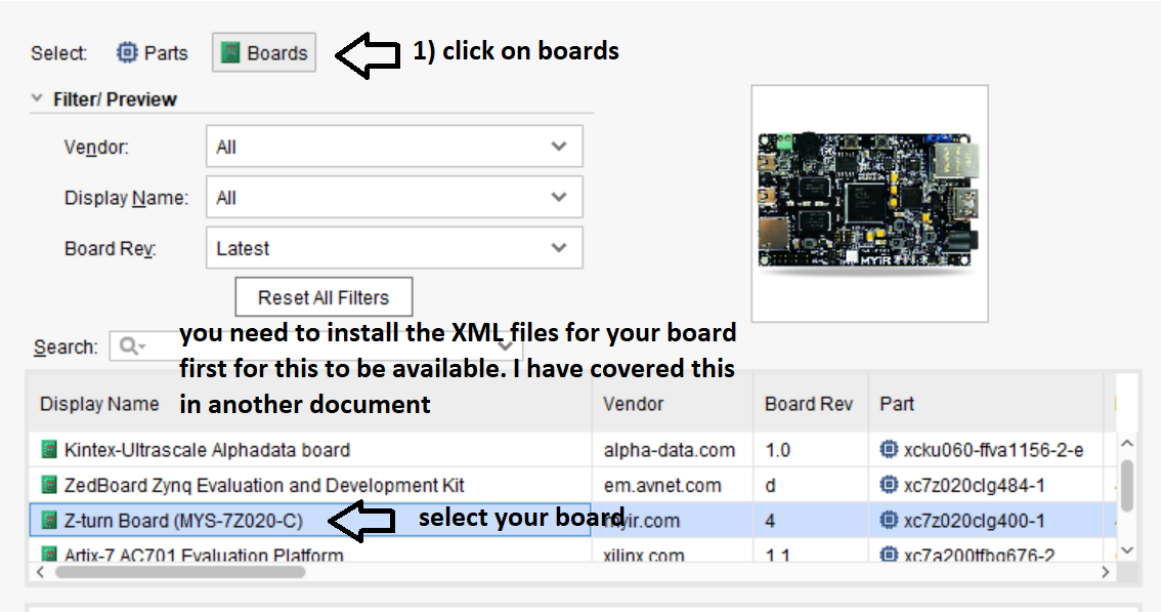


Figure 14: Selecting the Z-turn Board from the list

Click on ***NEXT***.

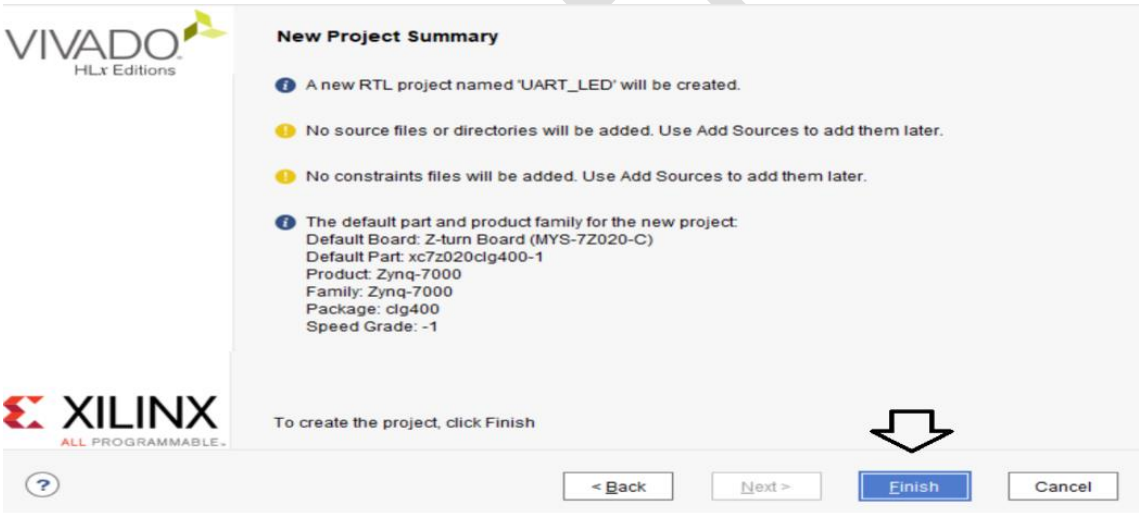


Figure 15: Project Summary

Figure 15 is the last window that pops up while setting up the project. It contains a summary of all the previous settings done. All that needs to be done is to click on ***FINISH***.

Once ***FINISH*** is pressed, a new window pops up. In the new window, one can enter the inputs and outputs of the VHDL module that was created before.

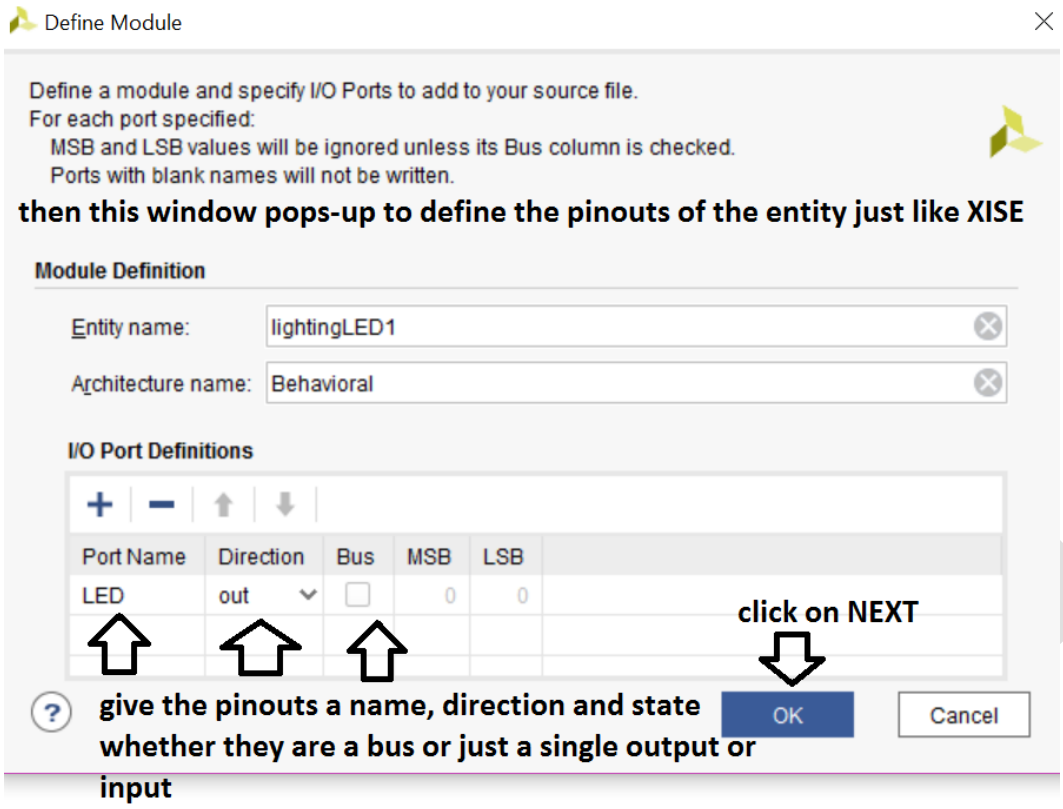


Figure 16: Define the inputs and outputs of the VHDL module

Now, at this point, one might not know exactly how many inputs and outputs, the module might end up with, however if one has any idea of any common inputs and outputs that the module might have(such as the clock and the reset inputs), one could include them immediately in the table shown in figure 16 above. However, if currently, the user does not have any idea what the names of the inputs/outputs are going to be, it is perfectly safe to just click on the ***OK*** button and continue with the next pop-up window without submitting any names.

Vivado opens and one can find the source files as shown in *Figure 17* below.

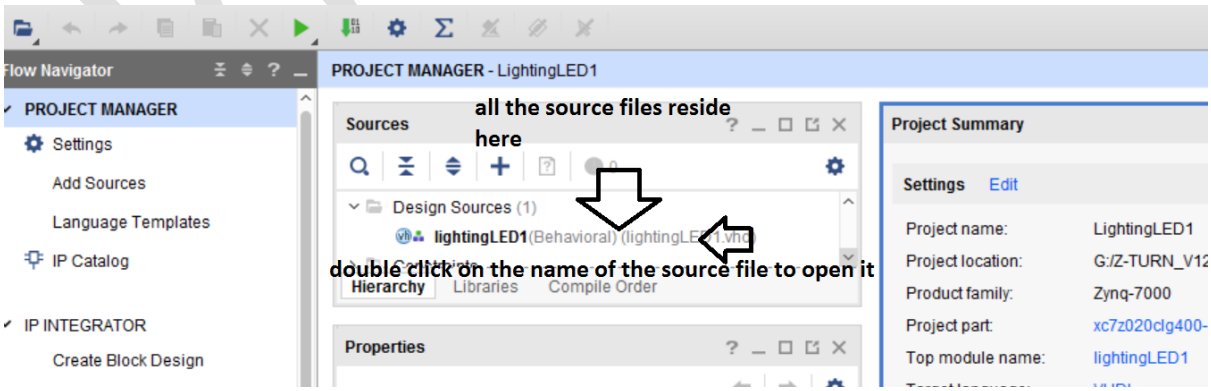


Figure 17: Location of the Source Files

Double clicking on the VHDL source file so that one can write the VHDL code that eventually will be translated into hardware later by Vivado.

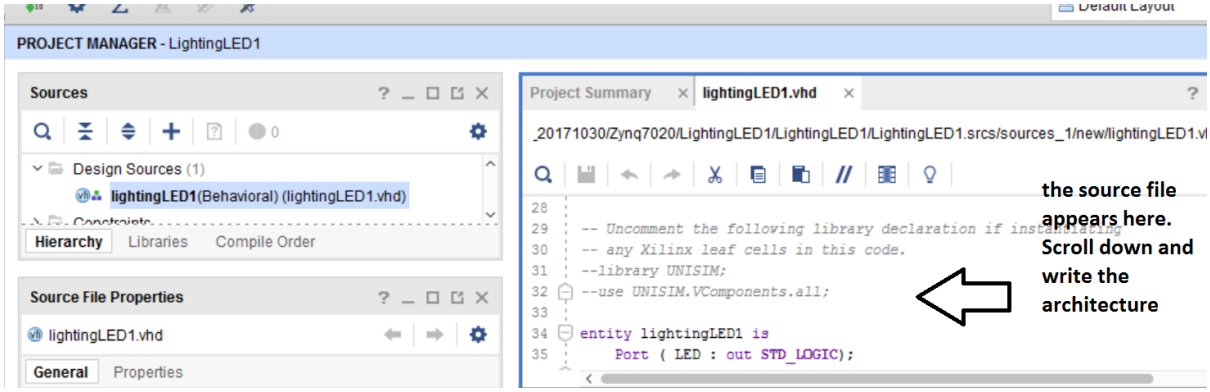


Figure 18: Typical VHDL source file

Figure 18 and Figure 19 show the same VHDL file and code. In this project, an LED will be lit. It is a simple instruction, however at this point, the objective of this chapter is to show all the steps needed to develop a Zynq 7 project that will operate only the Programmable Logic part.

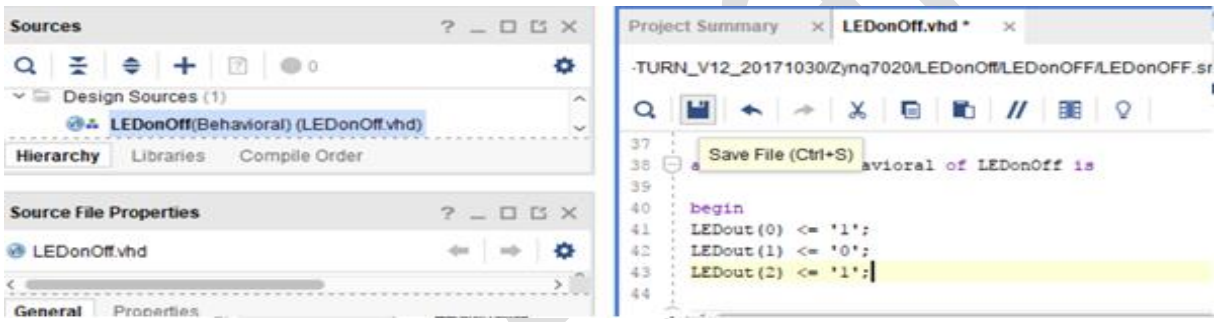


Figure 19: Simple VHDL instruction

Once the VHDL code is written, the code must be saved from the icon shown in Figure 19. By saving the file, Vivado is also checking the syntax and if there is any syntax error, Vivado will pop up a window.

Creating a Block Design

The next step is to create a ***Block Design***. This approach is the easiest approach one should take when developing a project for the Zynq 7 System-on-Chip especially because the board support files of the Z-turn board are already included in Vivado. That way, Vivado would know the features and parameters of the Z-turn board and would give warnings or error messages if one would try to use hardware that is outside the hardware settings of the Z-turn board. Figure 20 on the next page, shows the steps one has to go through to create a block design. Once the block design has been created, a new file-type will be generated by Vivado where a ***schematic representation*** of the hardware could be drawn. This replaced the *canvas* in Xilinx ISE. The next step is to include the processing system as shown in figure 21 on the next page.

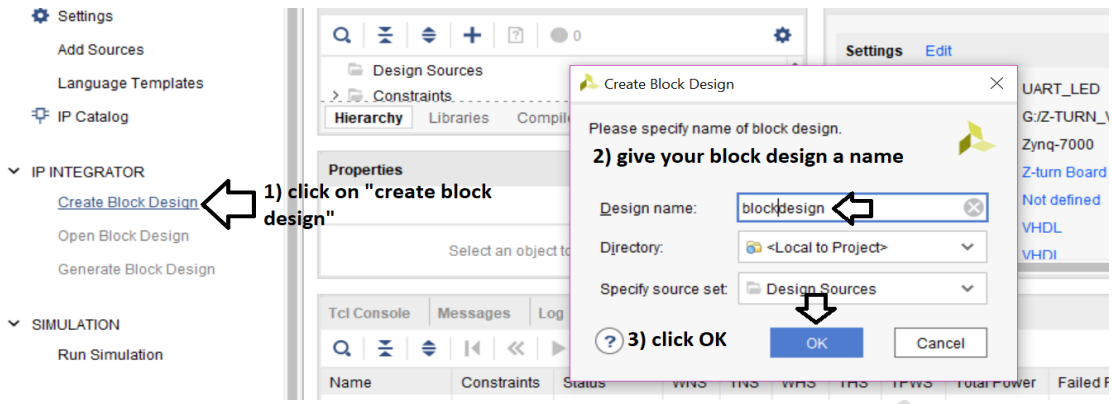


Figure 20: Creating a Block Design

<u>Why do we have to include the Zynq Processing System in our Design?</u>

The reason for including a ***Zynq Processing System*** in our design is because ***it is the only way*** how one can download the VHDL code to create hardware in the ***Programmable Logic*** part! This means that when the ***boot-image file*** is created later in SDK, this will be fetched by the Processing System of the SoC and after reading it, the hardware part will be configured in the Programmable Logic. So even though in this example, only the Programmable Logic part is going to be active, the Processing System part must also be included in the hardware design! Another reason for including the Zynq Processing System in the hardware design is because the 100 MHz clock required by the sequential circuits within the Programmable Logic part can only be provided by the Zynq Processing Part! Figure 21 shows the three steps needed to create a Zynq Processing System. There are ***two ways*** how to add the Zynq Processing System both shown in Figure 21.

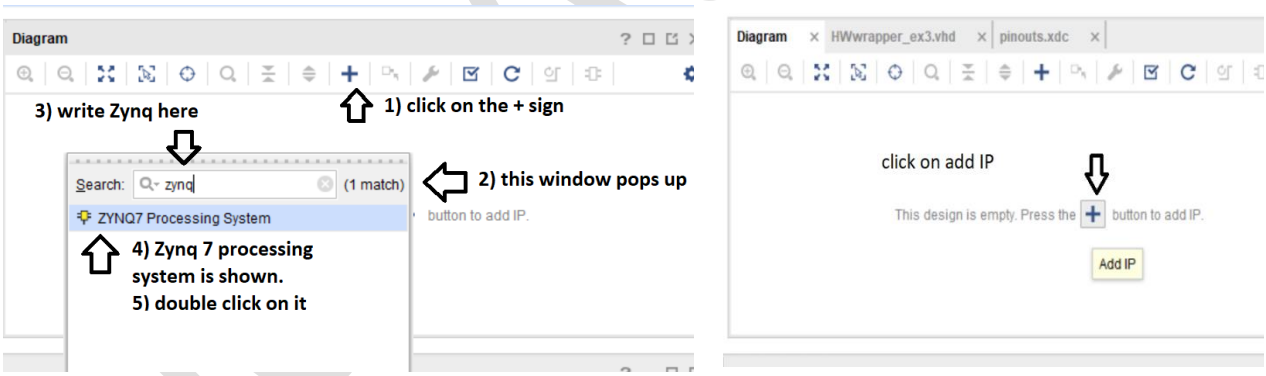


Figure 21: Including the Processing System

Figure 22: Zynq Processing System on Canvas

Now that the Zynq processing system is included in the schematic, it would be a good idea to click on ***Run Block Automation*** as shown in Figure 22. This will enable the board settings as provided by Github and therefore

establish the peripherals that are already connected on the Z-turn board. Figure 23 below shows a pop-up window that asks the user to confirm the pre-set board settings.

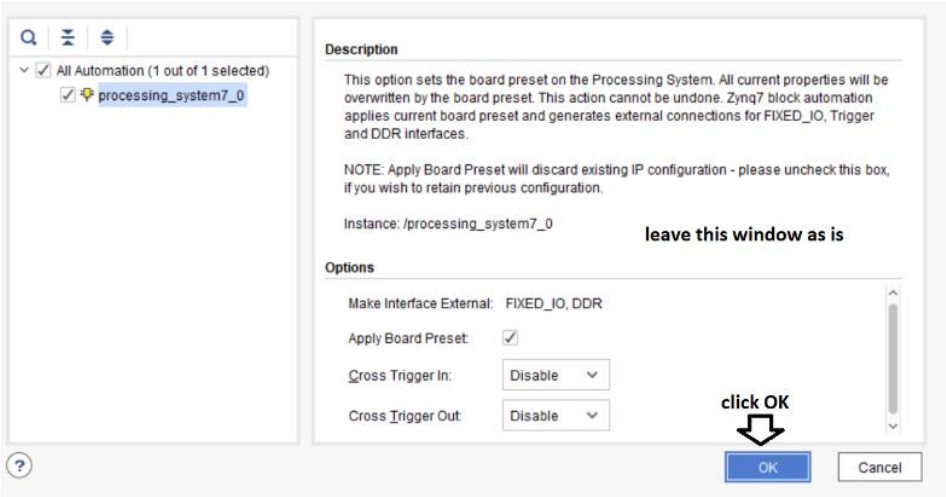


Figure 23: Preset Z-turn Board Settings

Just click ***OK*** on this window and leave everything as is.

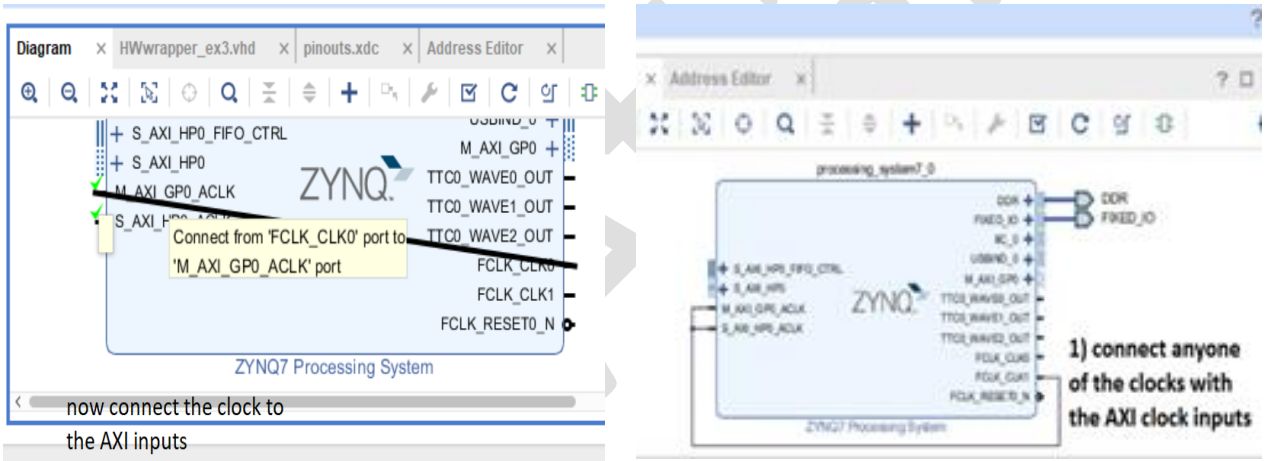


Figure 24: Expanded Zynq Processing System Block Diagram

Figure 24 shows the expanded version of the *Zynq Processing block* after the ***Run Block Automation*** has been enabled. Not to generate any errors, the next step is to connect the ***FCLK_CLK0*** which is the main ***100 MHz*** clock output of the Zynq Processing System, to the AXI inputs at the left of the Processing System block.

Next include the VHDL module written previously. This is done by right clicking on the canvas or Block Design window, select Add Module from the list. A new window pops up with the suggested VHDL

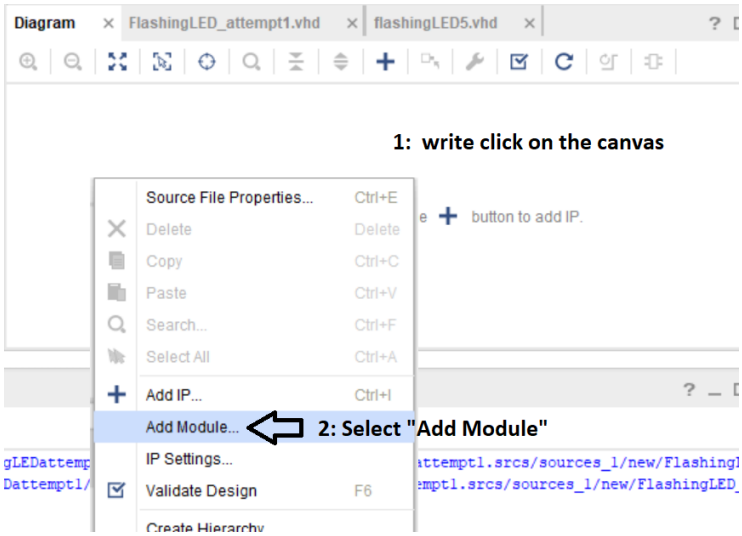


Figure 25: Adding a VHDL module to the schematic

modules highlighted. Double click on the module and it will be added in block form in the schematic diagram.

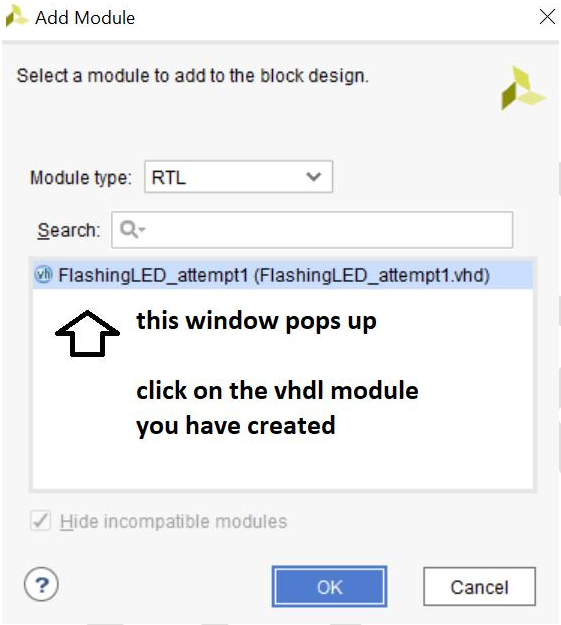


Figure 26: Selecting the VHDL module

Figure 26 shows the VHDL module that can be converted into block diagram to be added to the schematic diagram o f the project. It must be noted here that if more than one VHDL modules have been written and are part of the project, all of these modules will be listed in the lower part of this window. One can select which one of the modules could be included in the schematic.

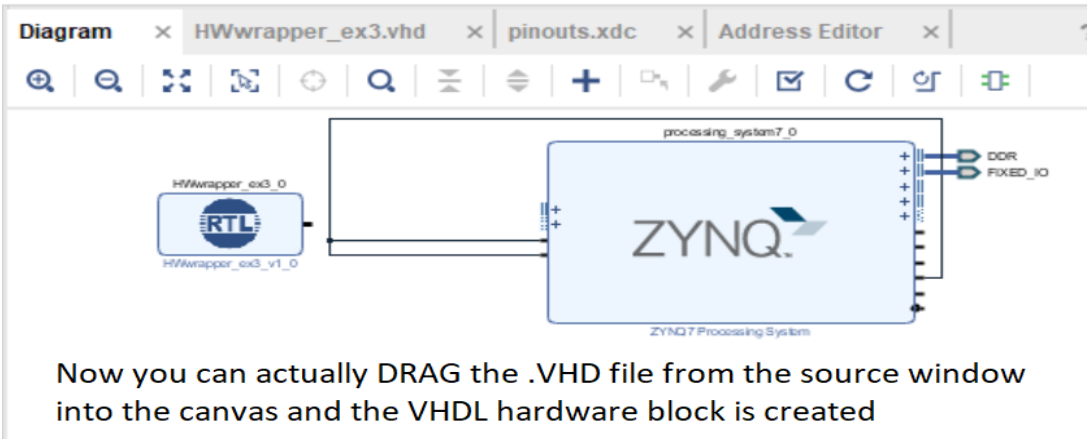


Figure 27: VHDL module in schematic

Figure 27 shows the VHDL module is part of the schematic. Since in this particular project, an LED is only to be lit first and then code could be changed for the same LED to flash instead of just having the LED lit steadily, there is no need for a clock input to the VHDL block - it is going to be free-running! That is, it has nothing to do with the PS part of the SoC!

Creating a Hardware Wrapper

After deciding on which VHDL modules are going to form part of the system, the next step is to create a ***hardware-wrapper***. This is another technical name for the ***top-level module*** of a system where it will include all the components of the system. Failing to do the steps in Figure 28 will result in an incomplete system and therefore will not work in practice!

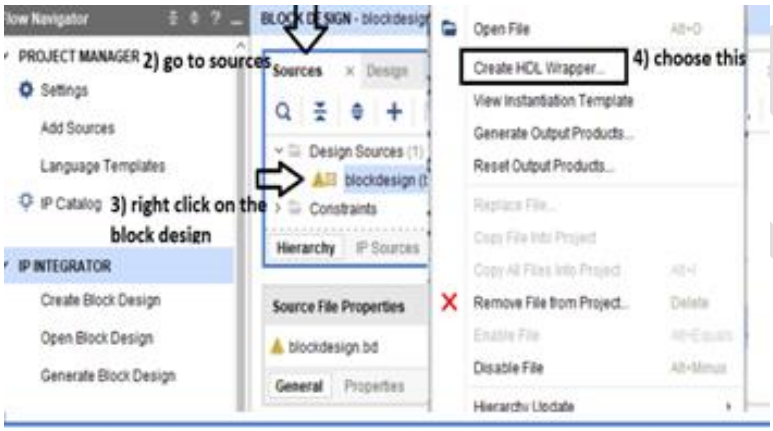


Figure 28: Creating a Hardware Wrapper

Once the hardware wrapper is created, it is time to synthesize the design. Figure 29 shows how to synthesize the design by clicking once on run Synthesis.

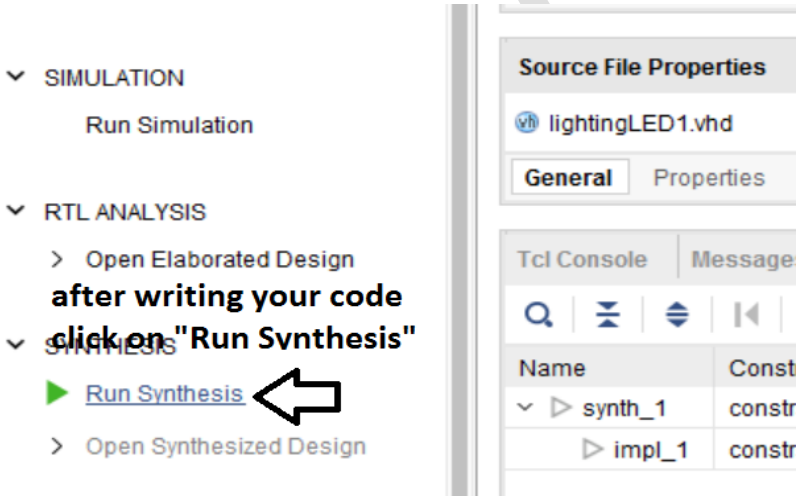


Figure 29: Synthesize the code

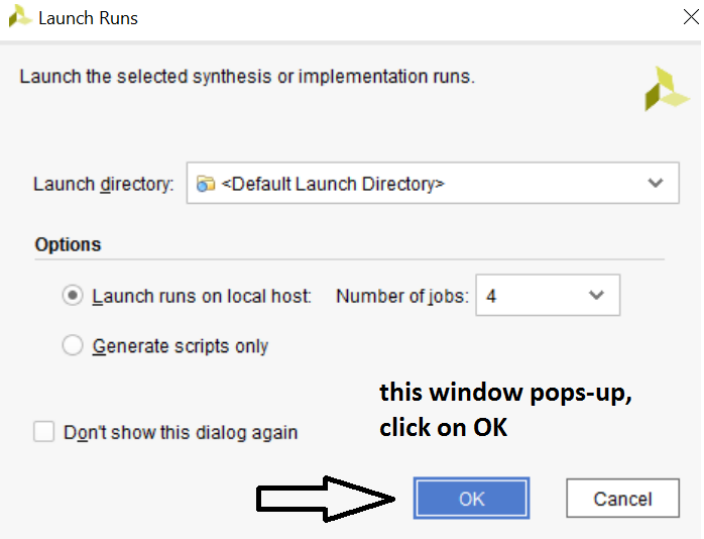


Figure 30: Launching Synthesis

The window of Figure 30 pops up, Click on ***OK*** because it is better to let the default settings as they are.

Leave the synthesis to complete. Figure 31 show the design runs.

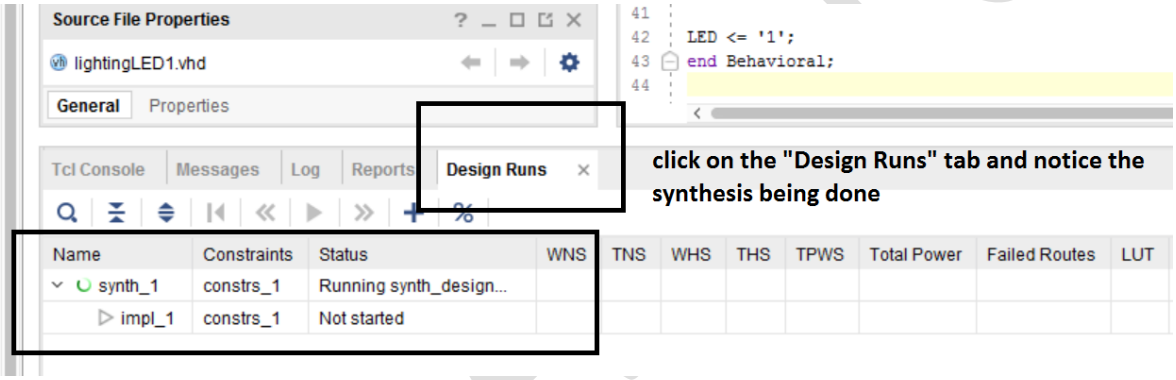
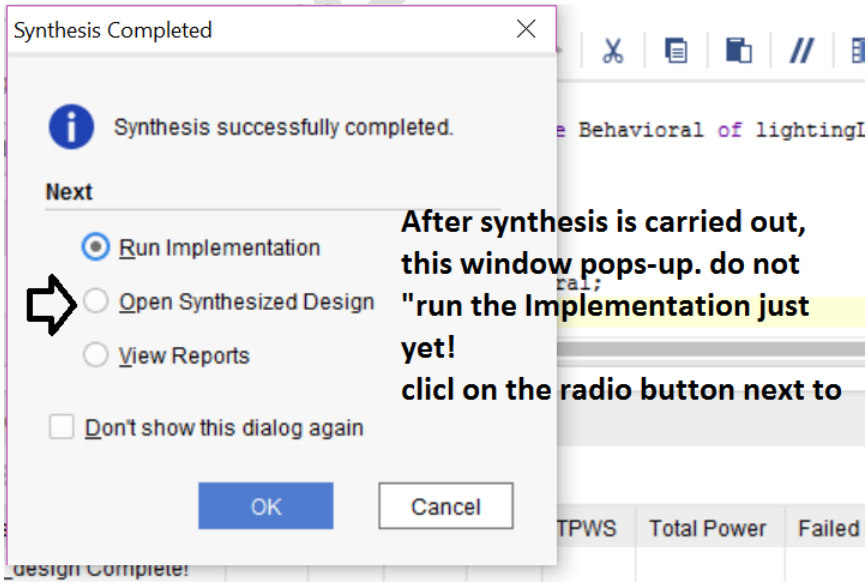


Figure 31: Design Runs



Once Synthesis is done, even though Vivado suggests to ***Run Implementation***, it is a good idea to ***Open Synthesized Design*** because at this point, it would be a good idea to assign the external pins to the inputs and outputs.

Figure 32: Synthesis Ready window

Assigning Pin numbers to the System

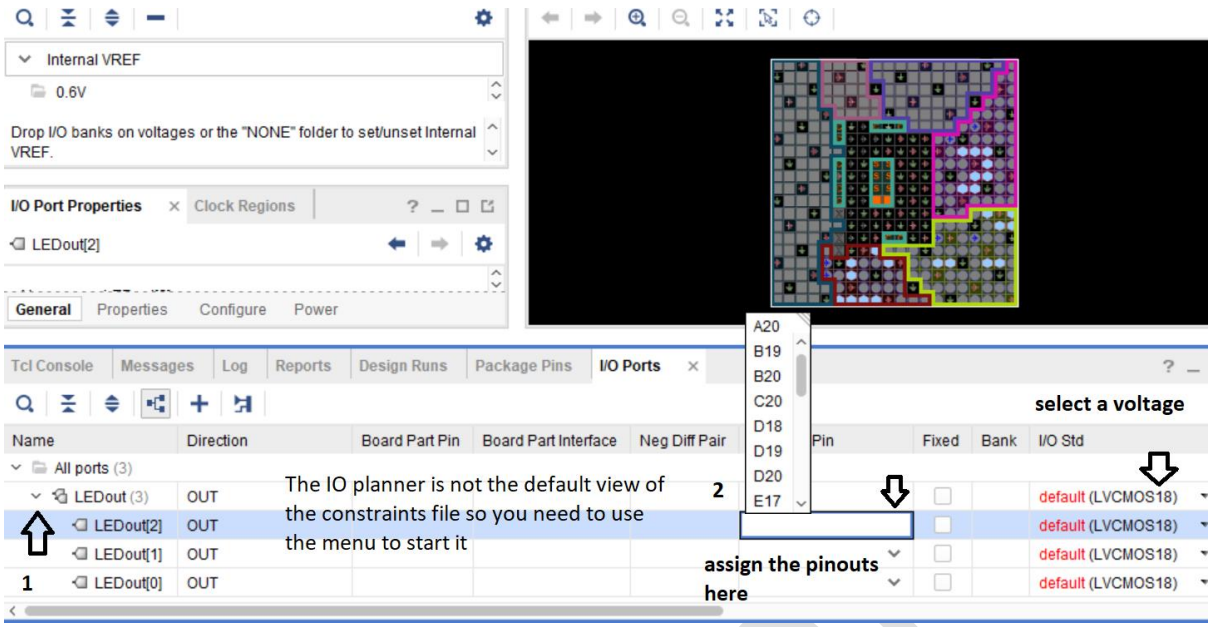


Figure 33: Assigning Pin numbers

Figure 33 illustrates one way to assign pin numbers to the inputs/outputs of the system. Since in this system, there are only three LEDs used, only these LEDs must be connected. After opening the Synthesized model, click on the ***I/O Ports*** tab and adjust the pin numbers according to the Z-turn board circuit diagram.

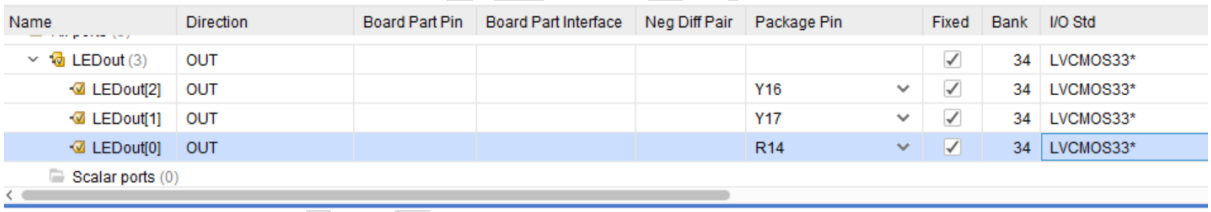


Figure 34: Assiging Voltage Levels to pins

Figure 34 shows how to assign voltage levels to pins. These must be ***LVCMOS33*** not to generate any errors meaning that the outputs are capable of outputting ***3V3***. Also, the *square boxes* under *Fixed* column must be *ticked* not to generate any misleading errors!

Now save the new constraints file as shown in Figure 35.

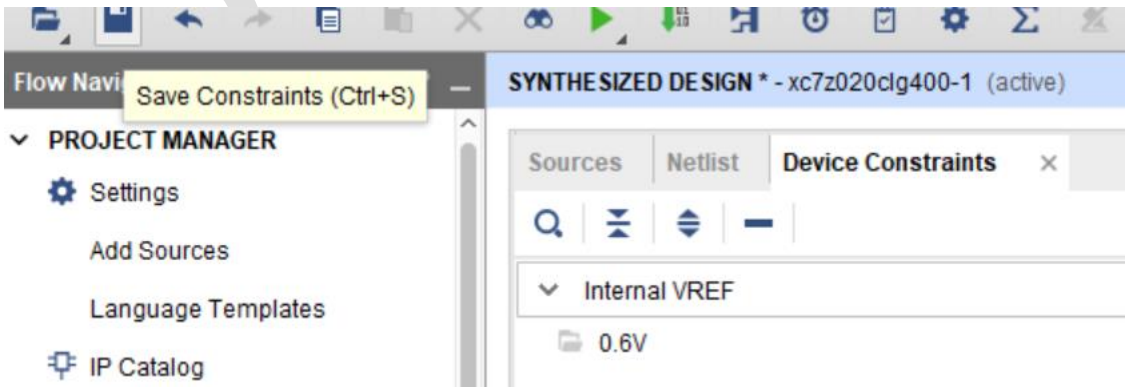


Figure 35: Saving the new constraints file

A pop-up window just like the one shown in Figure 36 shows up. Click on ***OK***.

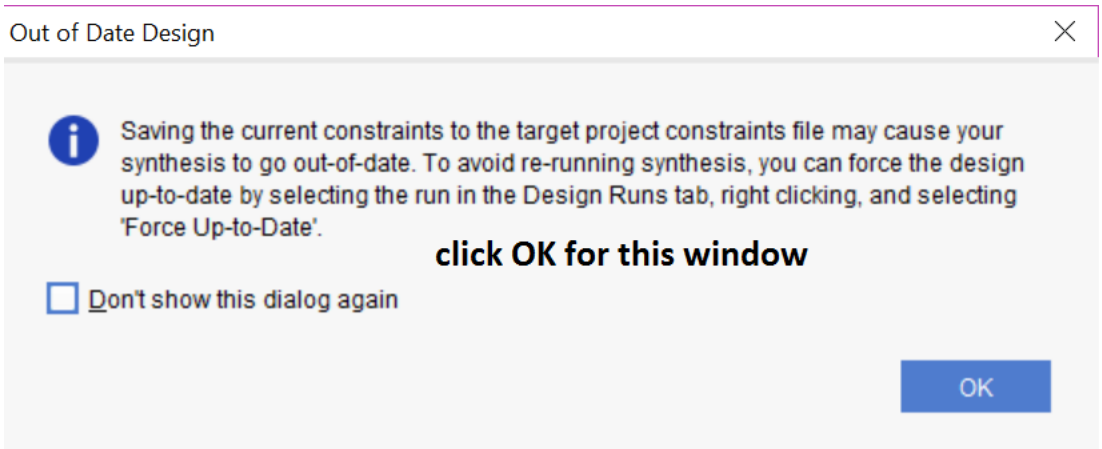


Figure 36: New Constraints File Warning Window

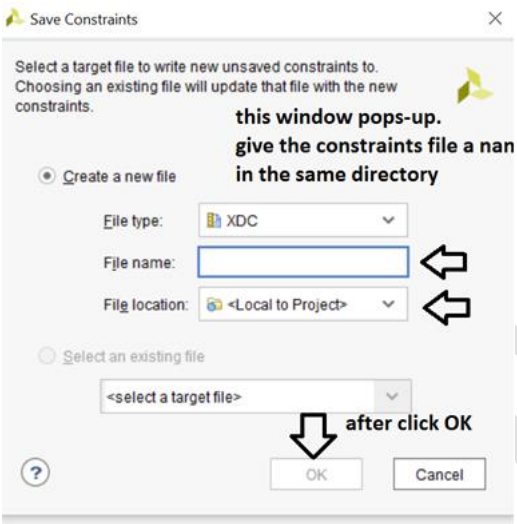


Figure 37: Naming the new constraints file

This means that since the constraints have been changed, a new constraints file will be generated - all there is to do is to give it a name - Vivado will take care of the rest.

Figure 38 shows the new constraints file name is pinouts.XDC and it is now part of the source file list.

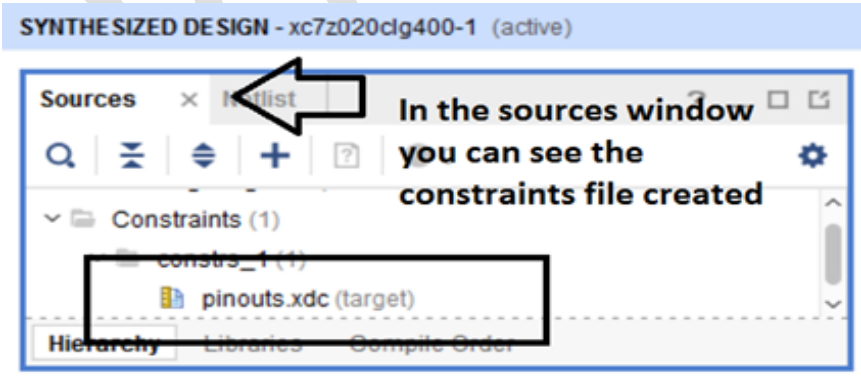


Figure 38: The New constraints file forming part of the source files

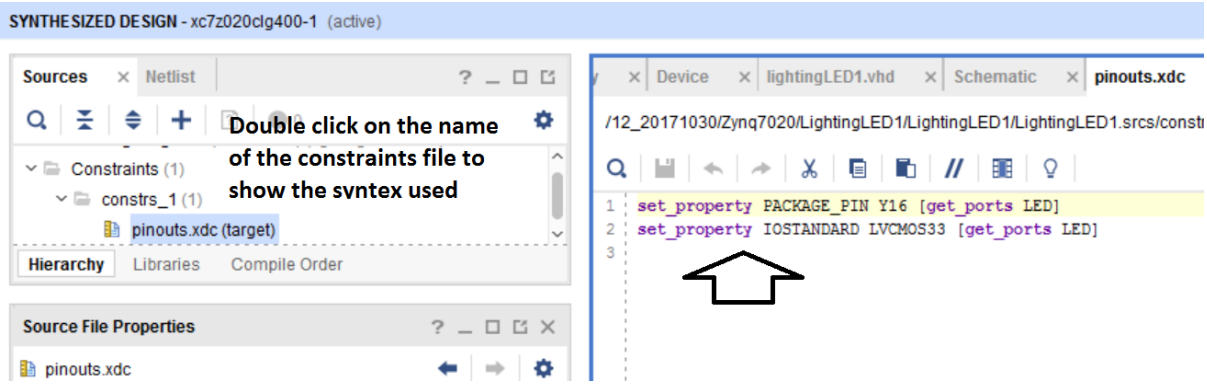


Figure 39: Generated Syntax for new constraints file

Figure 39 above shows the new way how to initialize the pinouts in Vivado. This is very different from Xilinx XISE code, so it is advisable to let Vivado do all the work!

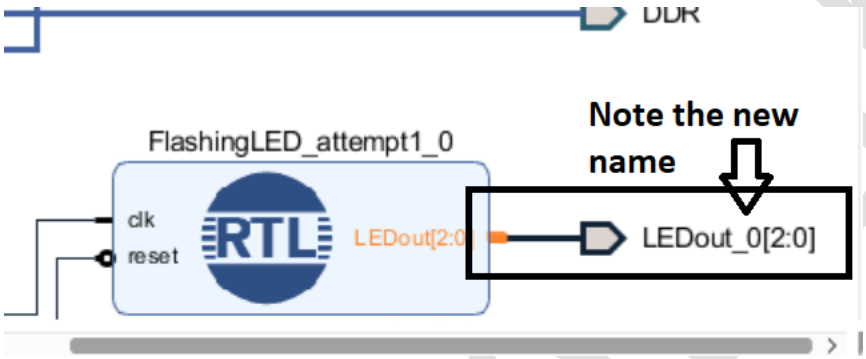


Figure 40: Naming the pins

Note in Figure 40 above the _0 naming convention. This is due to creating a ***hardware wrapper*** which is necessary for the bitstream to be generated correctly. So this indicates that the hardware wrapper has been created correctly and now one can proceed to generating the bitstream file.

However, to generate the bitstream file one has to re-synthesize again. However, this time, instead of running synthesis, one can ***run implementation*** as shown in Figure 40.

Figure 40 also shows that the old synthesis files are all out-dated and that new ones have to be re-generated due to the changes done in the constraints file.

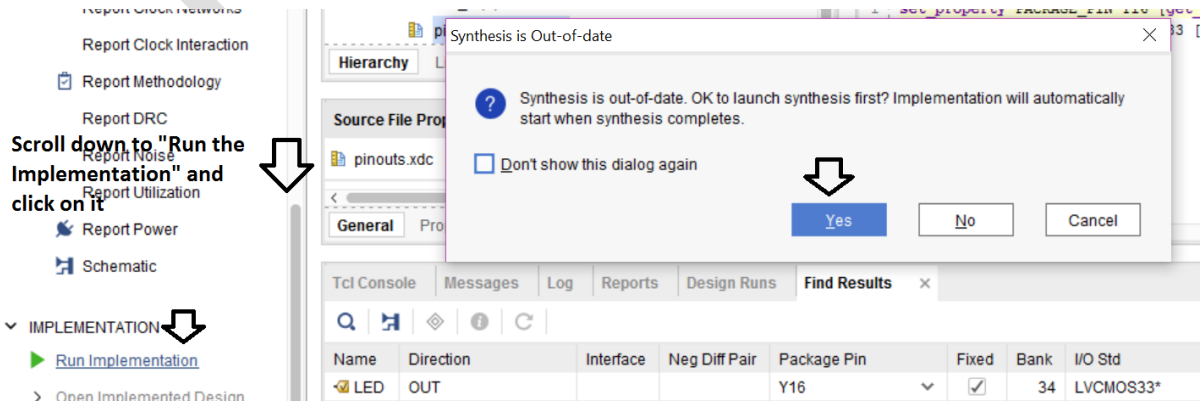
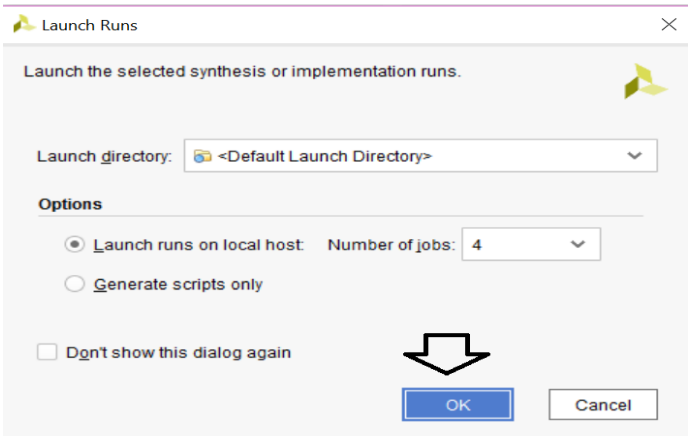
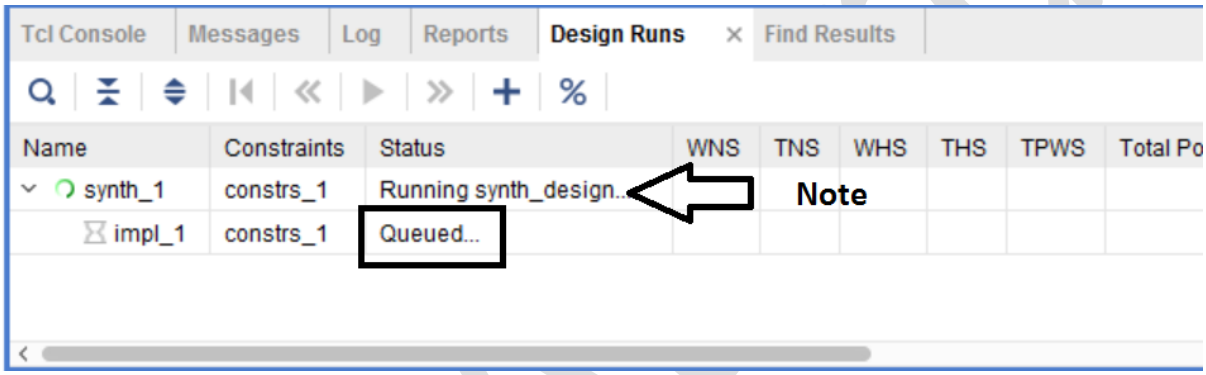


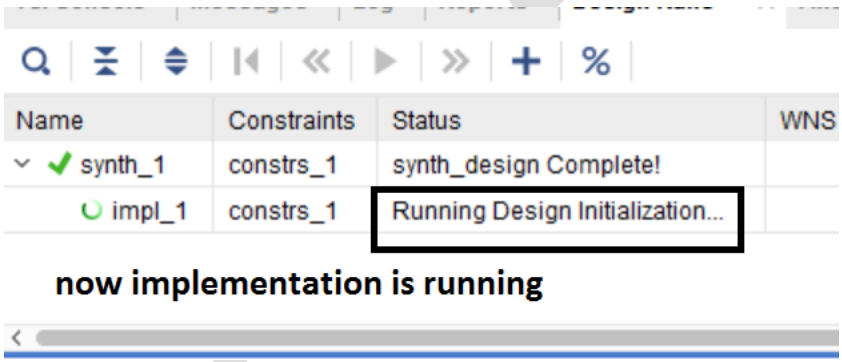
Figure 41: Launching synthesis again



click on ***OK*** again.

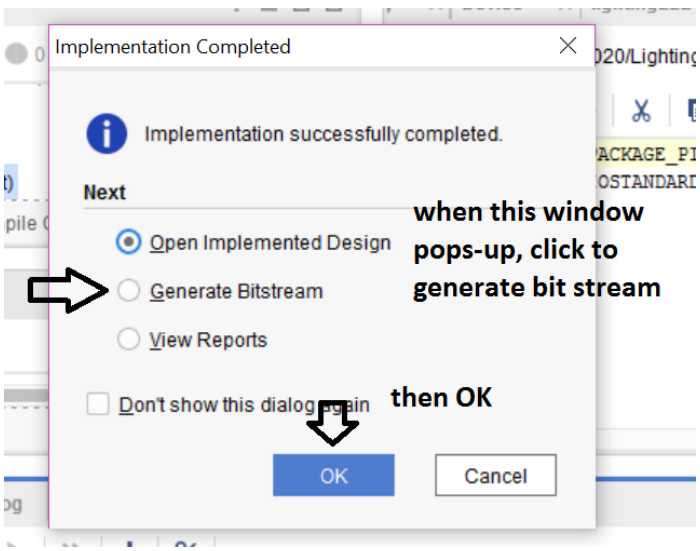
Figure 42: re-launching synthesis again





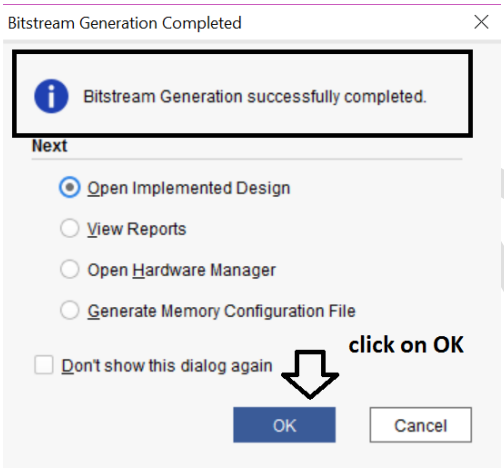
In these snapshots one can see Vivado going through the synthesis stage and also through the implementation stage. Once these are done without any errors, the window of figure 43 pops up.

Figure 43: running through the synthesis and implementation stages



Once the window of Figure 43 pops up, choose the ***Generate Bitstream*** radio button and then click on ***OK*** to generate a bitstream file.

Figure 44: Implementation ready



Wait for the bitstream file to be generated and once successful, do not open the Implemented Design but click on ***OK.***

Figure 45: Bitstream file ready

<u>Export Hardware</u>

Once the bitstream file is generated, the next step is to export the hardware. This is done by making sure that Vivado is in the dashboard, that is, there are no files opened. Click on File, then scroll down to Export, select Export Hardware from the list. This is shown below.

Figure 46: Export Hardware

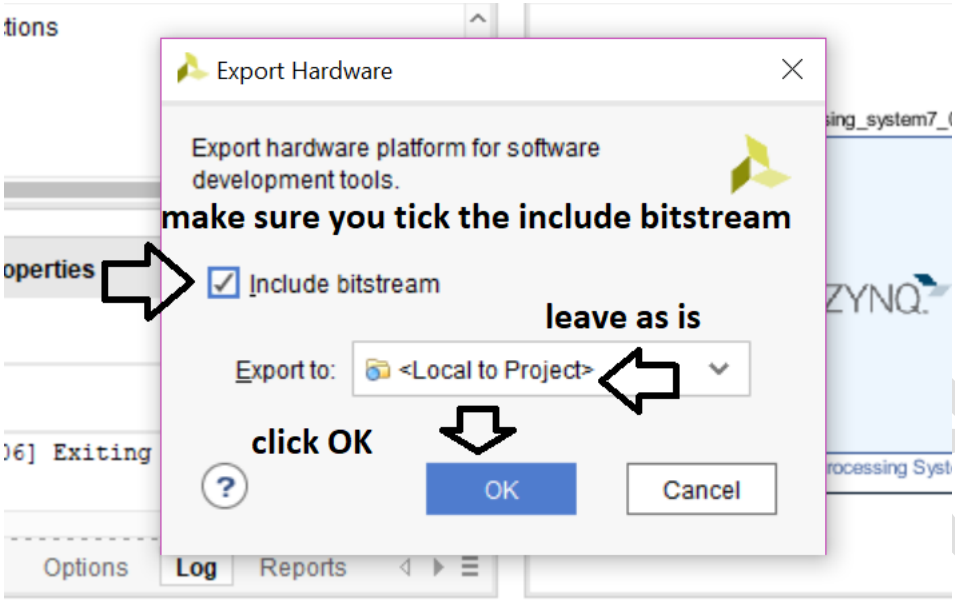


Figure 47: Include the Bitstream file

Make sure to include the bitstream file by ***ticking the square box*** before hitting ***OK*** below!

<u>Launch SDK</u>

The next step is to ***Launch SDK*** from within the project itself. This is done by once again, clicking on ***File***, scrolling down to ***Launch SDK*** and click on it. It must be mentioned here that if one is using a ***13" laptop*** or a small screen, it can happen that ***Launch SDK*** will ***not*** be immediately ***visible***. If this happens, all that must be done is scroll down to the last option on the ***File-List*** and then use the ***arrow-down key*** to reveal the ***Launch SDK*** option.

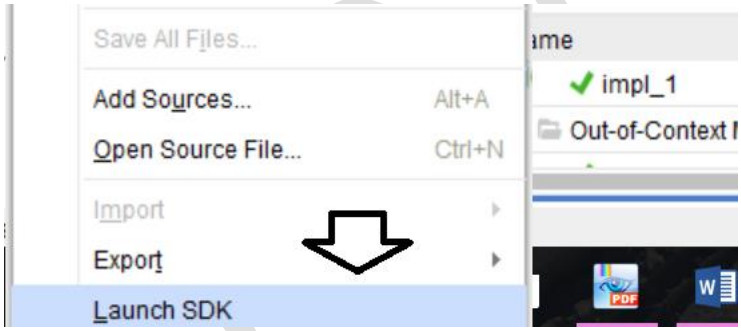


Figure 48: Launching SDK

The next window pops up, click on ***OK***. This means that the folders and files created by SDK will be stored in the same directory as the Vivado project itself.

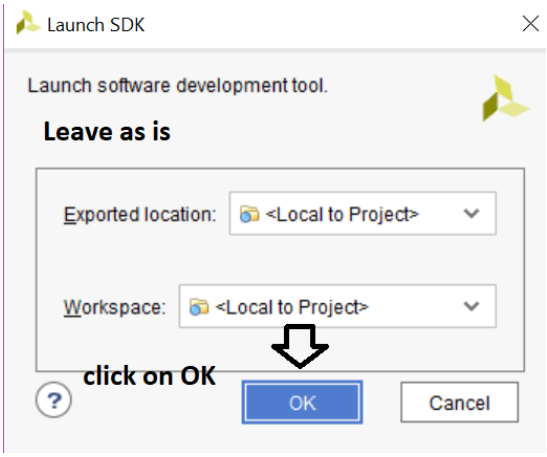


Figure 49: Store the SDK project within the Vivado Project

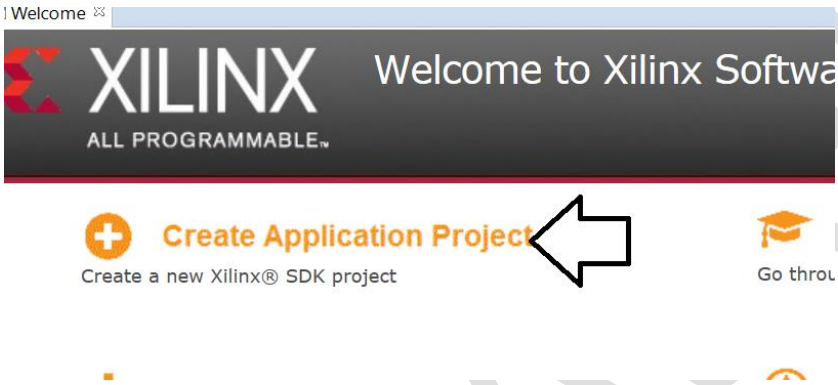


Figure 50: SDK opens

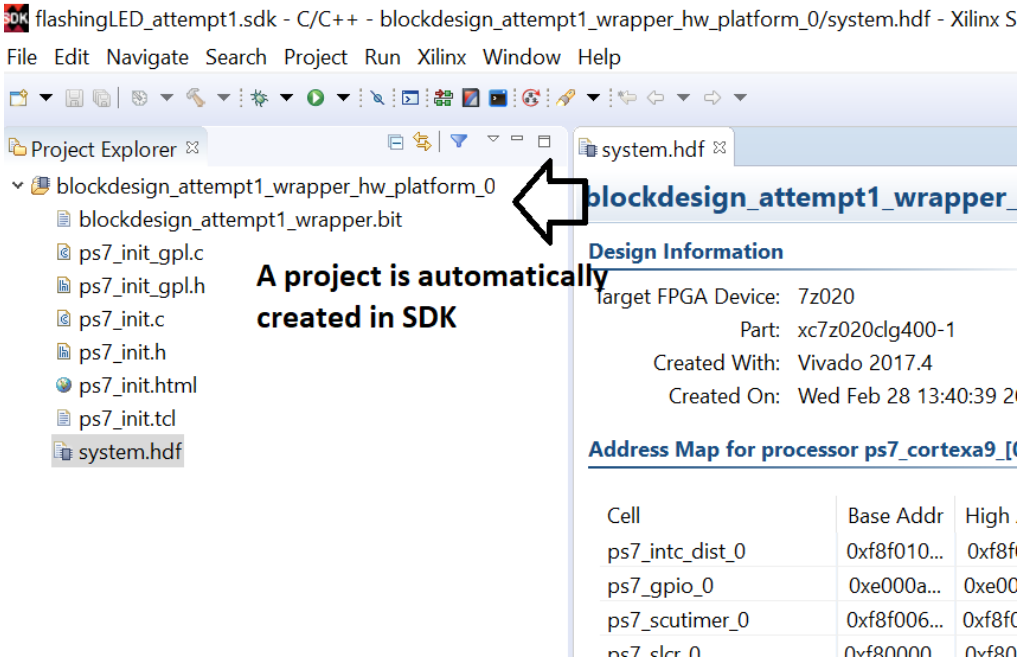


Figure 51: An SDK project is created automatically

Now that SDK is opened, one must create a ***First Stage Boot Loader File*** in short ***FSBL*** file. This file is a bootloader file and once copied to the SD card, and the Zynq 7 is powered up, it will look for this bootloader file to start operating.

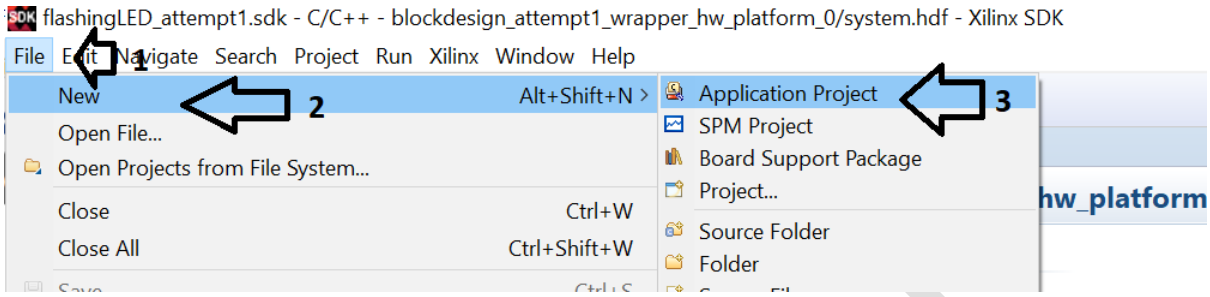


Figure 52: Creating a new application

Click on ***File***, then select ***New***, a sub-menu appears on the right, click on ***Application Project***.

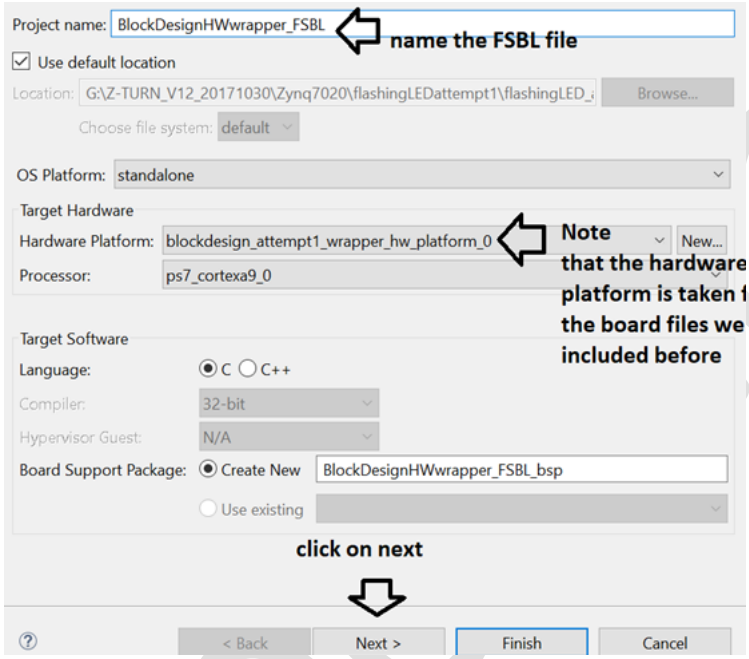


Figure 53: Naming the FSBL project

Figure 53 shows the new window that pops up. All that must be done is just give it a name. It is recommended to include the letters FSBL in your project name so that one can distinguish it from the C project that could be generated later!

As Figure 53 suggests, click on ***NEXT***.

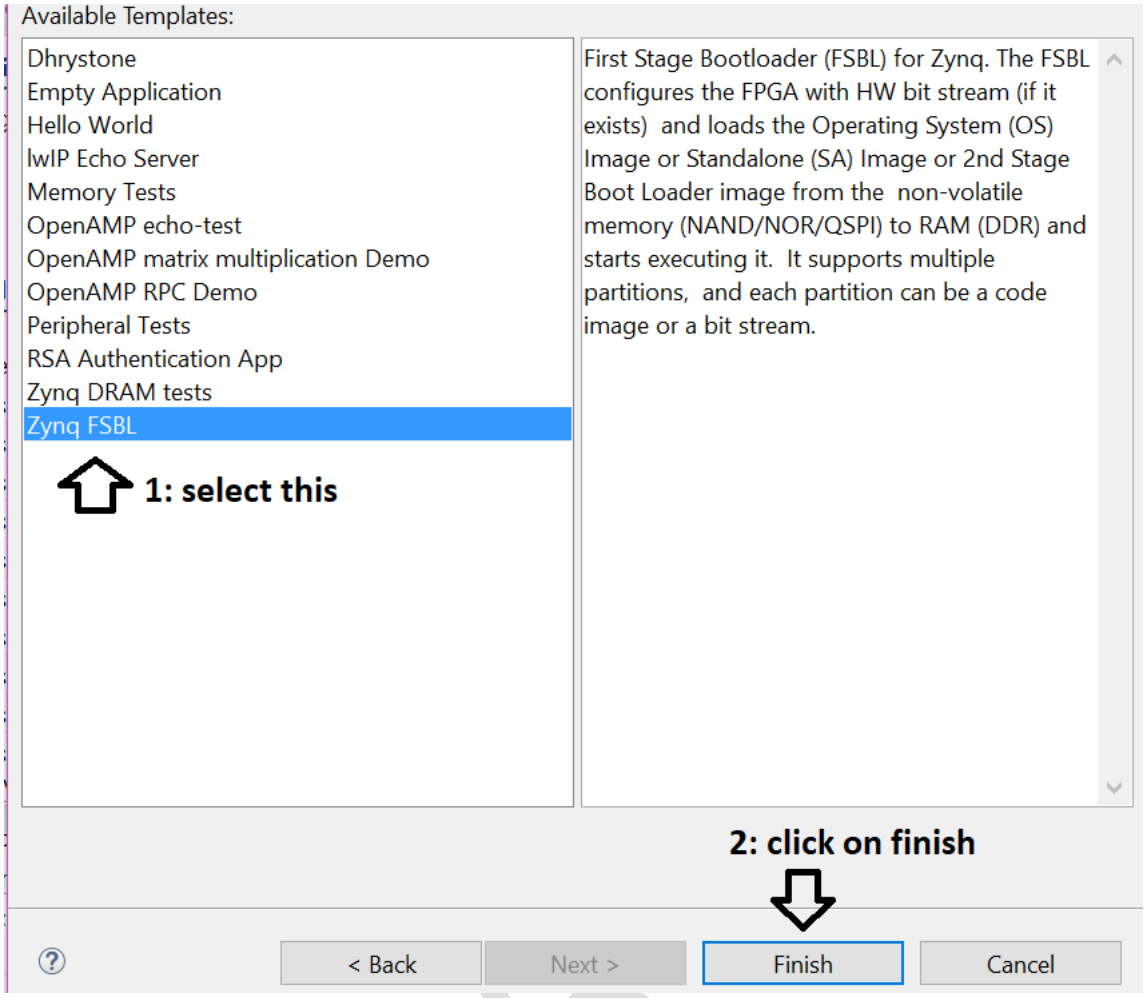


Figure 54: Selecting the type of the project

Figure 54 shows how the FSBL project type is selected. So, highlight ***Zynq FSBL*** first and then click on ***FINISH***.

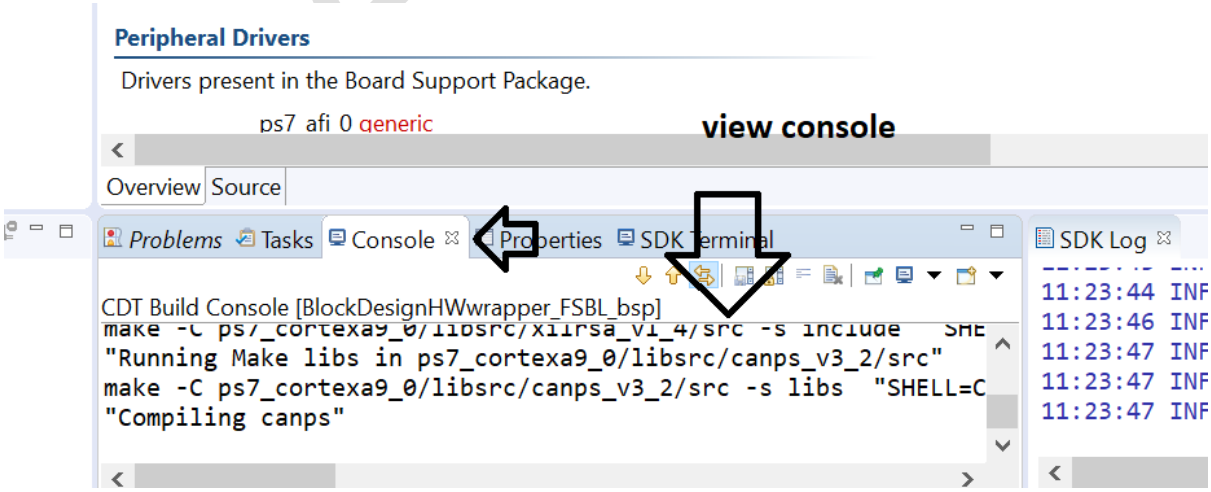


Figure 55: Viewing the Console in SDK

To create the FSBL project, SDK takes a while dependent on the type of processor one has, it may take several minutes.

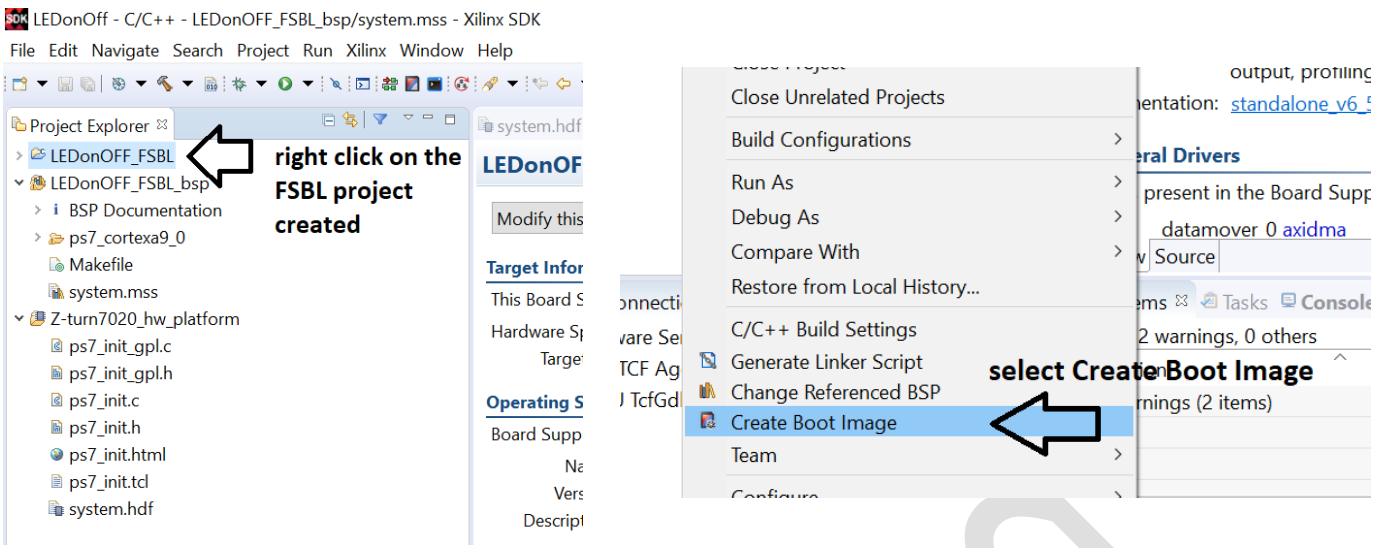


Figure 56: FSBL project created

Figure 56 shows a newly created FSBL project, that is part of the main Vivado project. The next step is to create a boot image file. This will be copied to the SD card, then the Zynq 7 will look for it, as soon as it is powered up. To create a boot image file, one must ***right-click*** on the newly created FSBL project and from the list, choose ***create boot image file***. This is shown in figure 56.

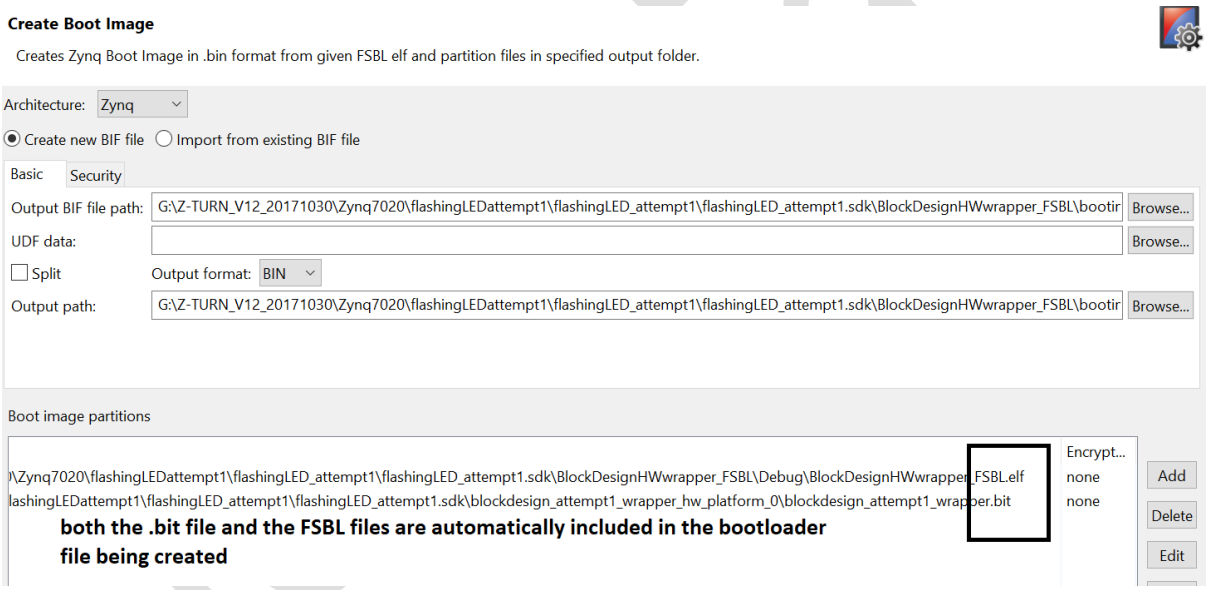
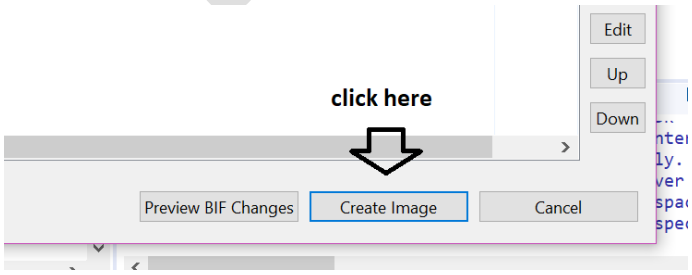


Figure 57: Create Boot Image Window



A new window pops up. In the lower part of this window, one can find the ***FSBL.elf*** file and the ***.bit*** file created earlier in Vivado. So, the Zynq 7 first reads the bootloader file and then reads the ***.bit*** file to create the hardware in the programmable logic part. Click on Create Image button and SDK will generate an image file that has to be copied to the SD card for booting the Zynq 7.

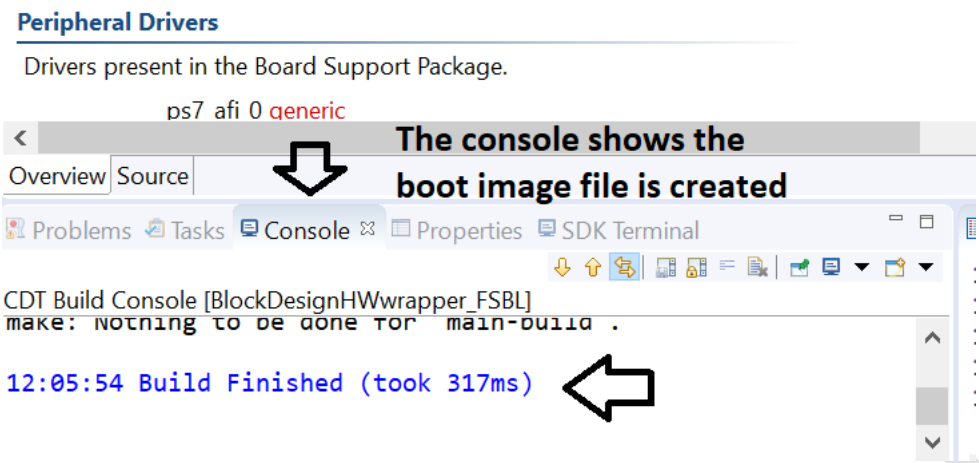


Figure 58: Console shows the image file ready

Copying the image file to SD card

After the image file is created, the next step is to copy it on SD card. This is done just using the normal copy/paste combination of commands in Windows. However, one has to find the right image file and the following snap shots shows the steps.

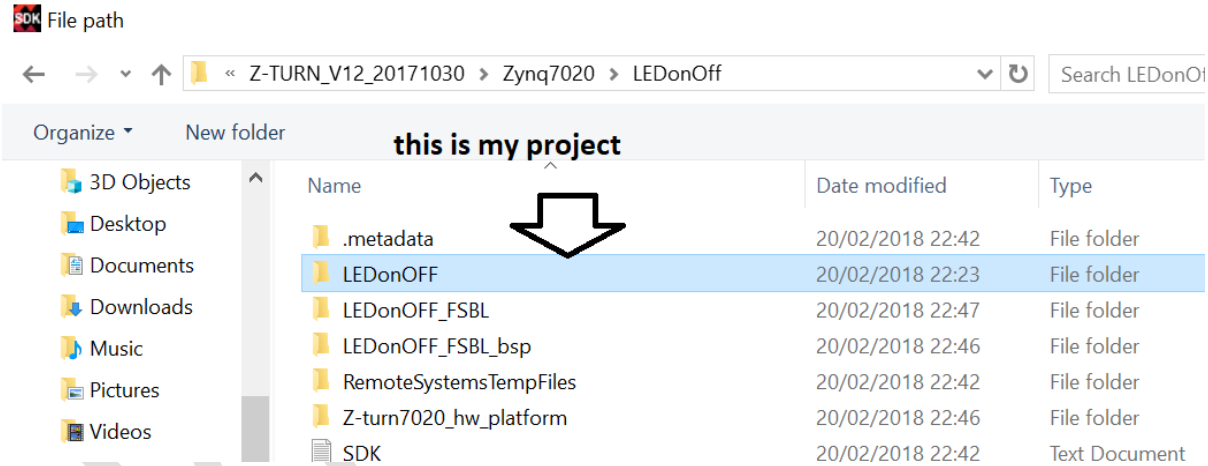


Figure 59: The Project's location

Figure 59 shows the project location in the hard disk.

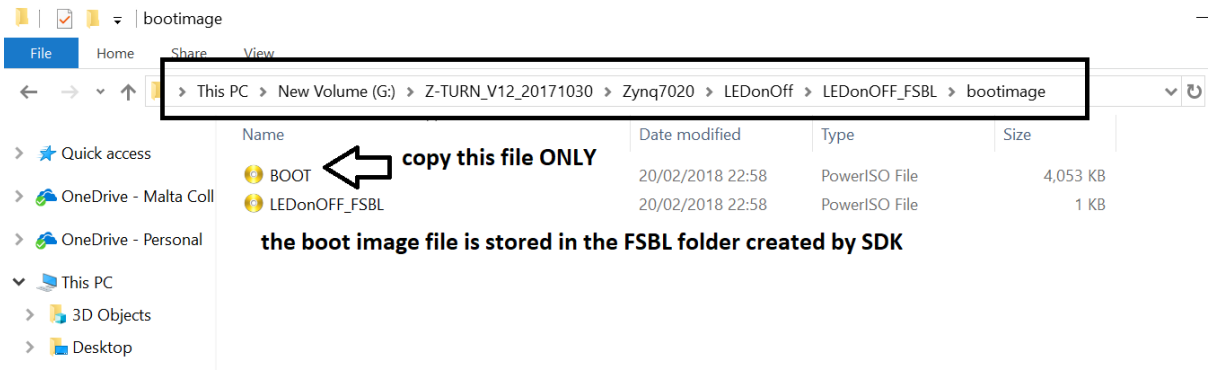


Figure 60: Location of the image file

Figure 60 shows the location of the boot image file that must be copied to SD card using the copy/paste commands in Windows. Figure 60 also shows that only the ***BOOT*** file has to be copied only!

Insert an SD card in an SD card to USB adapter. Insert the adapter in the USB port in the laptop or PC. Windows should detect the SD card and opens a new window showing the contents of the SD card. Use the copy/paste commands in Windows to copy the BOOT image file.

After copying the file, make sure to disable or disengage the SD card from Windows not to corrupt it. Insert it in the slot of the Z-turn board, power up the board and the LEDs should light up. Please note that as the board schematics dictate, the LEDs need a logic 0 on the pins for them to light up. Enjoy!

Steps to create an A9 Soft core project using both Vivado and SDK

Introduction

In this chapter, the ARM Cortex A9 will be used to control the flashing of LEDs. This time, instead of writing VHDL code to create hardware, ***C*** instructions will be written in a typical ***C*** project. So, in this project, the author will show how to create a ***C*** project from SDK and which functions should be used to light the LEDs located at MIO 0 and MIO 9. Once again, the whole process how to create a project will be shown so that one will become more familiar with the steps needed to create a project in Vivado.

Starting a new project in Vivado

Figure 2. 1 : Starting a Project in Vivado

Figure 2.1 speaks for itself, click on ***Create Project***

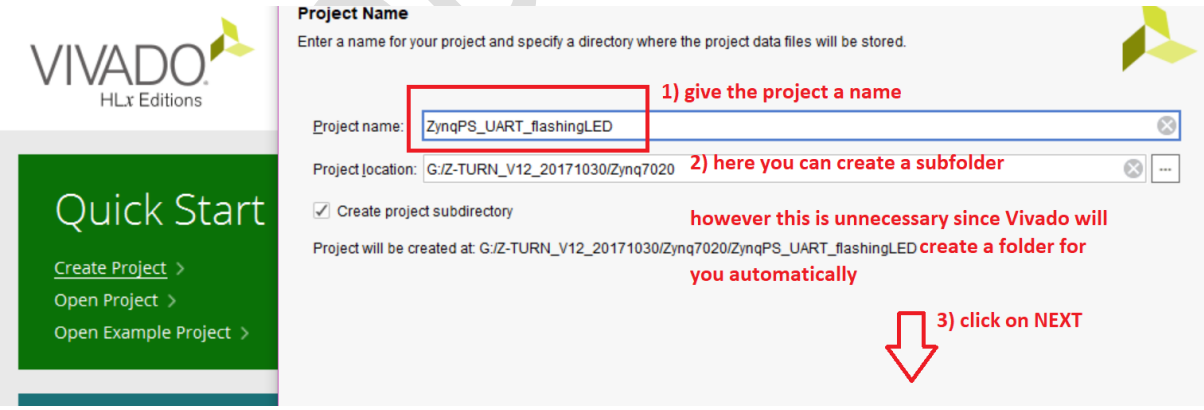


Figure 2. 2: Name the Project

Give a name to the project and make sure that it will be stored in the desired folder. Click on ***NEXT*** below.

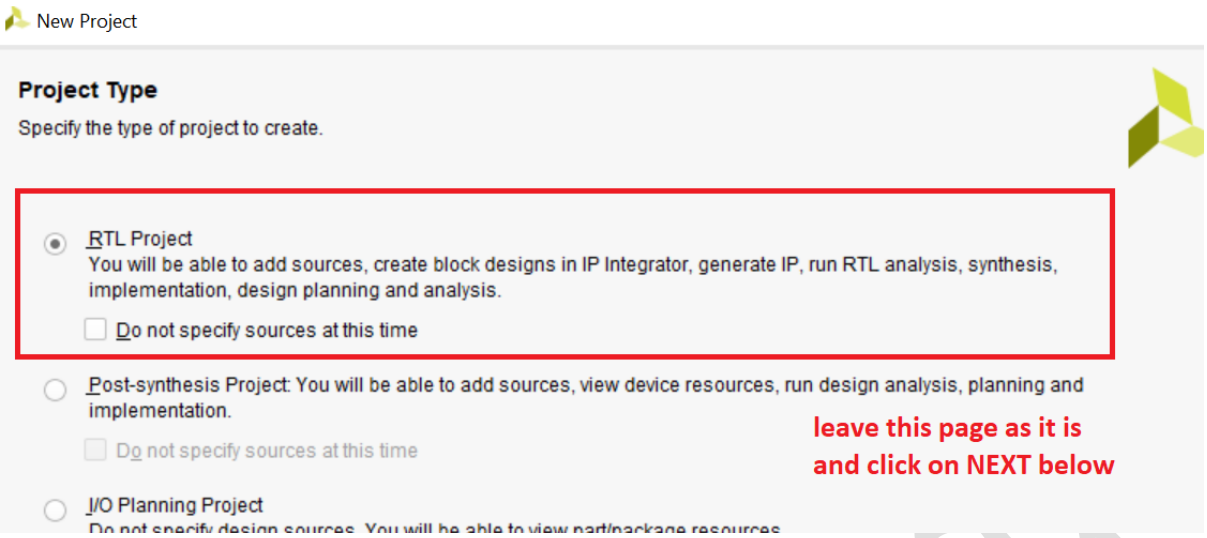


Figure 2. 3: Type of Project Page

Figure 2.3 shows the type of project one can use, since this project will be based on the hard core A9 processor or the ***Processing System*** part of the Zynq 7, this page will not affect the project, so it will be left untouched. Click ***NEXT***.

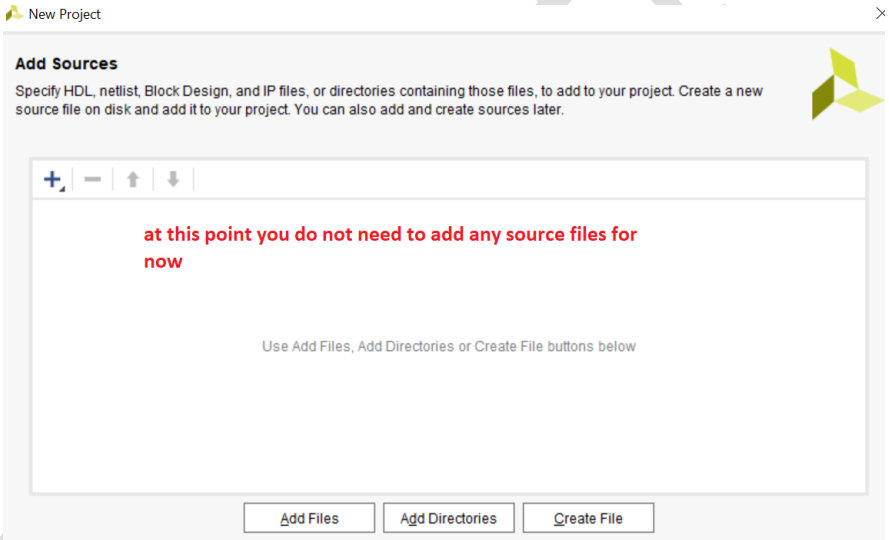


Figure 2. 4: Adding Source Files

Figure 2.4 asks to add source files. Again, since this project will focus on the Processing System part of the Zynq 7, no source files need to be created at this point. Click on ***NEXT***.

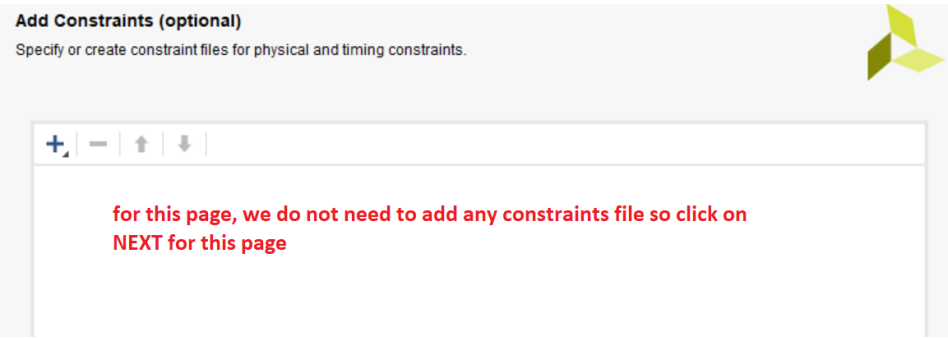


Figure 2. 5: Adding Constraints File

The next page asks for the ***constraints file*** or how the pins will be connected. This is ***unnecessary*** because since the ***Board Support Files*** have been included in Vivado folder as described in chapter 1. So, click on ***NEXT***.

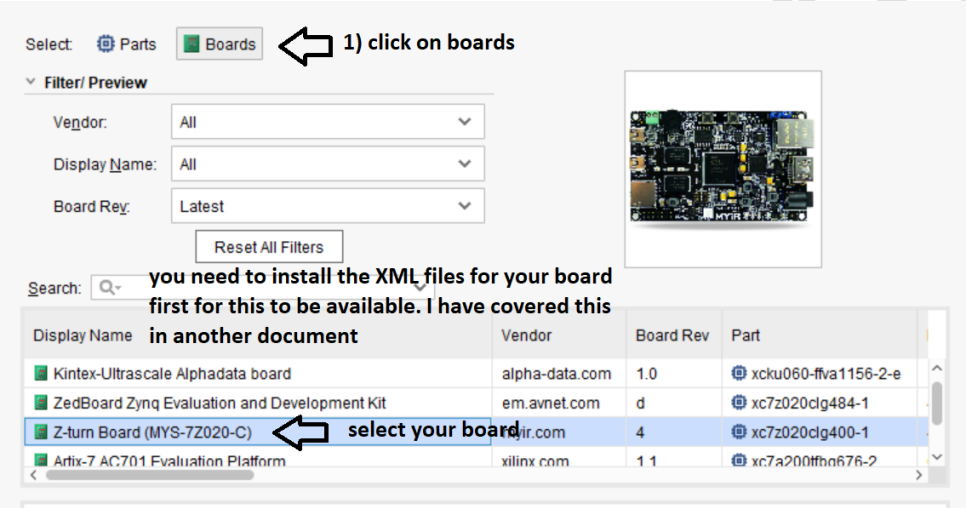


Figure 2. 6: Selecting the Z-turn Board

Figure 6 shows how to select the Z-turn board from the list. This will make sure that Vivado and SDK will configure the environments to comply with the Z-turn board peripherals and characteristics.

Just to refresh one's memory, figure 2.7 and 2.8 below, show the XML files downloaded from Github and where these files should be stored within Vivado directory.

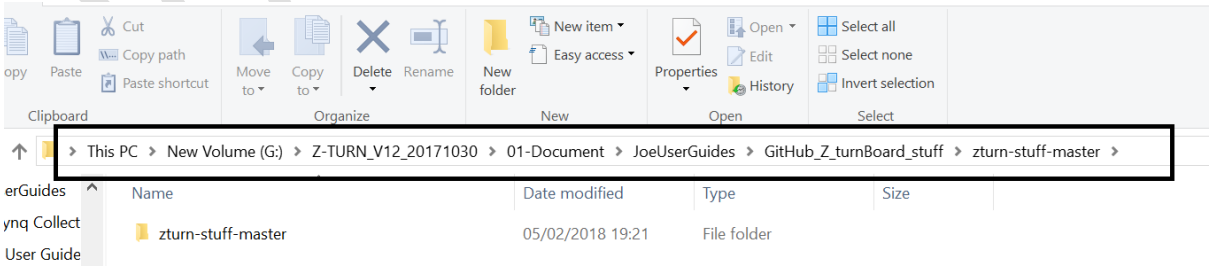


Figure 2. 7: Github folder

Then after extracting the files, copy them to:

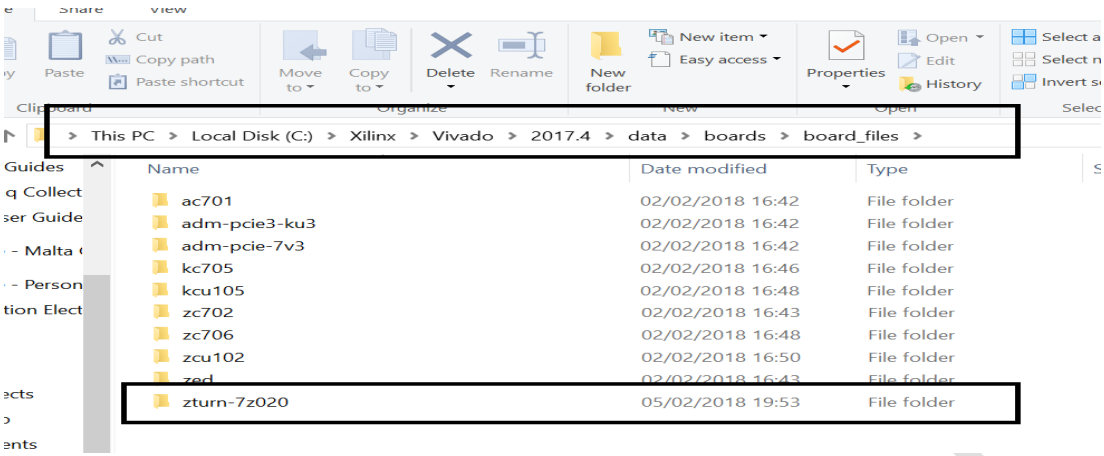


Figure 2. 8: Z-turn folder within Vivado

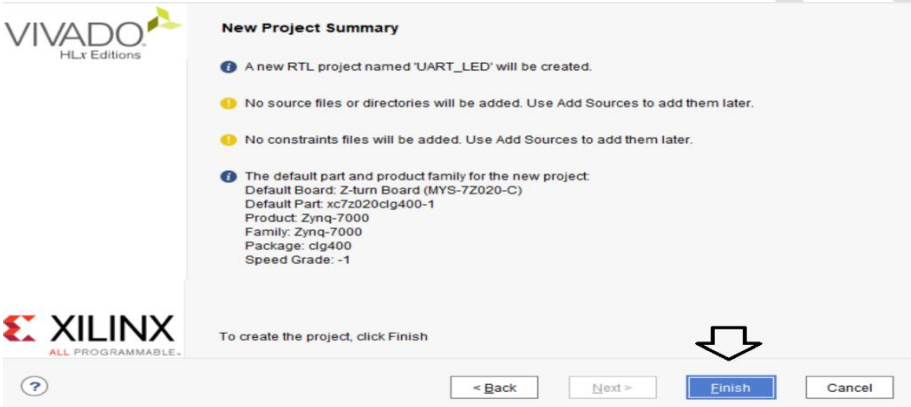


Figure 2. 9: Project Summary Page

Figure 2.9 shows the project summary page. Click on ***FINISH*** to open the project in Vivado.

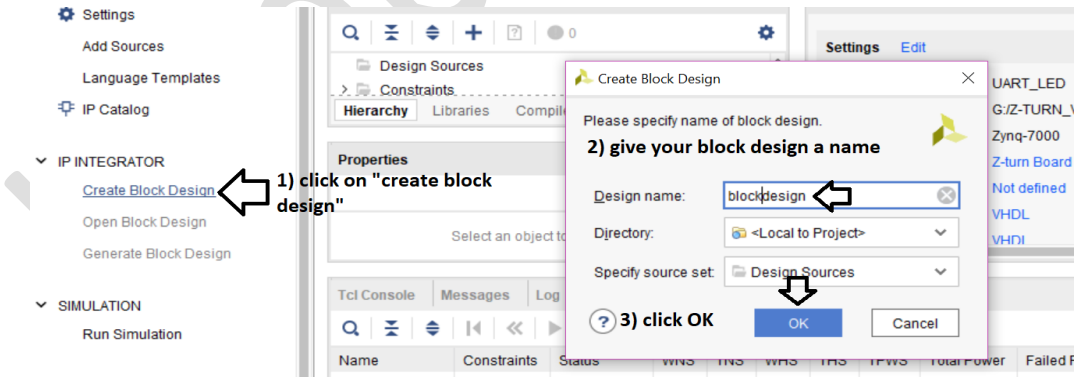


Figure 2. 10: Creating a new Block Design

Once the project environment is opened in Vivado, one can immediately create a new Block Design. This will open a new schematic window where one can connect all the components in the design. Follow the steps in Figure 2.10 above to create a new schematic.

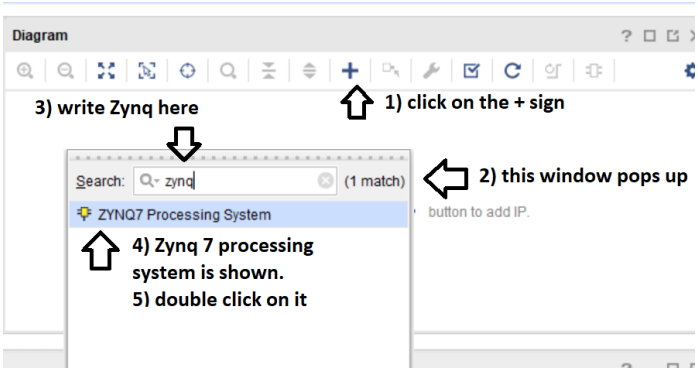


Figure 2. 11: Schematics Window

Figure 2.11 shows an opened schematics window. It also shows the steps to create a new ***A9 hardcore processor*** always referred to as the ***Processing System***.

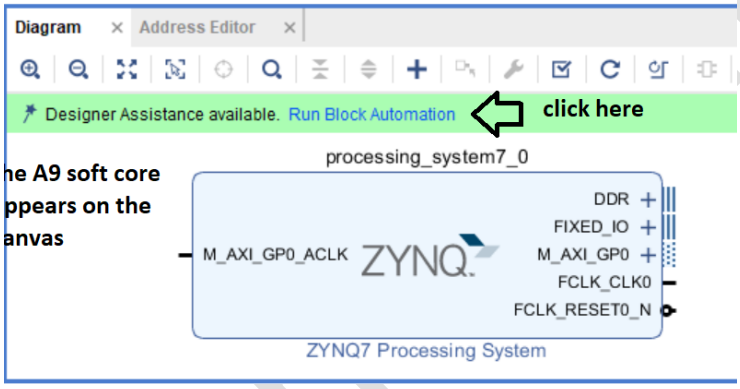


Figure 2. 12: Block Automation

After the ***Processing System*** part is created, one must click on ***Run Block Automation*** so that all the hardware peripherals of the A9 core will be enabled. This is shown in Figure 2.13. Make sure that ***Apply Board Preset*** will be ticked!

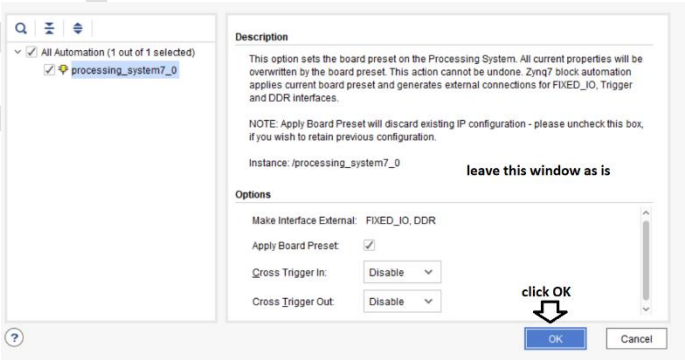


Figure 2. 13: Apply Board Presets

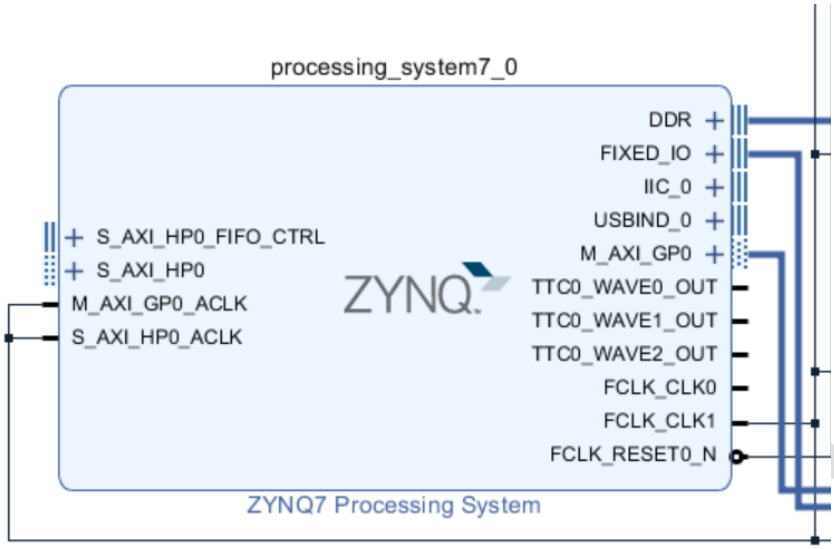


Figure 2. 14: Connecting the AXI clocks

Make sure to connect the AXI clocks as shown in Figure 2.14 to avoid errors.

Now it is time to create a ***Hardware Wrapper***. This serves as a ***top-level*** block to the sub-systems in the system.

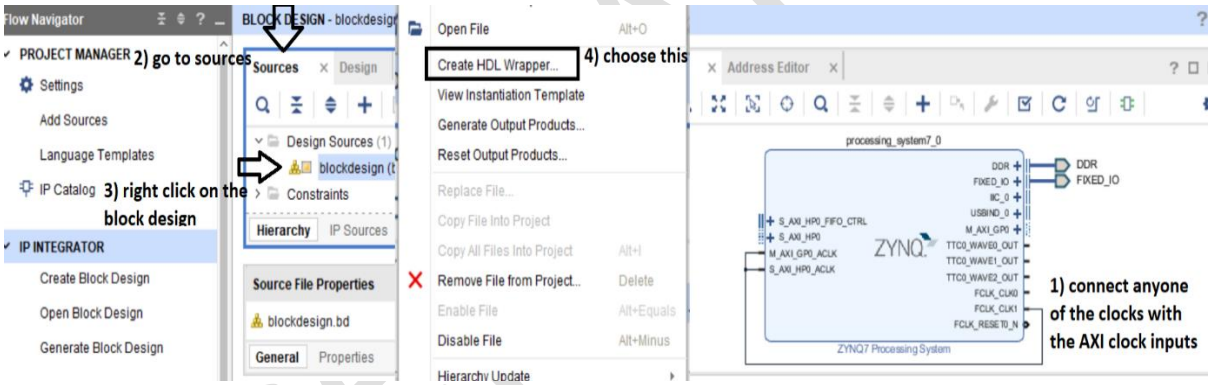


Figure 2. 15: Creating a Hardware Wrapper

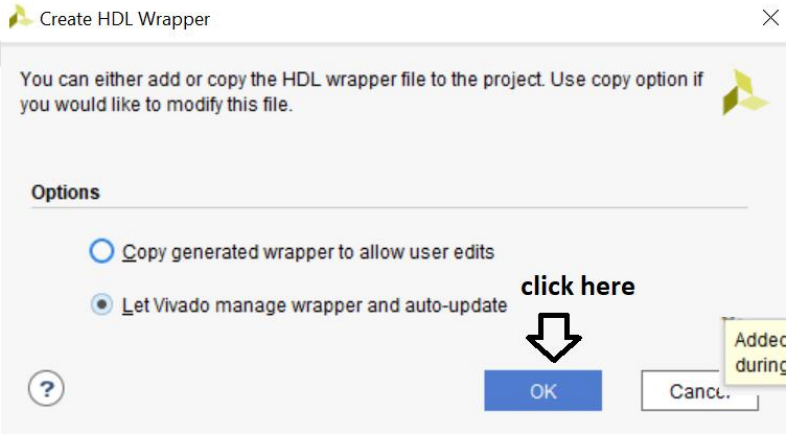


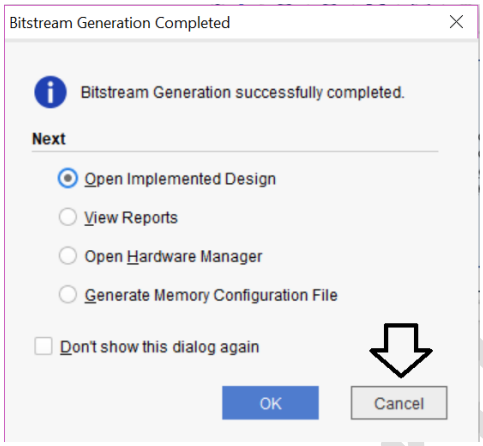
Figure 2. 16: Let Vivado create all the settings

Referring to Figure 2.13 - after double clicking the block diagram of the A9 core, one must be aware of how Vivado shows which peripherals can be used in software. Apart from others, there are two UARTs. ***At least UART 1 must be enabled, so that one will be able to use the basic C project template called "Hello World" within SDK. Using this C project template, one will solve all the linker problems between the software part and the hardware part. It is recommended to use this template and not the blank C project template! If the UART peripherals are removed from the Zynq 7 Processing System, then this C project could not be used in SDK***.

If peripherals that are not used in the processing system part are disabled, the firmware might not work, so it is recommended that when using the Board Support Files, one will NOT disable any of the peripherals already enabled in the preset version of the Zynq Processing System!

Since in this project, the focus is on how to enable the Processing System of the Zynq 7 and also how to use the built-in functions in SDK to enable certain peripherals, at this point, no VHDL module will be created, so after the HDL wrapper is created, one can generate the bit-stream file straight away!

After creating the HDL wrapper, one can generate the bit stream.



At this point, one must note that no constraints files have been created or used. This is because this is taken care of by Vivado. The pins of the SoC are allocated according to the *board support files.* So one can do without the constraints file when working with the IP core.

Figure 2. 17: Bitstream File generated

Figure 2.18 shows the next step after the bitstream file has been created. The hardware file must be exported and this is done from the ***File*** menu. ***File → Export → Export Hardware***

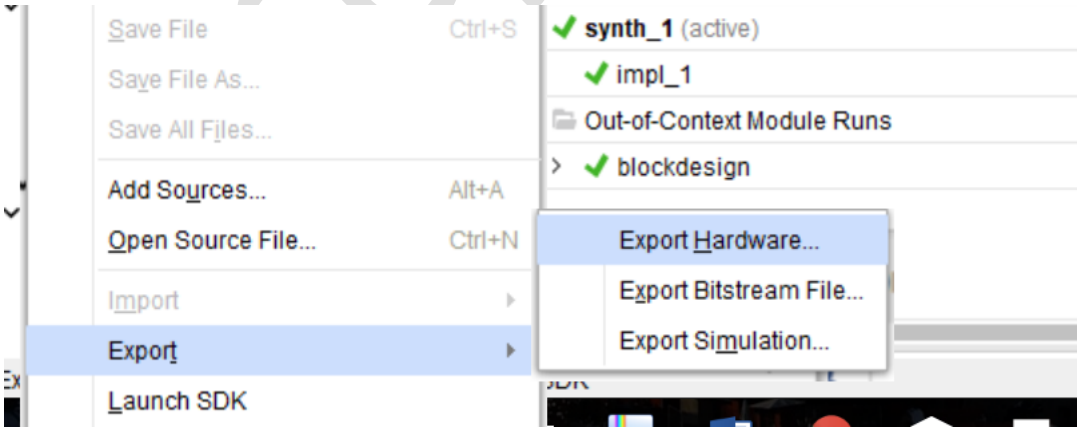


Figure 2. 18: Exporting the Bitstream File

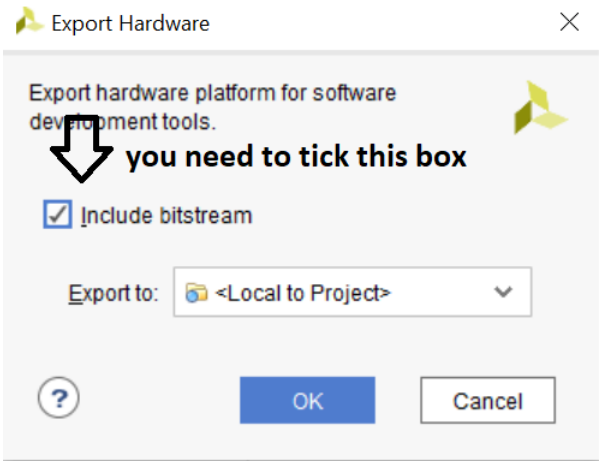
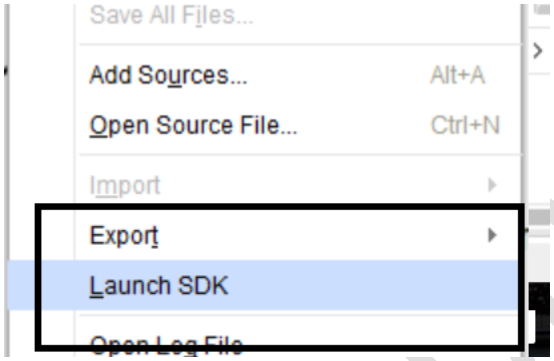


Figure 2. 19: Include the Bitstream File

Make sure to tick the square box as shown in Figure2.19.

Then one can open SDK from Vivado. This will create a subfolder within the hardware-project-folder where all the SDK files will be saved.



Click on File → Launch SDK

Save the SDK project ***<Local to the project>***

Figure 2. 20: Launching SDK fromo within Vivado

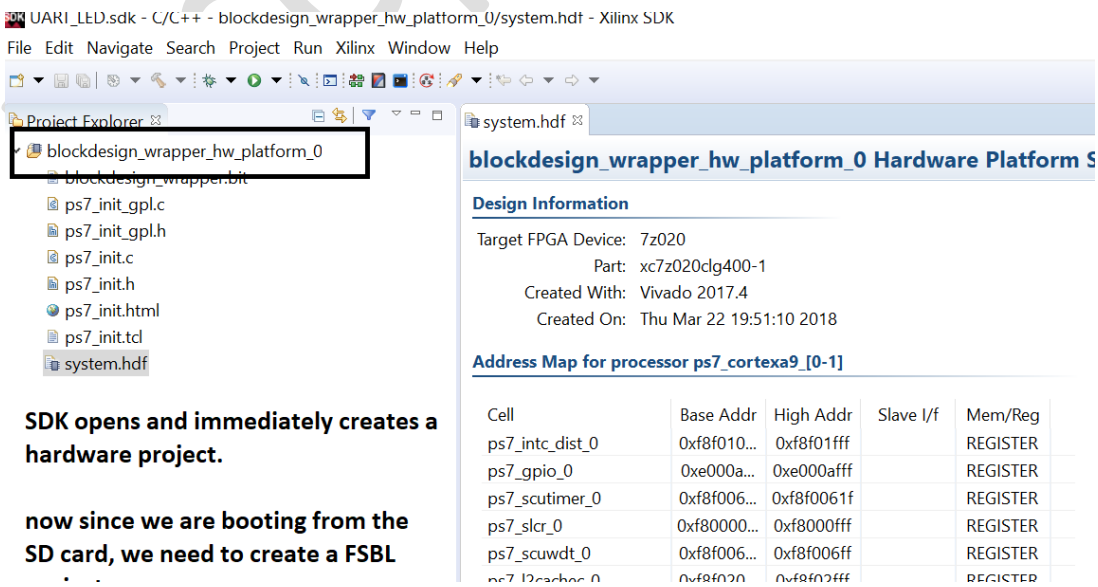


Figure 2. 21: Opening SDK

Figure 2.21 shows the SDK linked to the Vivado project. The next step is to create a ***First Stage Boot Loader (FSBL)*** project that will link all the C and VHDL project together. Figure 2.22 shows the steps to create an FSBL project which will be used by the Zynq 7 Processing System to load both the hardware and software of the system.

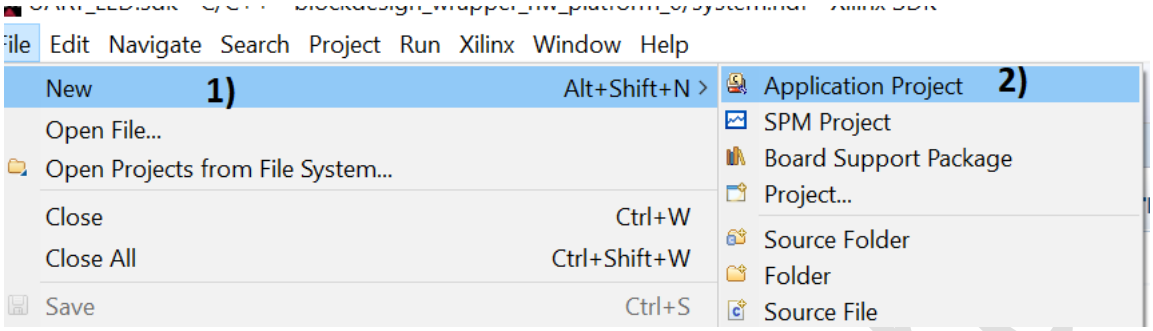


Figure 2. 22: Creating an FSBL project

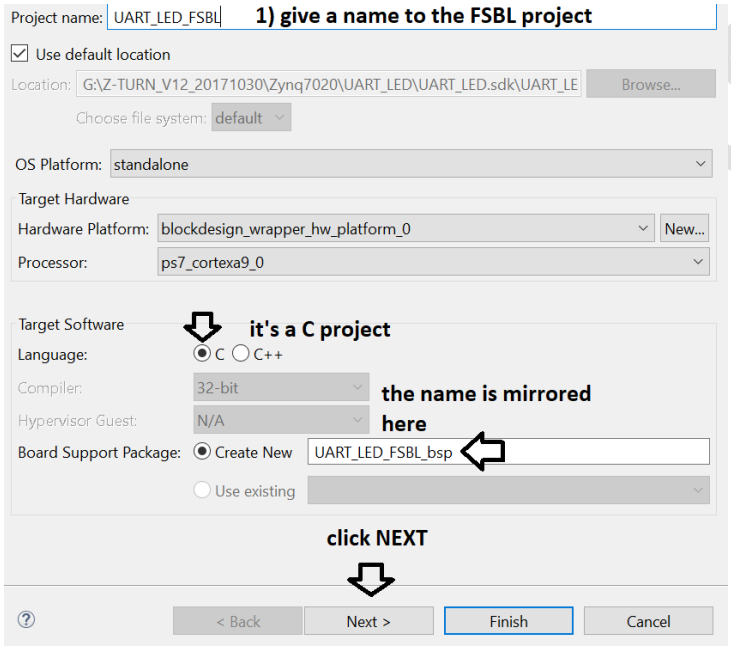


Figure 2. 23: Naming the FSBL project

Figure 2.23 show what one must fill, to create a new FSBL project.

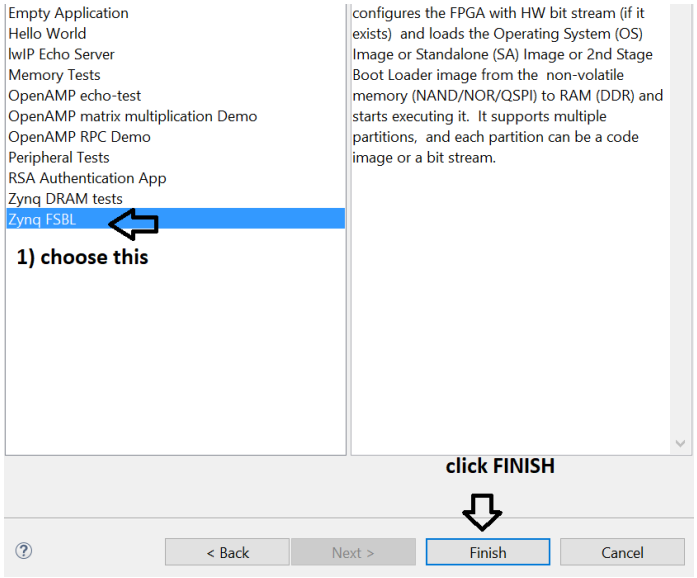


Figure 2. 24: Choosing FSBL form the List

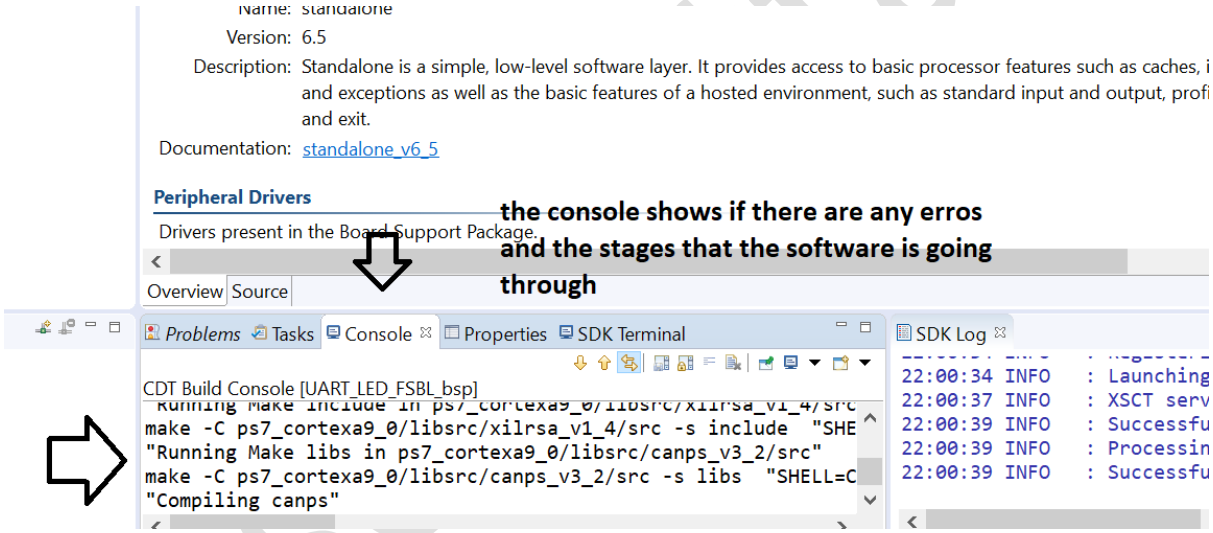


Figure 2. 25: SDK console

Figure 2.25 show the SDK console running. This is a useful part of SDK and therefore one must make sure to listen to it!

Now after the FSBL project has been successfully created, one must create a C project. This project consists of many C source files and header files, together with the functions needed by the application in quation!

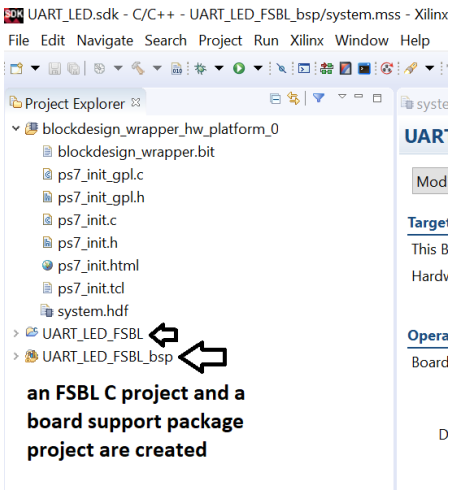


Figure 2. 26: FSBL project created

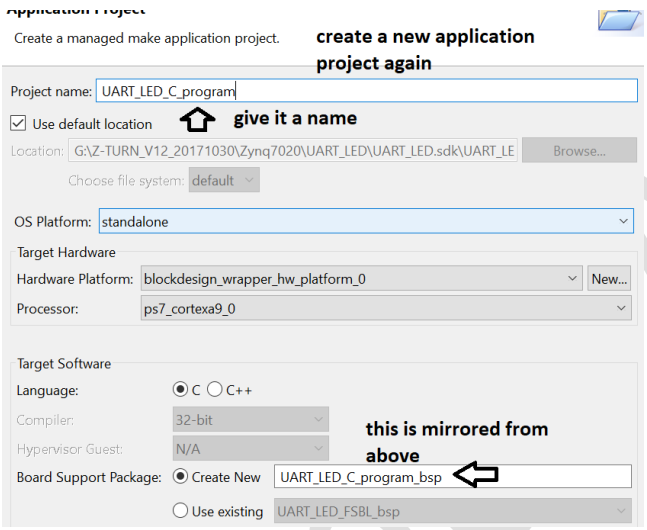


Figure 2. 27: Creating a C project

To create a new C application, one must file click on FILE -> new -> new application -> the window of Figure 2.27 pops up. Give the C project a name and leave the rest is it is.

As already stated previously, it is advisable to use this template as the base of the C project because Vivado will not generate any unnecessary errors due to incorrectly created C project.

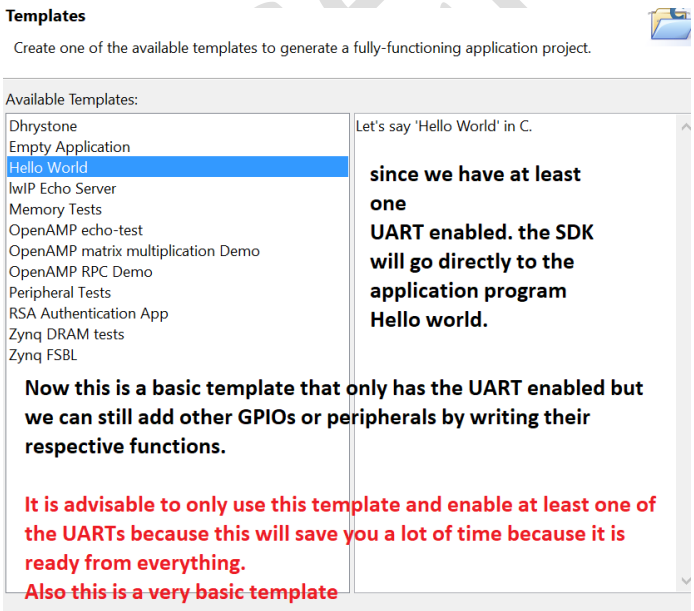


Figure 2. 28: Creating a C project

Only the **Hello World** template is good to build a C application. ***It is advisable to make sure that one of the UARTs is enabled as otherwise this application <u>will not be available to the user</u>.***

The above happened to the author when he tried to open the ***hello world*** template when in Vivado the author disabled all the peripherals and only the ***blank-application-project*** was available!!

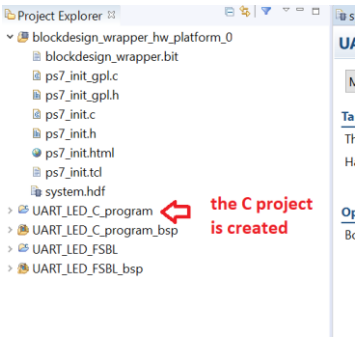


Figure 2. 29: The C project residing on top of the other two projects

Now one must make sure that ***UART 1 is enabled*** because the USB to UART chip is only connected to UART 1 which resides at MIO-48 and MIO-49.

This is done by changing the settings of the board support package of <u>the project</u>!!

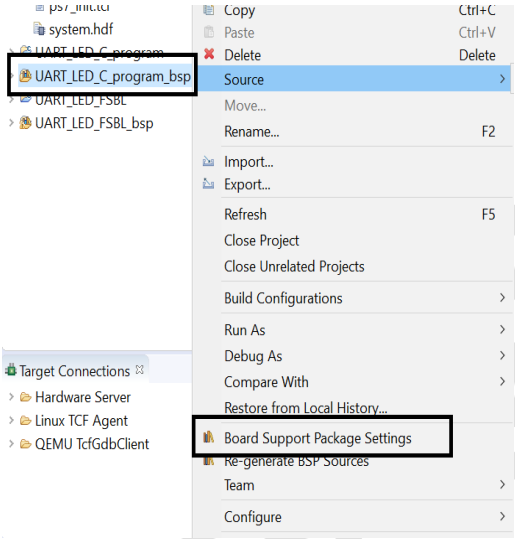


Figure 2. 30: Modifying the Board Support Package

1) Right-click on the name of the C project.
2) Click on the ***Board support package*** of the **C project**.
3) Then click on ***Board Support Package Settings***

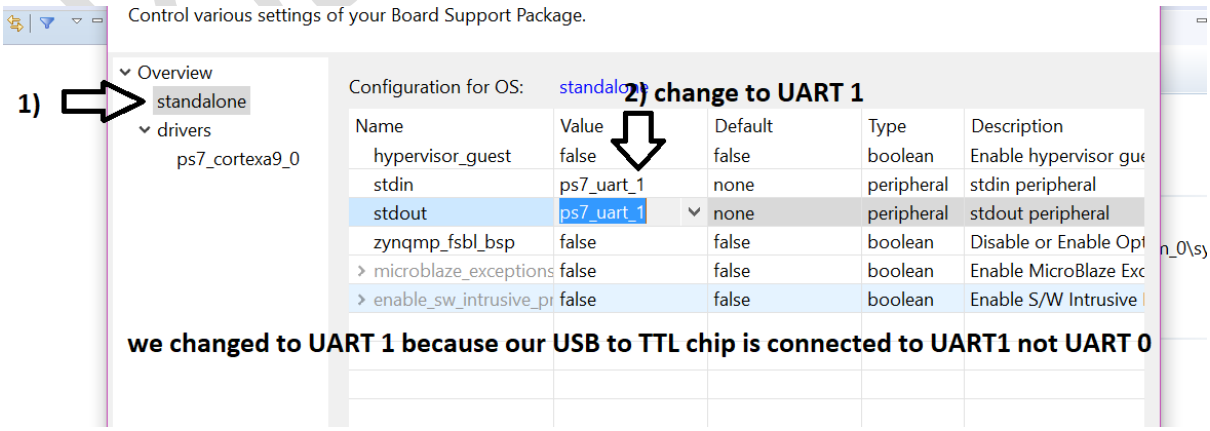


Figure 2. 31: Changing the Board Support Packages to change UART 1 settings

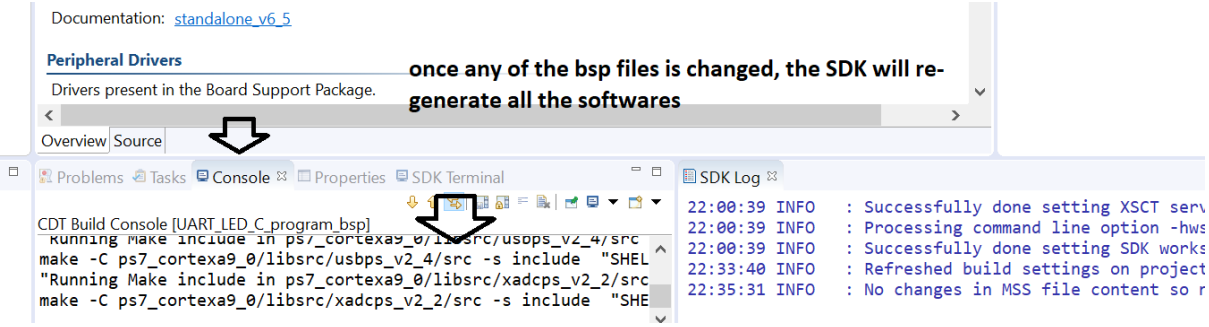


Figure 2. 32: Console in SDK shows all the transformation being done

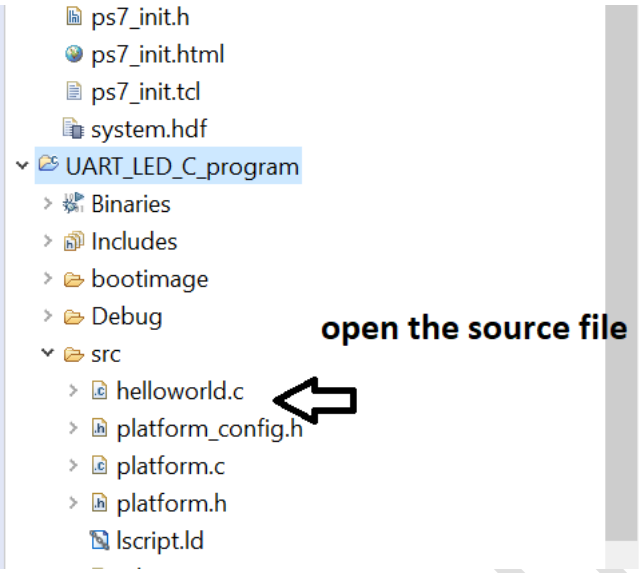


Figure 2.33 show the location of the ***Hello World***, C program. Now it is time to open it and write some code to flash an LED and send data on the serial port.

Vivado also has an extra feature that once you save the project it will start re-building on its own.

Figure 2. 33: Location of the C project in SDK

The Software Part in SDK

So far, the Vivado project was created and linked to the SDK from where C instruction could be written to control in this application the two LEDs connected with pins MIO0 and MIO9 of the Processing System part of the Zynq 7. Double-clicking on helloworld.c file will open it on SDK to be edited as follows:

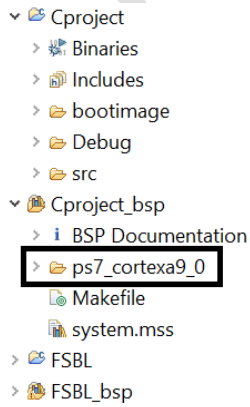


Figure 2. 34: Location of all C functions

Figure 2.34 show the folder where all the C functions available to the user reside. Click on the arrow pointing to this folder to reveal more folders.

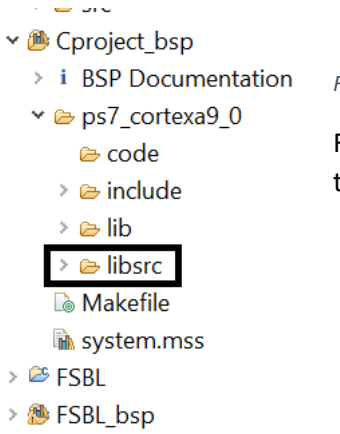


Figure 2. 35: C Functions Library

Figure 2.35 shows one level deeper in the folder to reveal more folders that contain the functions of all peripherals enabled in the system.

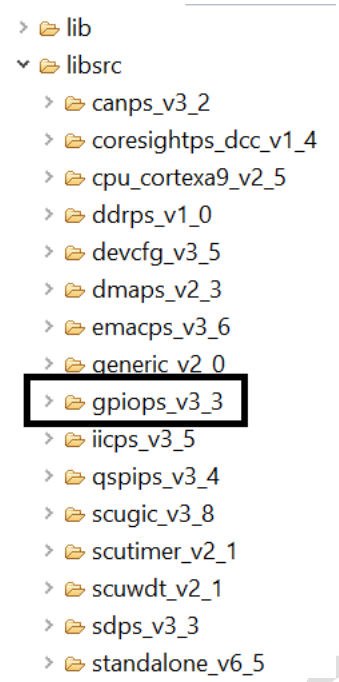


Figure 2. 36: Folders containing functions

Figure 2.36 show various folders that contain the respective functions already written and ready to be used by the user. In this particular application where only two LEDs will be lit, and these LEDs are connected to MIO0 and MIO9, the gpiops_v3_3 folder will be used because this folder contains all the functions needed to configure the GPIO bank located on the PS part of the Zynq 7.

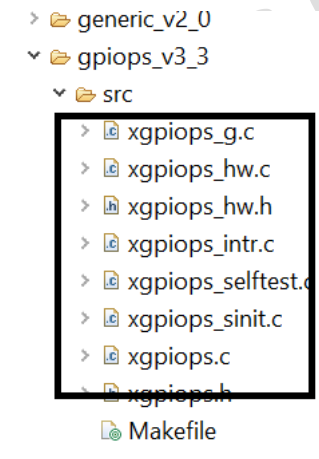


Figure 2. 37: GPIO_PS functions

The adjacent files shown in Figure 2.37 contain all the functions needed to control the GPIOs that are located on the PS part of the Zynq 7. Remember that these GPIOs have nothing to do with the GPIOs connected to the PL part and therefore they have different configurations and also different names! Make sure to select the right folder from the list!

Configuring the Processing System (PS) GPIOs

The following routine should be used to configure nearly all the peripherals from the Processing System side of the Zynq 7. One must include the following include directive for the functions to be eligible:

#include "xgpiops.h"

First look for the lookupConfig():

```
XGpioPs_Config *XGpioPs_LookupConfig(u16 DeviceId)
```

The above function is located in ***xgpiops_sinit.c*** file.

The above function ***returns*** a pointer of type ***XGpioPS_Config*** while is ***passed a parameter*** of type ***u16***.

The DeviceID parameter can be replaced with ***XPAR_PS7_GPIO_0_DEVICE_ID.*** This is located in ***xgpiops_g.c*** file

Now **cut** the XGpioPs_Config from the name of the function above and **paste** it as one of the initial data types at the beginning of the main (). Assign a name to this pointer as shown below:

XGpioPs_Config *ConfigPtr;

Then equate the above function to the name given to the pointer. The final result is shown below:

ConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);

Now its time to initialize the GPIO_PS peripheral. Use the following function:

s32 XGpioPs_CfgInitialize(XGpioPs *InstancePtr, XGpioPs_Config *ConfigPtr,u32 EffectiveAddr)

which is located in ***xgpiops.c*** file. The above function returns a type of ***signed 32 (s32)*** and is passed various parameters.

So **cut** the **s32** and **paste** it as part of the data type list at the top of the main function. Assign a name to a variable of type **s32**.

s32 Status;

Cut and paste as one of the data types ***XGpioPs*** and give it a name as well at the beginning of the main ().

XGpioPS Gpio;

For ****ConfigPtr*** use ***ConfigPtr*** as before so the above function will now look like this:

Status = XGpioPs_CfgInitialize(&Gpio, ConfigPtr, ConfigPtr->BaseAddr);

The ***BaseAddr*** was taken from ***XGpioPs_hw.h*** file.

Now if one is to equate a function's return variable to a variable just like what has been done with ***XGpioPs_CfgInitialize()***, it is good practice to ***check its validity*** by writing the following ***if-statement***. If this if-statement is not included in the code, a warning is generated in the output file.

```
if (Status != XST_SUCCESS)
        {
    return XST_FAILURE;
        }
```

<u>Setting the Direction of the Pin and Enabling the Output</u>

Now for each pin in the GPIO, one has to ***set*** its direction -whether the pin is going to act as an **input** or an **output**. Also, if the pin is going to act as an output, one has to **enable** that output! The following functions are used:

void XGpioPs_SetDirectionPin(XGpioPs *InstancePtr, u32 Pin, u32 Direction)

void XGpioPs_SetOutputEnablePin(XGpioPs *InstancePtr, u32 Pin, u32 OpEnable)

The above two functions are located in ***XGpioPs.c*** file. <u>Before each function, there is a description of what the function should do</u> and sometimes there are also hints on the parameters. So, make sure to read the comments before every function to have a better understanding of its effects and also what type of parameters should be passed to it!

The first function above:

void XGpioPs_SetDirectionPin(XGpioPs *InstancePtr, u32 Pin, u32 Direction)

returns a **void** and therefore expect nothing from it,

the first parameter that should be passed is the name of the **instance** – in this case it is **&Gpio**

u32 Pin is the pin number the function will be affecting, since in this particular example, two LEDs connected to MIO0 and MIO9 are going to be used, then this function should be written for two pins – pin 0 and pin 9.

u32 Direction: for this parameter, the function accepts either 0 if the pin is going to act as an input, or 1, if the pin is going to act as an output

now for the second function:

void XGpioPs_SetOutputEnablePin(XGpioPs *InstancePtr, u32 Pin, u32 OpEnable)

again, it returns a void so nothing should be expected from it

XGpioPs *InstancePtr should be replaced once again with ***&Gpio***

u32 Pin should be replaced with the ***pin number*** – for this example Pin should be replaced with either **0 or 9**, while ***u32 OpEnable*** should be replaced with **1** if the outputs are **enabled** and **0** if the outputs are **disabled**!

Writing to the individual Pin

The last function that needs explanation is the following:

void XGpioPs_WritePin(XGpioPs *InstancePtr, u32 Pin, u32 Data)

The above function writes to the individual pins.

Once again it returns a void and therefore expect nothing once the function is ready.

Replace **XGpioPs *InstancePtr** with **&Gpio**

Replace **u32 Pin** with the pin number - in this case it has to be **0** or **9**

Replace **u32 Data** with either **1** for logic high or **0** for logic 0.

The LEDs connected to MIO 0 and MIO 9 on the Z-turn board are connected such that a logic 0 will switch them on while a logic 1 will switch them off!

So the main() will look like this:

```
int main()
{
       int Status;
              XGpioPs_Config *ConfigPtr;
              XGpioPs Gpio;/* The driver instance for GPIO Device. */

    init_platform();

    /* * Initialize the GPIO driver. */
              ConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
              Status = XGpioPs_CfgInitialize(&Gpio, ConfigPtr,
                                        ConfigPtr->BaseAddr);
              if (Status != XST_SUCCESS)
                 {

                    return XST_FAILURE;
                 }

              // LED1 gpio setting
              XGpioPs_SetDirectionPin(&Gpio, 0, 1);
              XGpioPs_SetDirectionPin(&Gpio, 9, 1);
              XGpioPs_SetOutputEnablePin(&Gpio, 0, 1);
              XGpioPs_SetOutputEnablePin(&Gpio, 9, 1);

          while (1) {
              XGpioPs_WritePin(&Gpio, 0, 0);
              XGpioPs_WritePin(&Gpio, 9, 0);
              delay();
              XGpioPs_WritePin(&Gpio, 0, 1);
              XGpioPs_WritePin(&Gpio, 9, 1);
              delay();
                   }
       cleanup_platform();
       return 0;
}
```

Now since the A9 core will execute the above code very rapidly, a delay function was introduced so that one can see the LEDs blinking. A typical delay function can be written as shown below:

```
void delay()
{
       int i;
       for (i = 0; i < 10000000; i++)
             {
               //do nothing
              }

}
```

Another way to flash the LEDs on MIO 0 and MIO 9 simultaneously is by using the following functions instead of the ones used in the main code previously.

void XGpioPs_Write(XGpioPs *InstancePtr, u8 Bank, u32 Data)

the only parameter that is different this time is the **u8 Bank** which has to be replaced with **0** as a number, **u32 Data** can be replaced with either a *decimal number* which is *not recommended* for this instance or better with a hexadecimal number. Thus, to write a logic 1 in both MIO 9 and MIO 0 simultaneously using the above function, one must convert it to:

void XGpioPs_Write(&Gpio, 0, 0x00000201);

The same procedure could be written for the direction function

void XGpioPs_SetDirection(XGpioPs *InstancePtr, u8 Bank, u32 Direction)

could be replaced with:

XGpioPs_SetDirection(Gpio, 0, 0x00000201);

And

void XGpioPs_SetOutputEnable(XGpioPs *InstancePtr, u8 Bank, u32 OpEnable)

could be replaced with:

XGpioPs_SetOutputEnable(&Gpio, 0, 0x00000201);

Again Gpio is according to how the XGpioPs *InstancePtr was equated at the beginning of the main()

XGpioPs Gpio;

Creating the Boot Image File

This time to create a boot-image file, one has to right-click on the C project and not on the FSBL project! This is shown in figure 2.38 below:

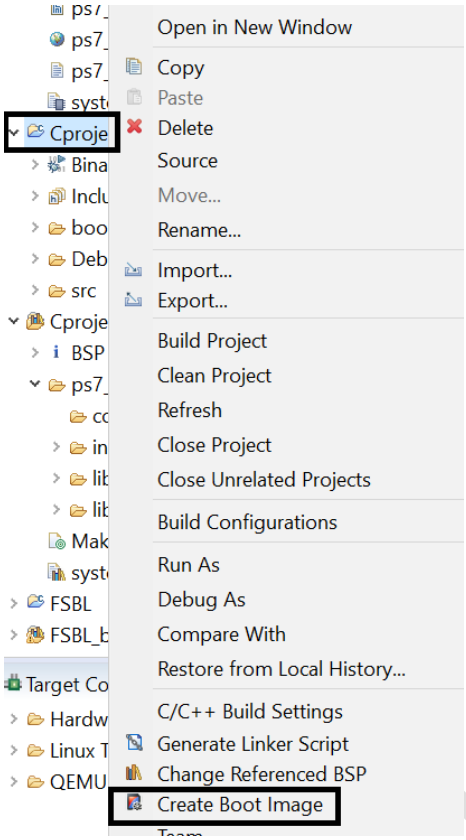


Figure 2. 38: Creating a Boot Image File

Figure 2.39 shows that this time, there are three files and not two in the boot image file. This shows that now the boot image file consists of the FSBL file, the bitstream file and also the software file.

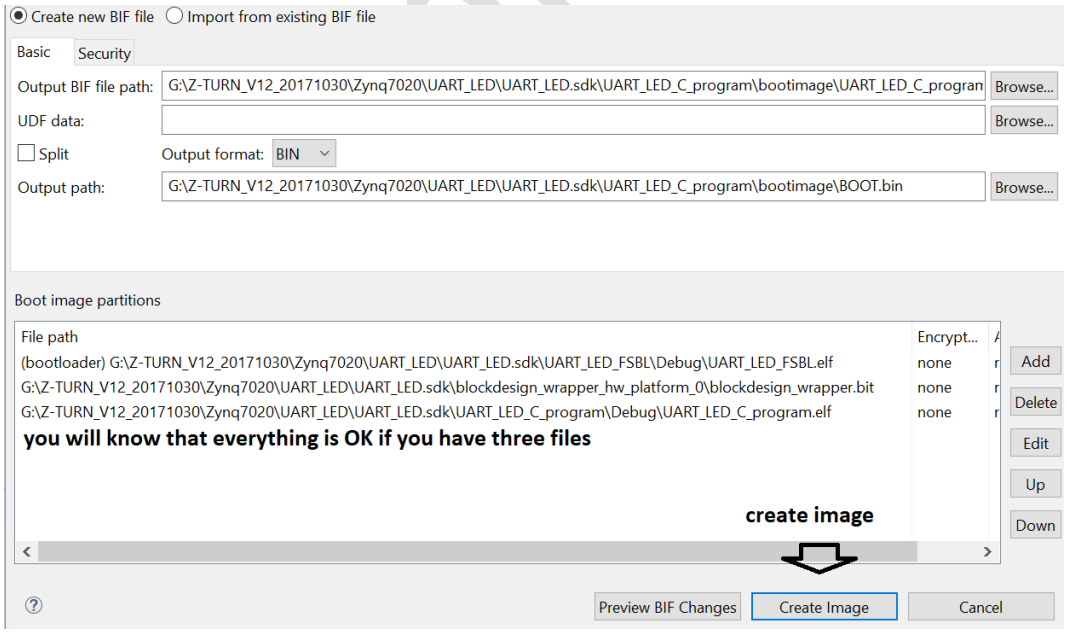


Figure 2. 39: Check files in Boot Image File

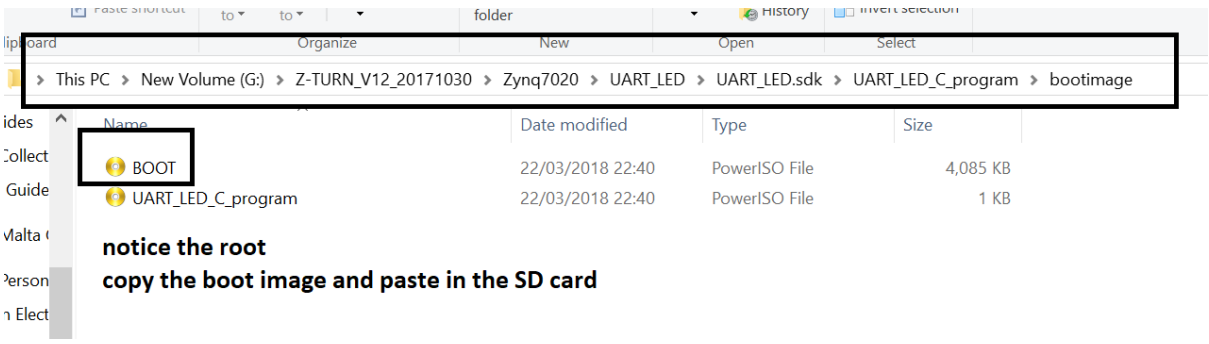


Figure 2. 40: Location of the Boot Image File

After the boot image file is created, it can be located in the C project and not the in the FSBL project, so make sure to select the right boot image file!

Copy the boot file to the SD card and insert it in the SD slot on the Z-turn board. Enjoy!

Flashing LEDs simultaneously but independently from the Processing System and Programmable Logic Fabric

Introduction

In this project, the versatility, flexibility and parallelism that could be achieved using the Zynq 7 System-on-Chip will be discussed. A VHDL entity that flash an LED will be created. After that, a software program will be written in ***C*** that also flashes an LED connected to the one of the MIO pins will be created. The VHDL entity will work *independently* from the program that will be flashed in the Zynq Processing System. So, in this project, there will be two independent hardware working in parallel!

The process how to create a project in Vivado and how to create a VHDL file have already been explained in chapters 1 and 2 so those will not be covered anymore. This chapter will go through the steps to achieve the goal of using both the Processing System and Programmable Logic independently for the first time.

Defining the Port terminals of the VHDL entity

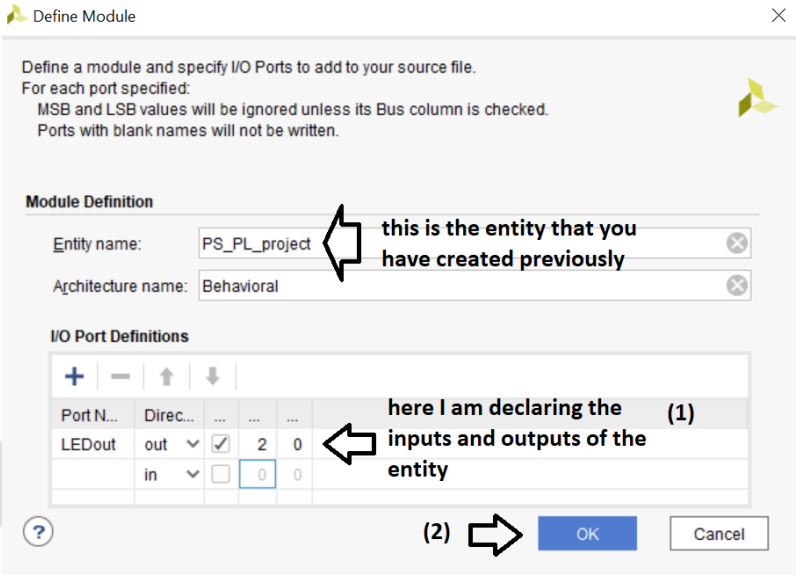


Figure 3. 1: Creating terminals for the VHDL entity

Figure 1 show the procedure to write the port terminals of the VHDL entity that has been created while setting up the project environment. Notice how a bus could be created!

Once Vivado has opened, look in the sources tab and double click on the name of the VHDL entity to open it up to start writing code. This is shown in Figure 2 on the next page.

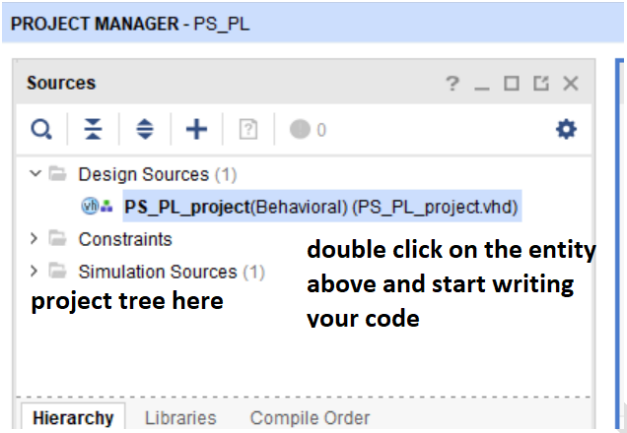


Figure 3. 2: Search for the VHDL entity

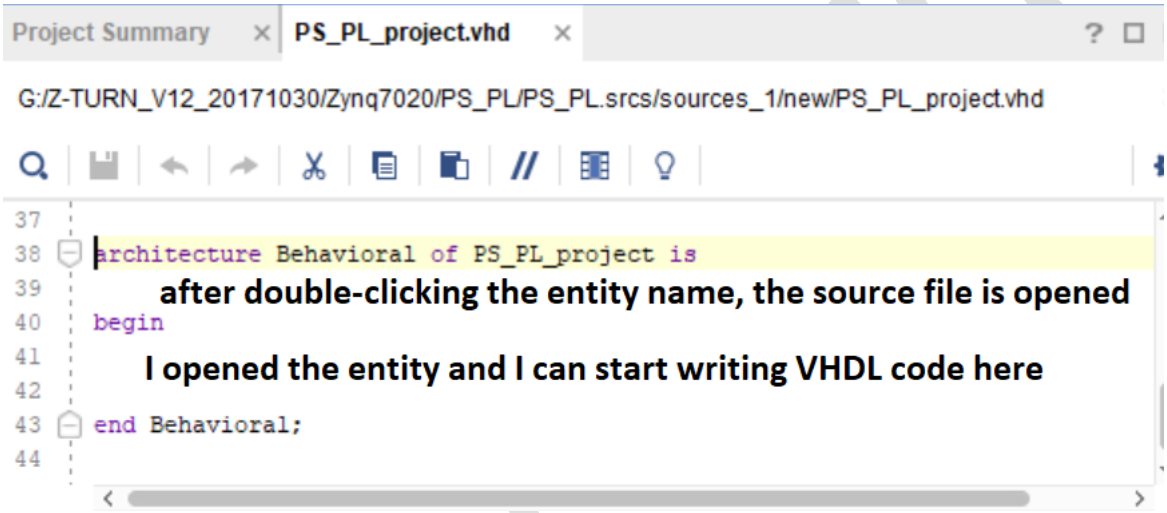


Figure 3. 3: Entity Architecture

The way of thinking when writing code in VHDL is a bit different when compared to writing code for a microprocessor. This is because even though VHDL contains sequential statements, one has to keep in mind that when writing code in VHDL for an FPGA, the code will be translated into hardware as opposed to writing code that will be flashed in memory of a microcontroller and then executed one instruction at a time. This will be explained in the following paragraphs.

First of all the primary system clock on the Z-turn board is 100 MHz. This is very convenient because its period is 10 ns and therefore it can track even relatively high frequency signals. So, let's say one would like to flash an LED with 100 milli-second frequency (100 ms). Let's also assume that the duty cycle is 50% so that the time the LED will be switched on will be the equal to the time the LED will be switched off. Do the following:

$$\frac{100\ ms}{10\ ns} = 10{,}000{,}000$$

10,000,000 is the number the 100 MHz clock has to count so that 100 ms pass.

So, assuming 50% duty cycle, create a variable within an process that counts up with every rising edge of the primary clock. From 0 up to 5,000,000, the LED will be swtiched off while from 5,000,000 up to

10,000,000, the LED will be switched on. When 10,000,000 is reached the counter is reset to zero and the process will start all over.

The VHDL code for the above explanation is shown in code snippet 3.1 Note that the *reset switch* is *acitve low*, this has to be according to how the reset switch is connected *in hardware*, otherwise the counter will remain reset all the time and the LED will never light up, so be careful!

```
process(clk,reset)
variable count: integer;
begin
if reset = '0' then count := 0;LEDs(2) <= '0';
elsif clk'event and clk = '1' then count := count + 1;
    if count < 5000000 then LEDs(2) <= '0';
    elsif count >= 5000000 and count < 10000000 then LEDs(2) <= '1';
    end if;
    if count >= 10000000 then count := 0; end if;
end if;
end process;

LEDs(1 downto 0) <= "11"; --for now these two LEDs are switched off
```

Code Snippet 3. 1: Flashing LED in VHDL

Save the source code and Synthesize the code.

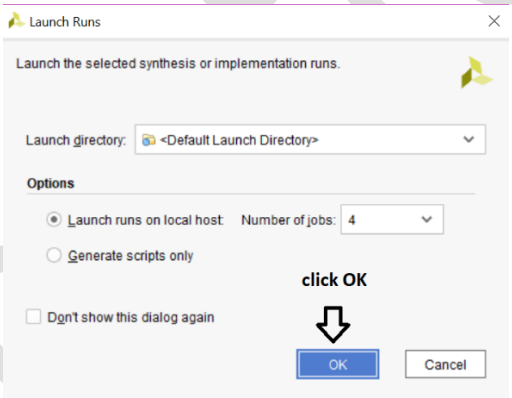


Figure 3. 4: Launch synthesis

Now, create a block design to include the VHDL entity together with the Zynq processing system.

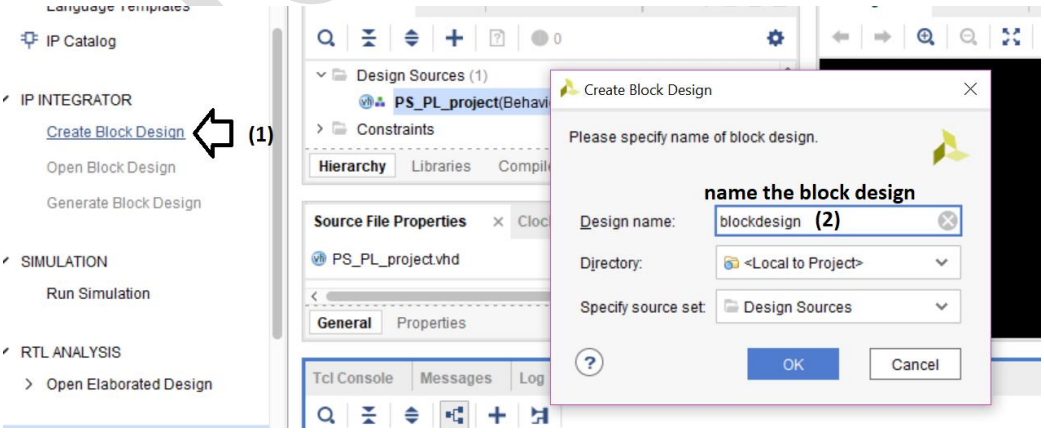


Figure 3. 5: Creating a block Design

After the block design file is created, in the empty canvas, right click on the canvas and choose Add Module from the list. This will access the VHDL entity that has been created previously and will be made available in block-form to be added in the schematic. Figure 3.6 show the procedure:

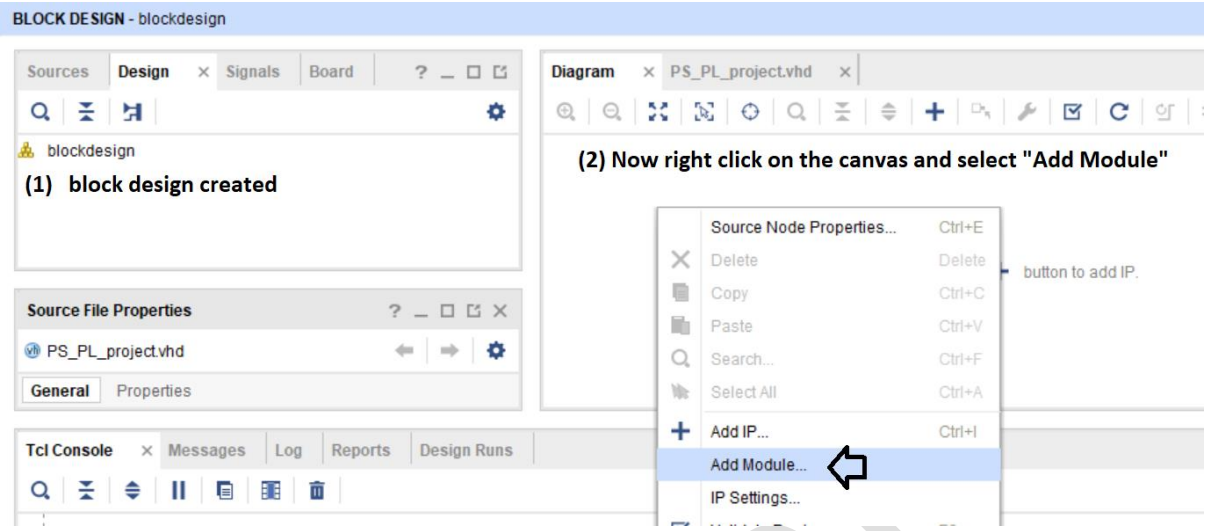


Figure 3. 6: Adding the VHDL entity in Block Design Schematic

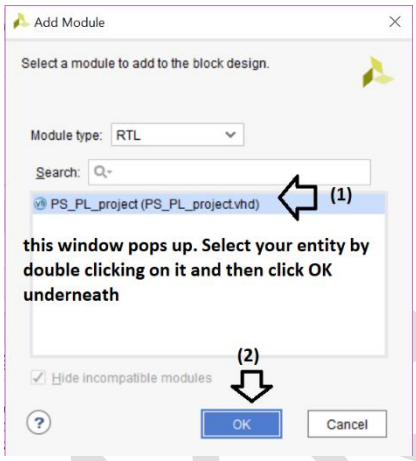


Figure 3. 7: The VHDL entity is available in the list

The adjacent pop up window shows the VHDL entities that could be added in the schematic. Double Click on it.

The entity shows up in block form in the canvas as shown in Figure 3.8 below.

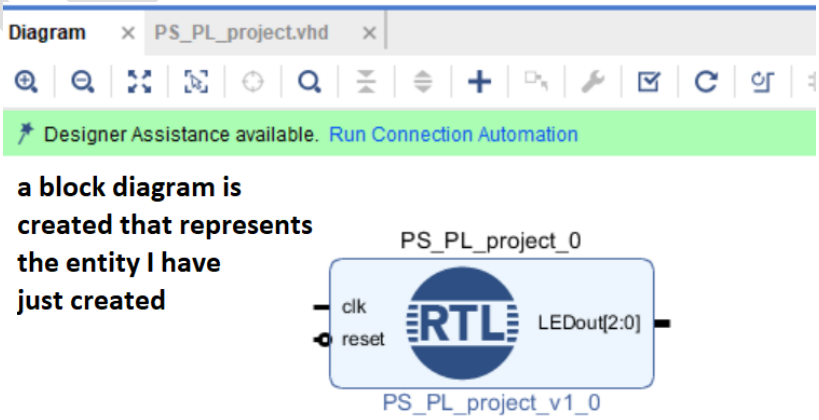


Figure 3. 8: VHDL entity in block form

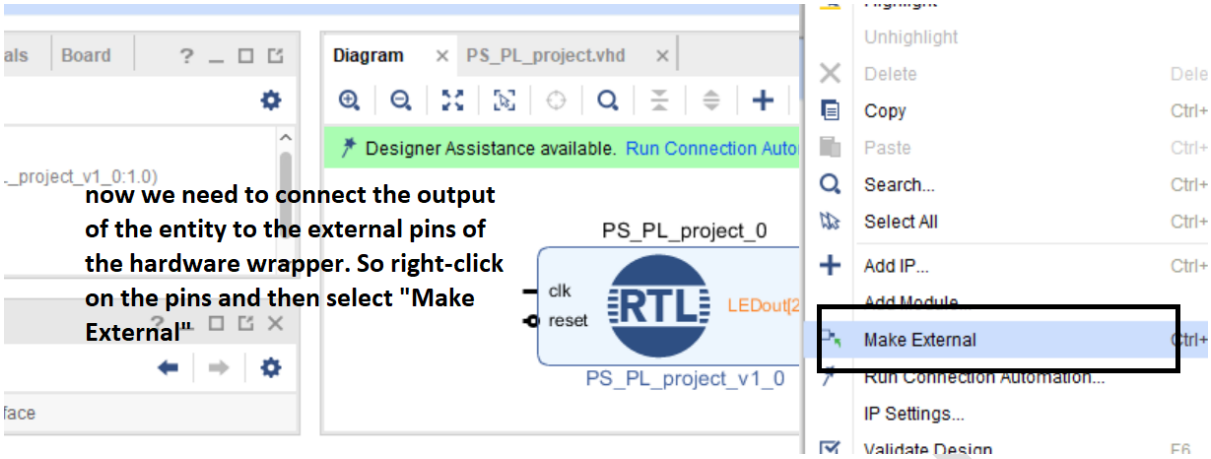
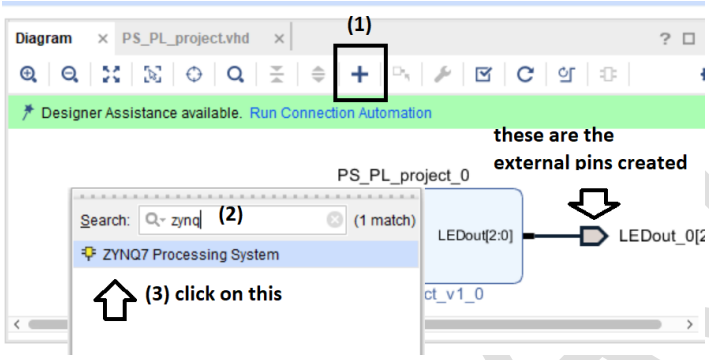


Figure 3. 9: Connecting the terminals to external pins

Figure 3.9 shows the procedure to define the pins as external pins so that later on they could be connected to physical pins of the SoC.



Now it is time to create an A9 hard-processor so click on the plus (+) sign and write "Zynq" in the field provided in the pop up window. Then click on OK.

Figure 3. 10: Including the Zynq Processing System in the Block Design

Figure 3. 11: Processing System is part of the schematic

Click on ***Run Block Automation*** to include the parameters and peripherals of the Z-turn board.

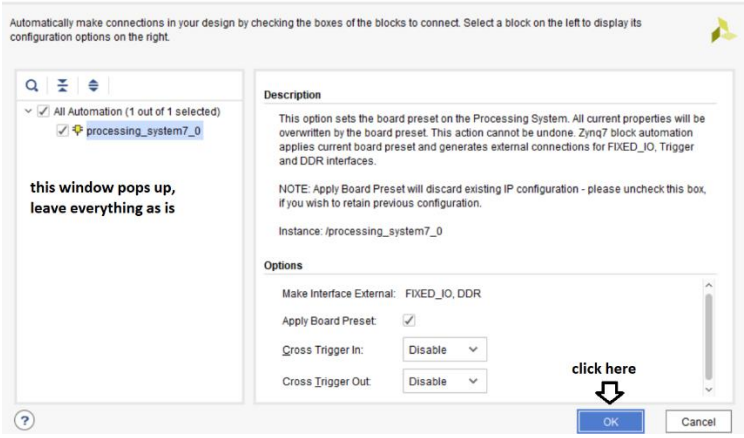
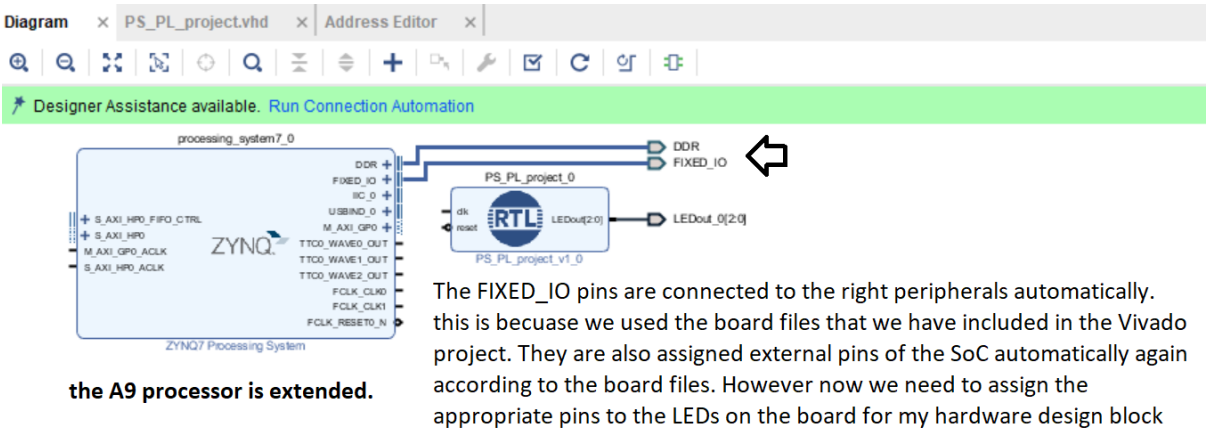


Figure 3. 12: Leave all Pre-set Settings

Click on OK for the adjacent window so that all the Z-turn board's peripherals will be included in the Vivado project.



the A9 processor is extended.

The FIXED_IO pins are connected to the right peripherals automatically. this is becuase we used the board files that we have included in the Vivado project. They are also assigned external pins of the SoC automatically again according to the board files. However now we need to assign the appropriate pins to the LEDs on the board for my hardware design block

Figure 3. 13: Zynq Processing System co-exist with VHDL entity in Block Design

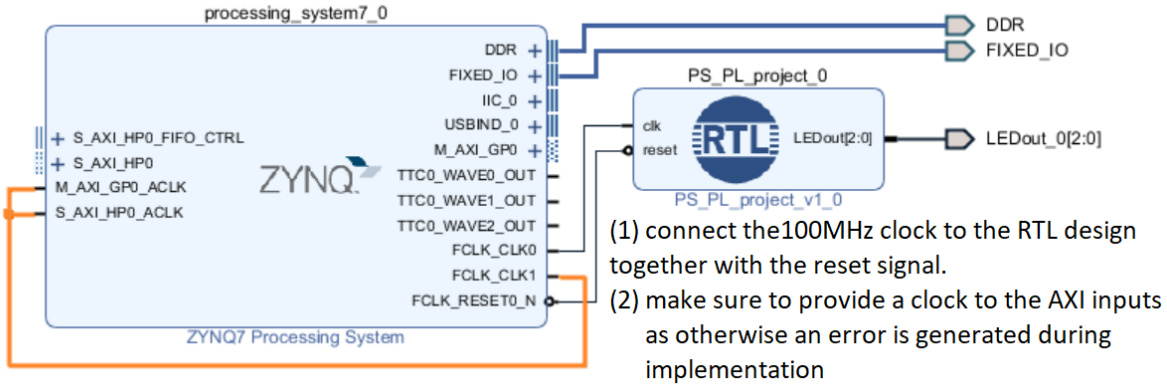


Figure 3. 14: connect the 100 MHz clock

Connect the ***100 MHz*** clock which is denoted as ***FCLK_CLK0*** on the Zynq Processing System diagram to the VHDL entity's clock input. Even though Figure 3.14 shows that the ***AXI_GPn*** inputs are connected to ***FCLK_CLK1***, it is *advisable not to do so* but to connect them also to the 100 MHz clock ***FCLK_CLK0***!

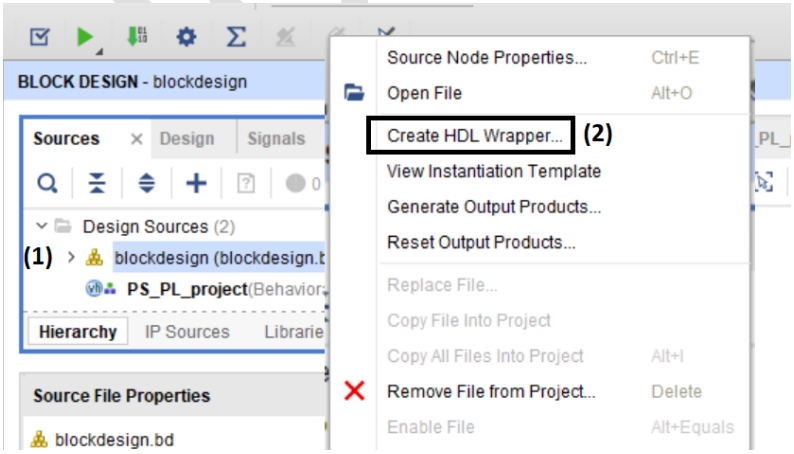
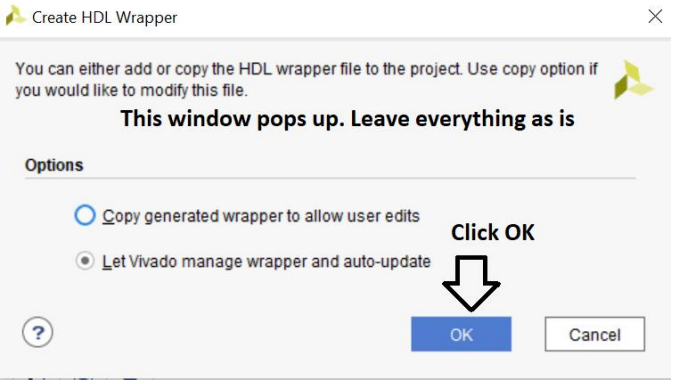


Figure 3. 15: Creating a Hardware Wrapper

To create a hardware wrapper, one needs to right-click on the block design name and select ***Create Hardware Wrapper*** from the list.



Leave Vivado to do all the work

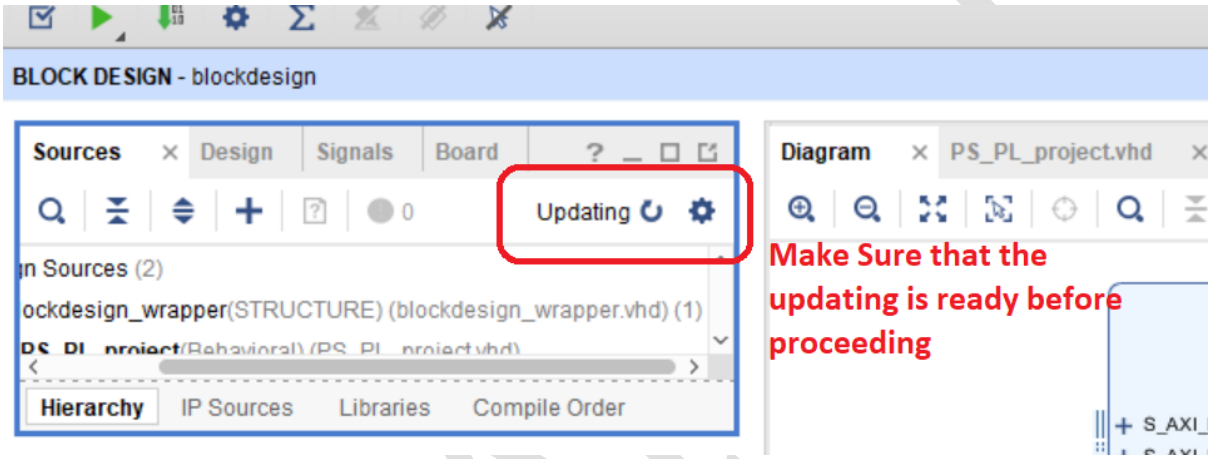


Figure 3. 16: Leave Vivado to Update

Leave Vivado to update before continuing as otherwise Vivado will get mixed up!

Now click on ***Run Implementation.***

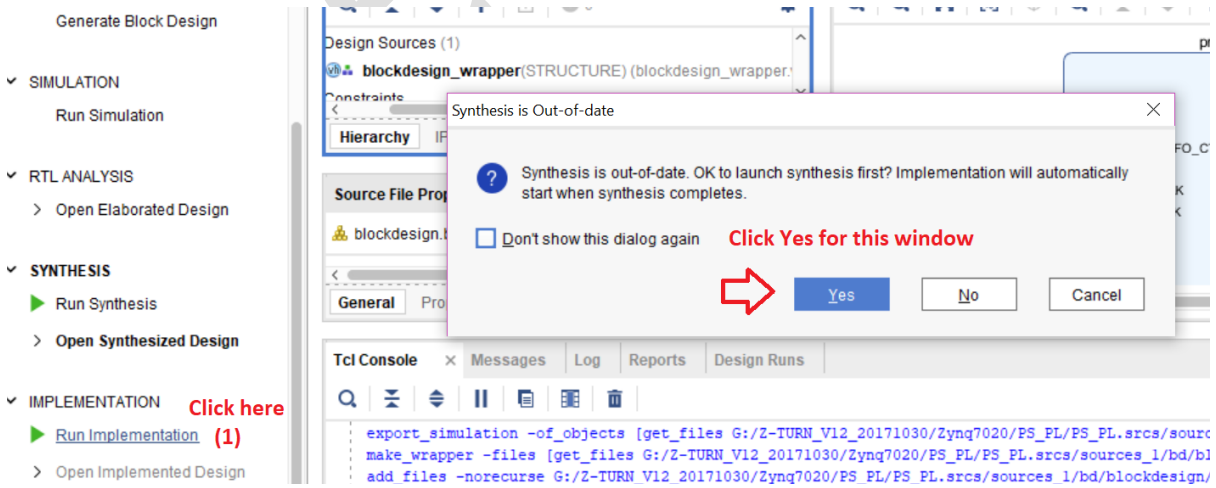
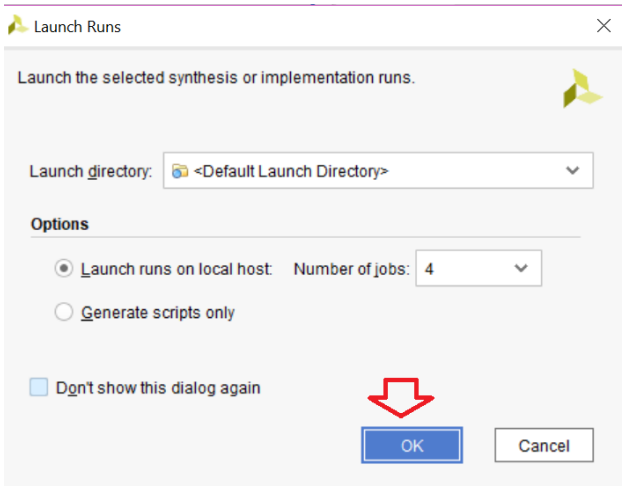


Figure 3. 17: Synthesis Out-of-date

For Figure 3.17, click ***Yes.*** This is because of the new changes that have been done since the last synthesis.



Click Yes for this window as well.

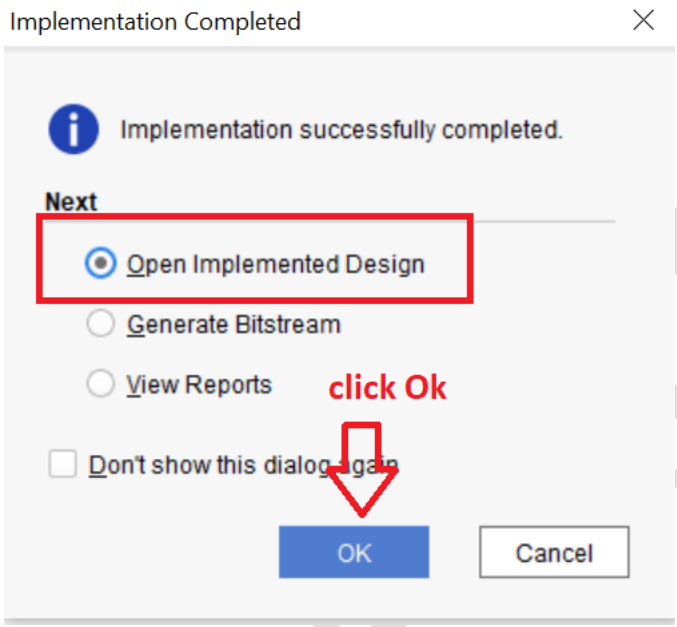


Figure 3. 18: Implementation Complete Window

Once implementation is done, it is time to assign the physical pins to the VHDL entity IO pins. These must be given according to the Z-turn board pin configuration so that the hardware will comply with the software.

On the other hand, the pinouts for the Zynq Processing System are already assigned by virtue of the Board Support Files that has been installed as described in chapter 1.

Now open the implemented design and click on the IO Port tab as shown in Figure 3.19.

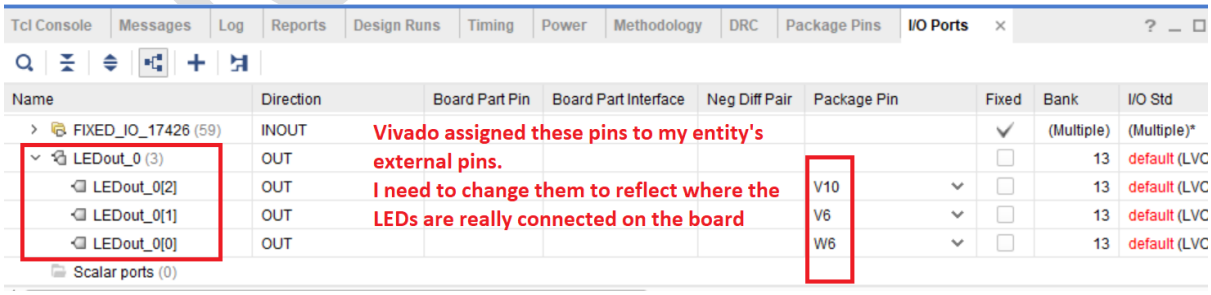


Figure 3. 19: Vivado assigns random pin numbers to terminals

Vivado assigns random pin numbers to the terminals of the hardware. These have to be changed to comply with the design of the Z-turn board.

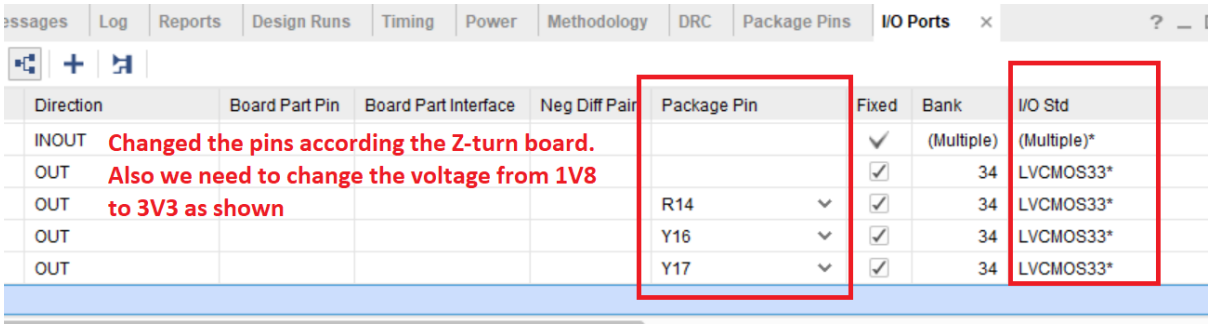


Figure 3. 20: Pins Comply with Z-turn Board

Now the pins are assigned the correct pin number so that the entity will be physically connected to the LEDs on the Z-turn board.

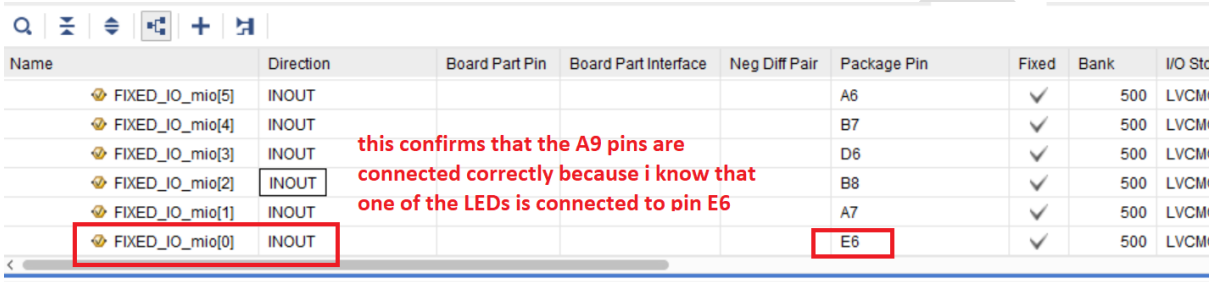


Figure 3. 21: Confirming that MIO pins are correctly assigned

Figure 3.21 confirms that the Zynq Processing System pins are correctly connected to the peripherals. This could be confirmed from the schematic diagrams provided by ***MYIR.***

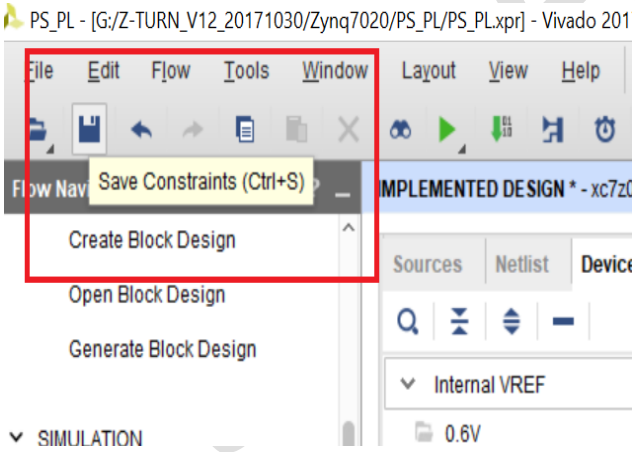
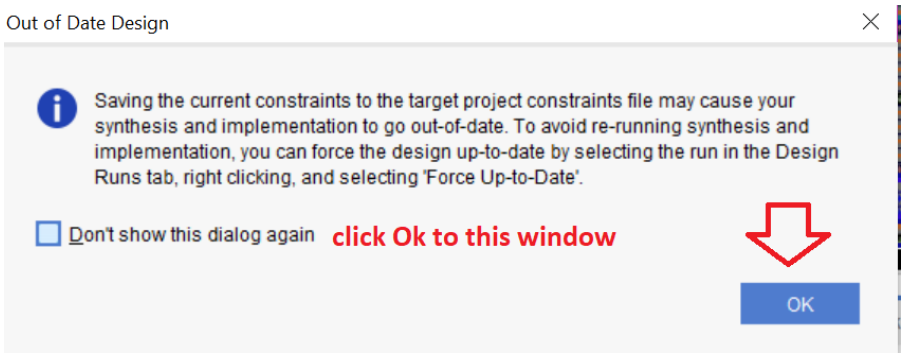


Figure 3. 22: Saving the new constraints file

Since pinouts have been changed from the ones assigned by Vivado, a new constraints file will be created and saved. This new constraints-file will be part of the project and takes precedence over the one created automatically by Vivado.



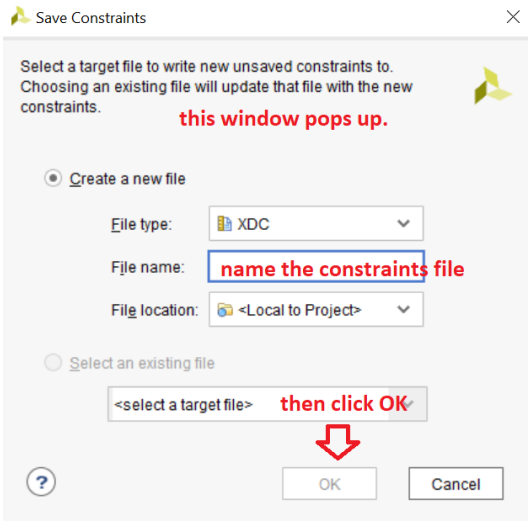


Figure 3. 23: Naming the new constraints file

Click on ***Generate BitStream***. If Any windows pop up because ***Synthesis*** or ***implementation*** are *out of date*, just click on ***OK*** to re-do these stages again.

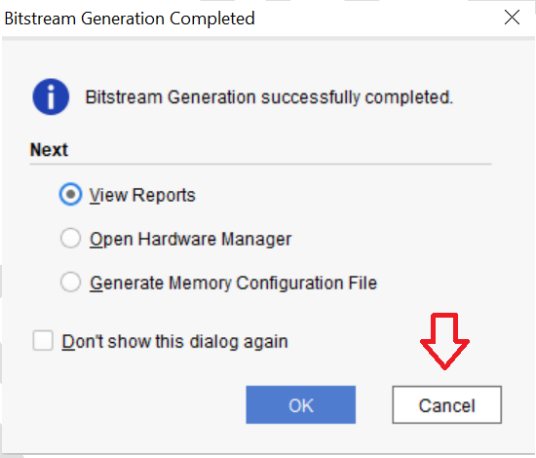


Figure 3. 24: Bitstream Generated

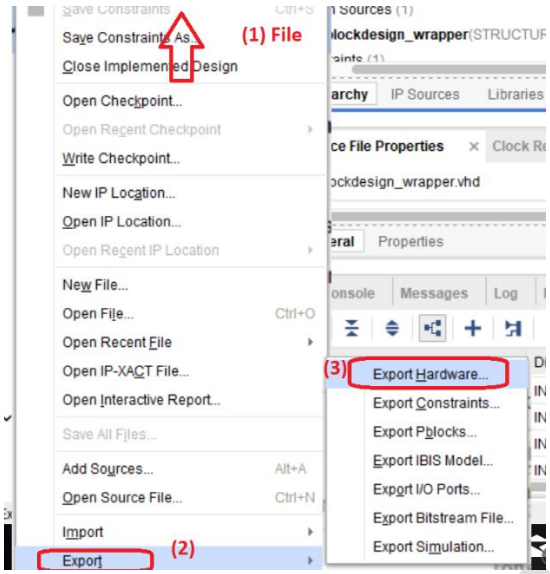
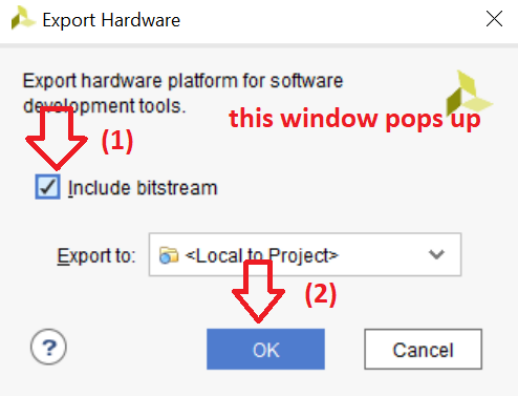


Figure 3. 25: Export the Hardware

After the bitstream file is generated, it is time to export hardware, so click on File → Export → Export Hardware.



Tick the ***include bitstream*** box and then click ***OK***

From the ***file menu***, select ***Launch SDK***. Now in the author's 13-inch laptop, sometimes this option is ***not seen*** on the screen so one might think that it is not included. Well ***it is at the very end of the file menu*** so one needs to scroll with the ***arrow-down button*** just one place down and then ***hit enter*** on the keyboard and the following window pops up as shown in Figure 3.26.

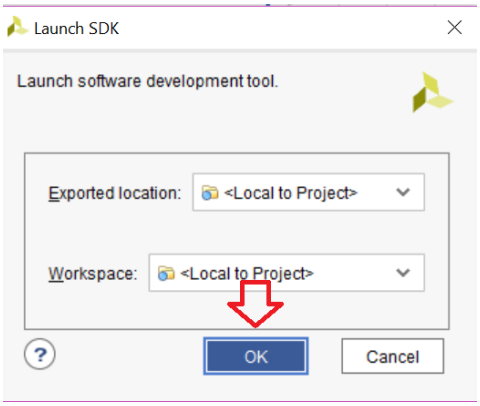


Figure 3. 26: Start SDK from within Vivado

SDK will launch automatically and by default it will be pointing to the workspace where the Vivado project is.

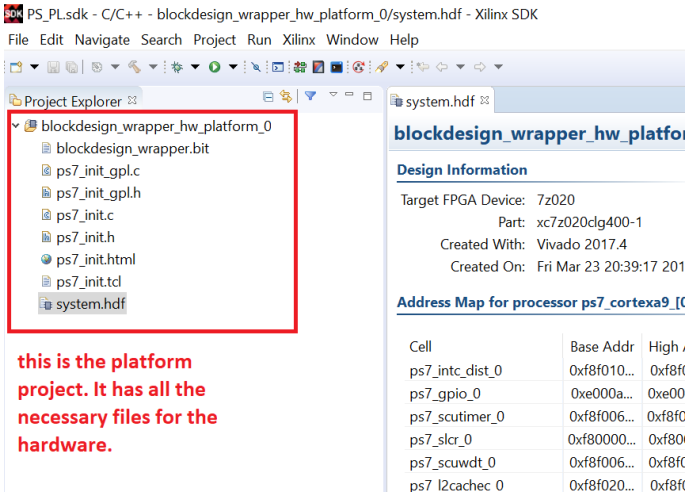


Figure 3. 27: SDK IDE

The hardware part of the project is complete, and the files needed by SDK are already loaded in SDK. At this point one must create a new application project that will generate the necessary files to boot from the SD card. This application is called the ***FSBL*** application and this is what follows next.

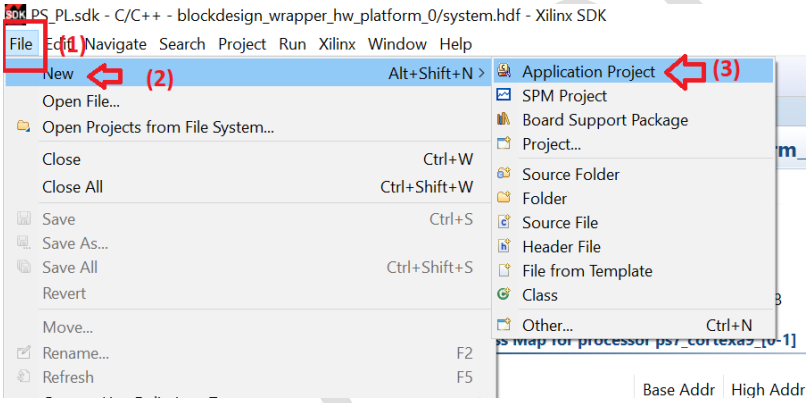


Figure 3. 28: Creating a new FSBL project

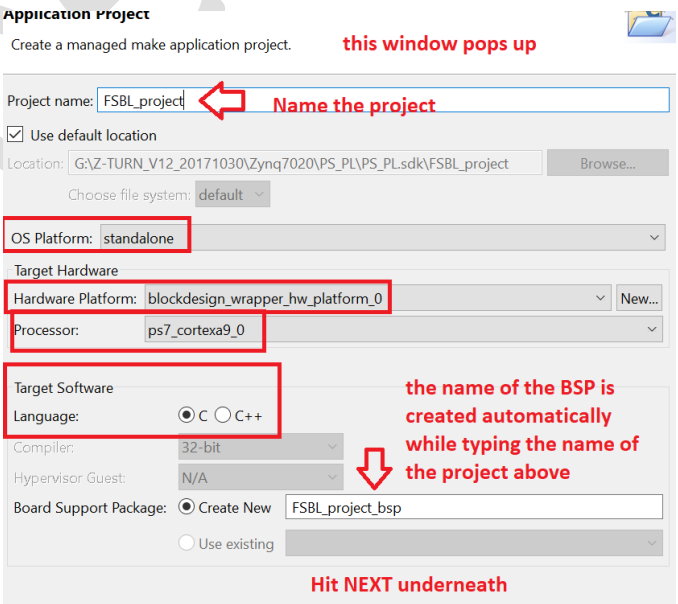


Figure 3. 29: Name the FSBL project

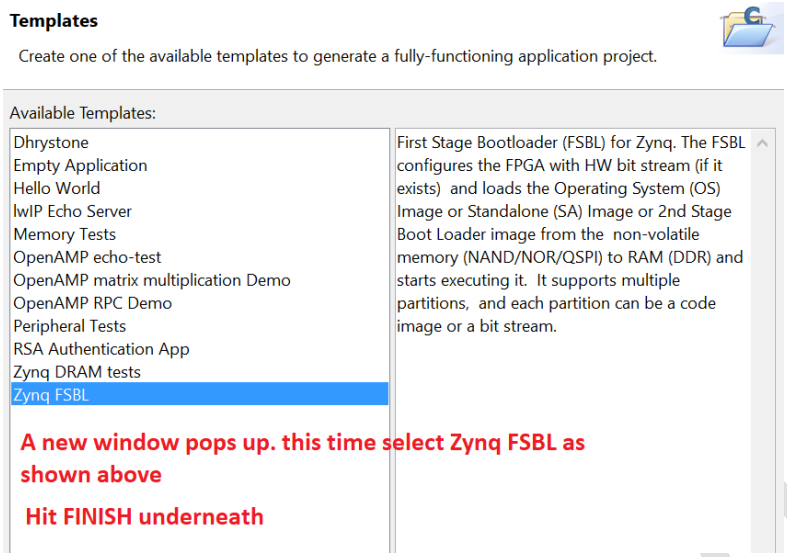


Figure 3. 30: Select the FSBL template

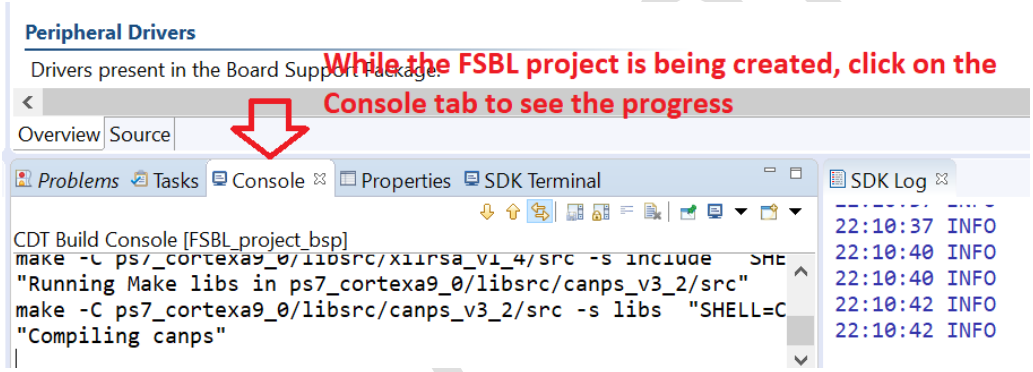


Figure 3. 31: FSBL files are being generated

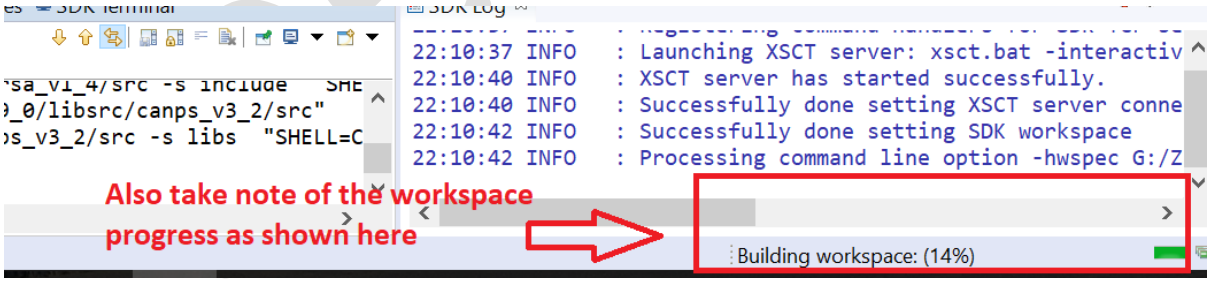


Figure 3. 32: Notice the progress bar

Make sure to check the progress bar on the lower right of the screen. This is necessary to understand when one should continue to open a new C project.

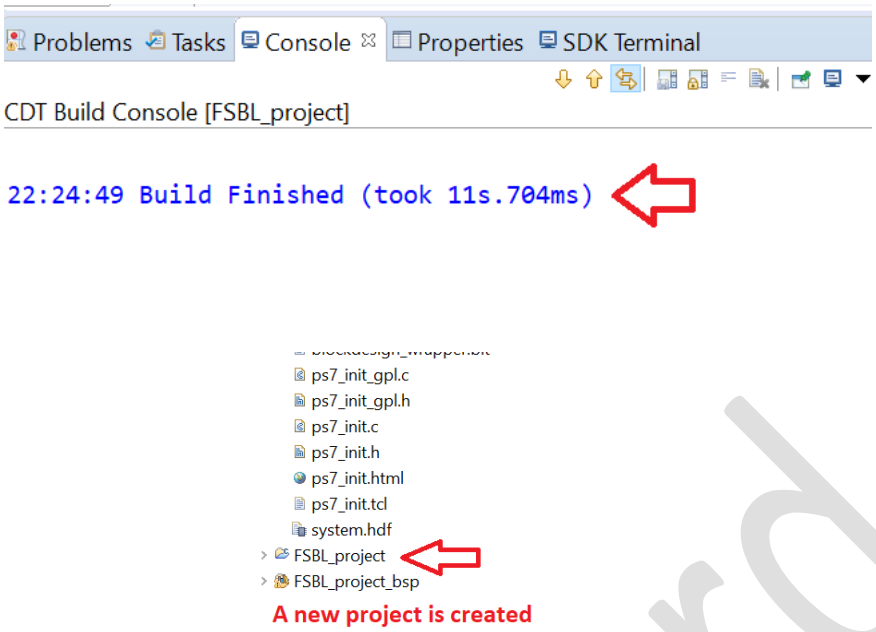


Figure 3. 33: FSBL project is created

At this point it is very important to check whether the hardware part of the system is working before trying the software part or the PS part. So create a boot image file from the FSBL project and check that the programmable logic is working.

Now create a new *C project* where C code will control the Zynq Processing System. This is done by clicking on ***File → New → Application Project then***

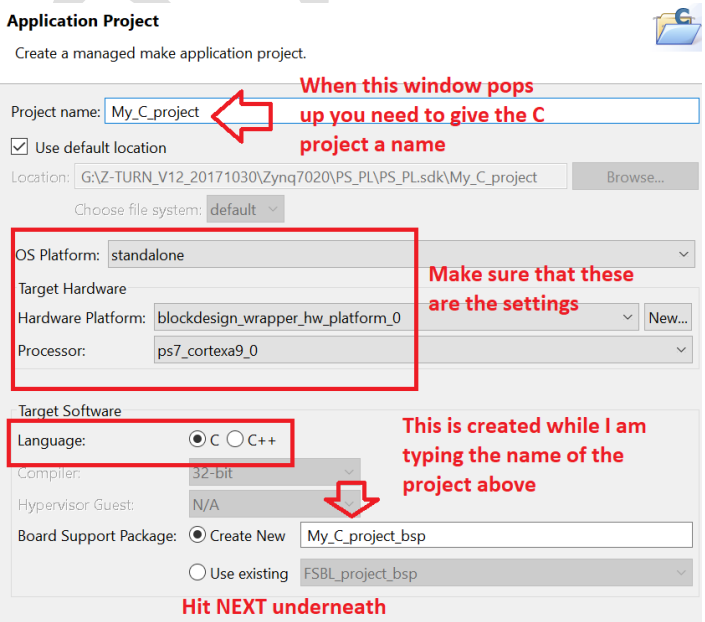


Figure 3. 34: Naming the C project

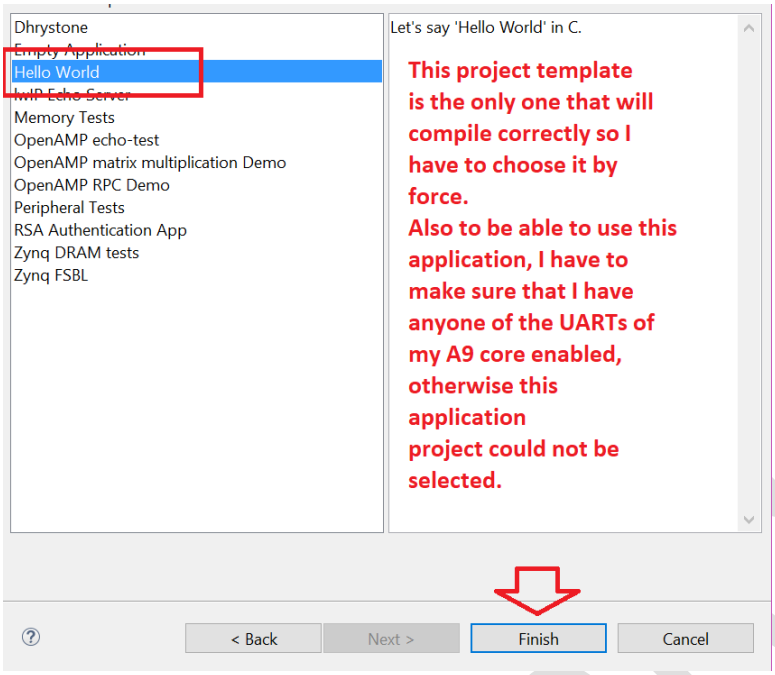


Figure 3. 35: Choosing the Hello World Template

Again, wait for SDK to finish compiling and building the workspace.

As stated before, at least one of the UARTs must be enabled for this C project to be available for the user. It must be said that the UART that could be used for debugging is ***UART 1*** and therefore one needs to make sure that it is properly enabled from the C-project's Board Support Package (BSP). The following figures will illustrate the steps.

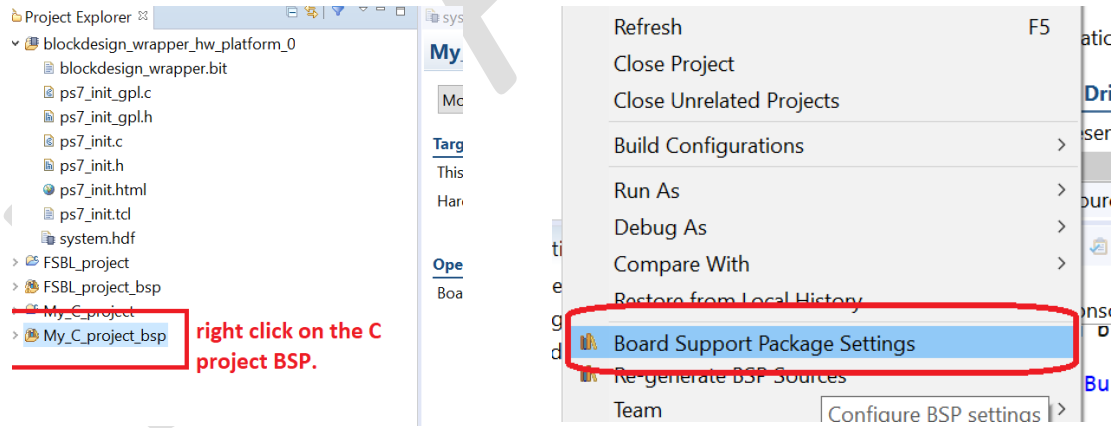
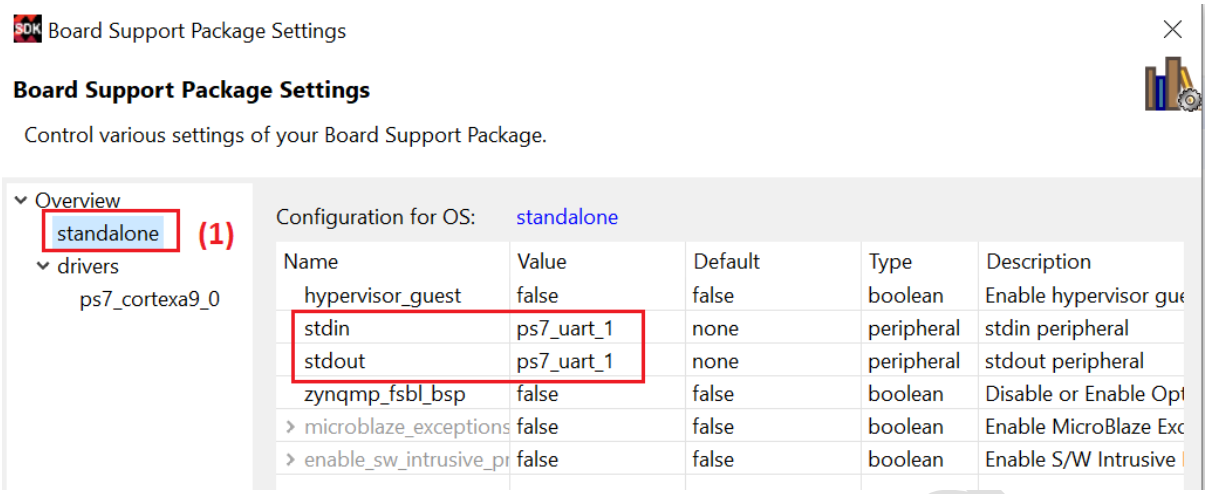


Figure 3. 36: Changing the board Support Files

When the changes are done, hit ***OK***, then the whole project is re-built again automatically.



Code Snippet 3. 2: Changing to UART 1

Locate the ***HelloWorld.c*** file:

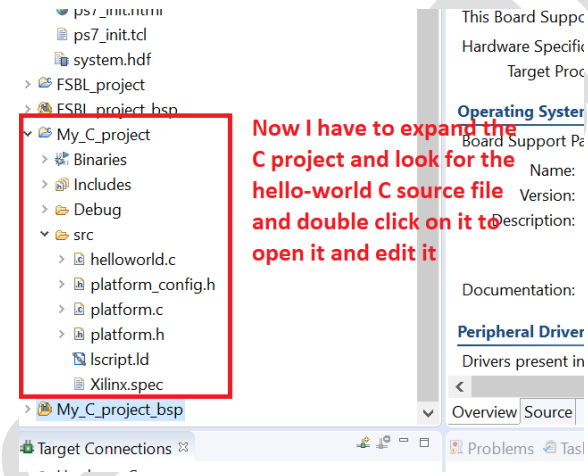


Figure 3. 37: Locating Hello World File

Software for the Processing System

Now it is time to write the software to flash the LED connected to MIO 0 and MIO 9 on the Z-turn board. The user does not have to assign any pins in the constraints file because these form part of the Board Support Files.

The first thing is to include the xgpiops.h file

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
```

Code Snippet 3. 3: Include the xgpiops.h

```c
int main()
{   int status;
    XGpioPs_Config *ConfigPtr;
    XGpioPs PS_GPIO;


    init_platform();

    /*Configuring Bank 0 which is actually bank 500 on Zynq SoC*/
     ConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
     status = XGpioPs_CfgInitialize(&PS_GPIO, ConfigPtr,ConfigPtr -> BaseAddr);
     if(status != XST_SUCCESS)
     {
         printf("initialisation failure");
         return XST_FAILURE;
     }
```

Code Snippet 3. 4: Initializing the MIO port

```c
/*set the port direction 0 = input 1=output*/
XGpioPs_SetDirection(&PS_GPIO, 0, 0xFFFFFFFF);

/*Enable the port 0 = pin disabled, 1=pin enabled*/
XGpioPs_SetOutputEnable(&PS_GPIO, 0, 0xFFFFFFFF);

/*On the Z-turn board the two LEDs assigned to the PS part of the Zynq 7
 *are connected to MIO0 and MIO9. In this program i used the Port-write
 *function instead of the pin-write function*/
```

Code Snippet 3. 5: Setting the Port Direction and Enabling the Output Port

```c
while(1)
{
    debounce();
    print("Hello World\n\r");
    XGpioPs_Write(&PS_GPIO, 0, 0x00000201);//MIO0 and MIO9
    delay();
    XGpioPs_Write(&PS_GPIO, 0,0x00000000);
    delay();

 }
cleanup_platform();
return 0;
```

Code Snippet 3. 6: Switching on and off the LEDs + printing on Serial Port

All the above instructions are explained in chapter 2 and therefore it will not be repeated here.

```c
void delay (void)
{
    for(unsigned long i = 0; i < 10000000; i++)
    {
        //do nothing
    }
}

void debounce (void)
{
    for(unsigned long i = 0; i < 100000000; i++)
    {
        //do nothing
    }
}
```

Code Snippet 3. 7: Software Delays

The delays are used so that the LEDs could be seen blinking.

Saving the C file will immediately start the build operation again. When this is finished one can create the boot image file that will be saved in the SD card. This is show in figure 3.38.

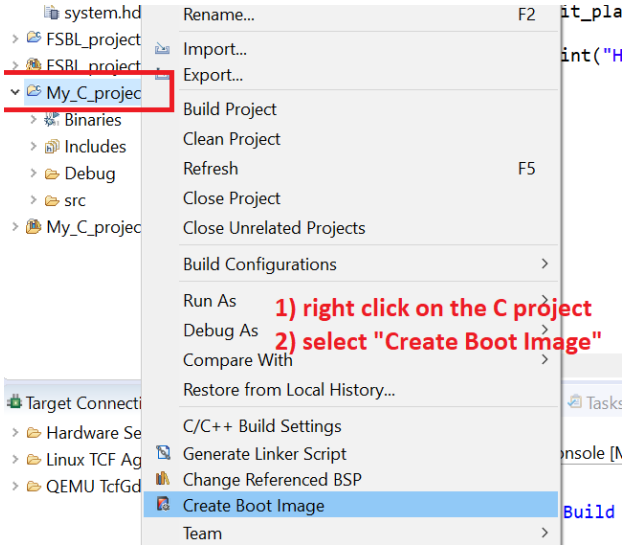


Figure 3. 38: creating a boot image file

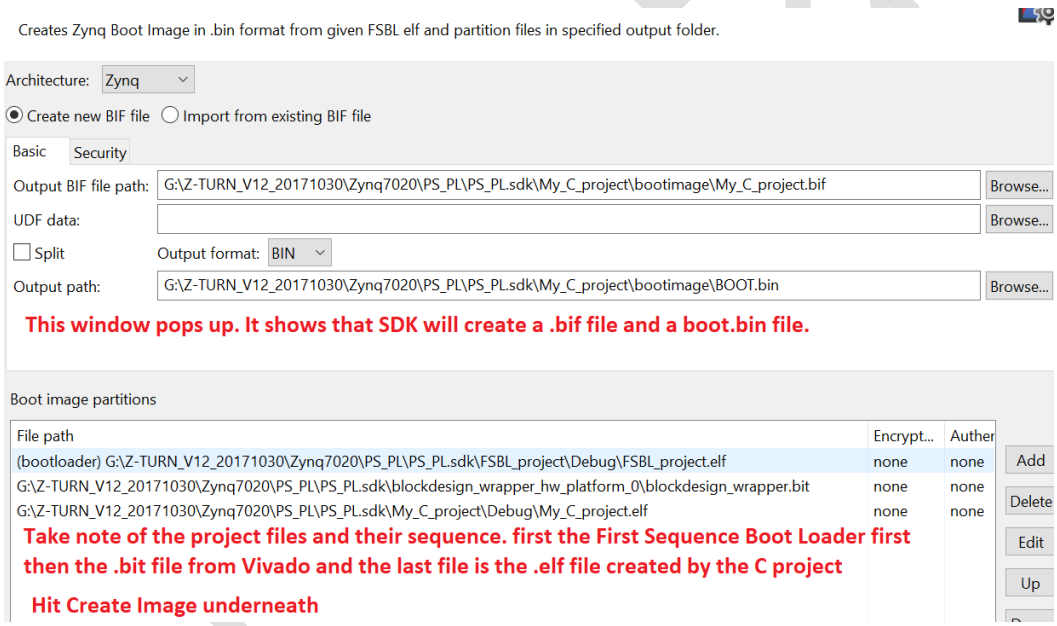


Figure 3. 39: Three files make up the boot image file

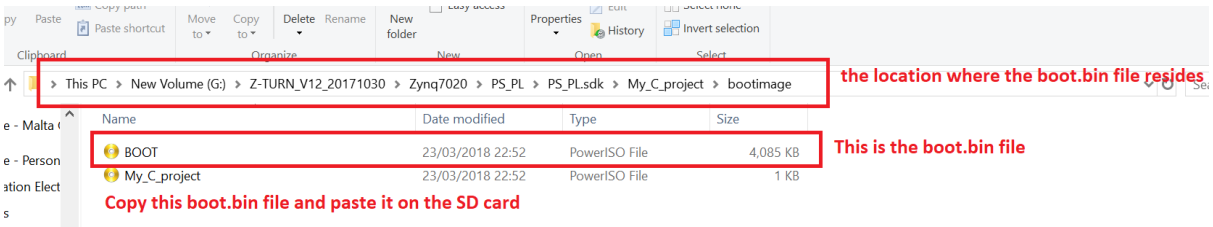
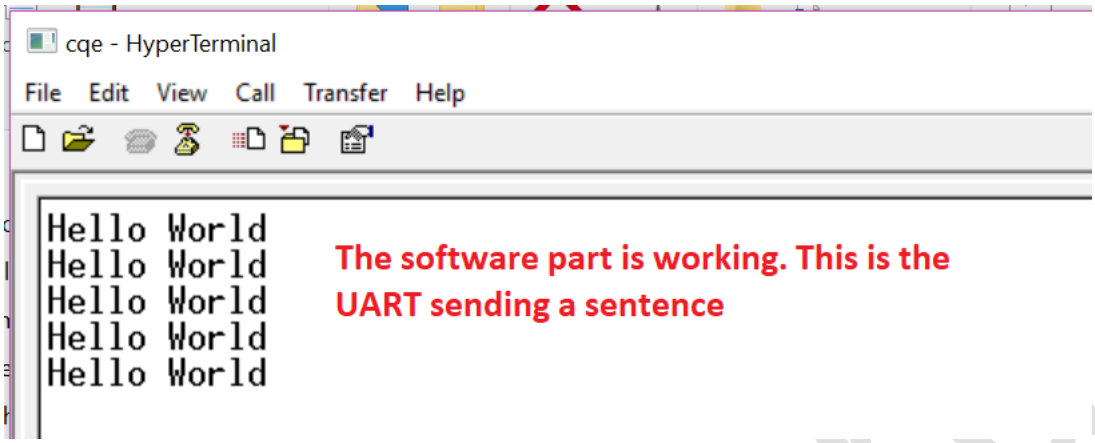


Figure 3. 40: Location of the Boot image file

Eject the SD card from the computer and insert it in the Z turn board. The program should start, there will be the two green LEDs on MIO 0 and MIO 9 blinking together with the LEDs that are connected to the Programmable Logic part. On the serial monitor, the Processing System will transmit the ***Hello World*** message.



Detecting the slide switches on the Z-turn Board from Programmable Logic

In this chapter, three out of the four DIP switches located on the Z-turn board will be used as select inputs to a multiplexer (MUX). The MUX will light a combination of LEDs according to the combination of the switches' inputs.

This chapter will focus on the interface part of the switches with Programmable logic and therefore the VHDL code is simple.

The process how to create a project together with how to include a VHDL source file has already been illustrated in previous chapters, so this will not be covered here.

Code snippet 4.1 shows the VHDL code to implement a MUX in hardware. The LEDs and the switches are both data busses.

```
entity MUX is
    Port ( sw_input : in STD_LOGIC_VECTOR (2 downto 0);
           LEDs : out STD_LOGIC_VECTOR (2 downto 0));
end MUX;

architecture Behavioral of MUX is

 begin

 LEDs <= "001" when sw_input = "001" else
         "010" when sw_input = "010" else
         "011" when sw_input = "011" else
         "100" when sw_input = "100" else
         "101" when sw_input = "101" else
         "110" when sw_input = "110" else
         "111" when sw_input = "111" else
         "ZZZ";

end Behavioral;
```

Code Snipper 4. 1: VHDL Code of a simple MUX

Create a block Design: This has already been shown in previous chapters, so it will not be repeated here. Make sure to include both the Zynq Processing System and the VHDL module (the MUX).

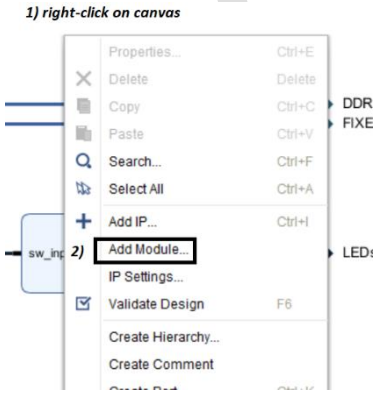
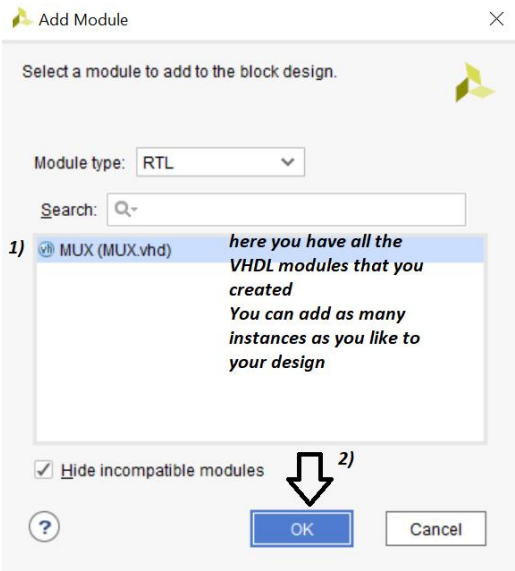


Figure 4. 1: Adding the VHDL module

1) Right click on the canvas
2) Choose ***Add Module*** from list



The adjacent window pops up. Select the VHDL module of the MUX, then click on ***OK***.

Click on ***Run Automation*** at the top of the canvas and click on OK.

Create a hardware wrapper as shown in Figure 4.2.

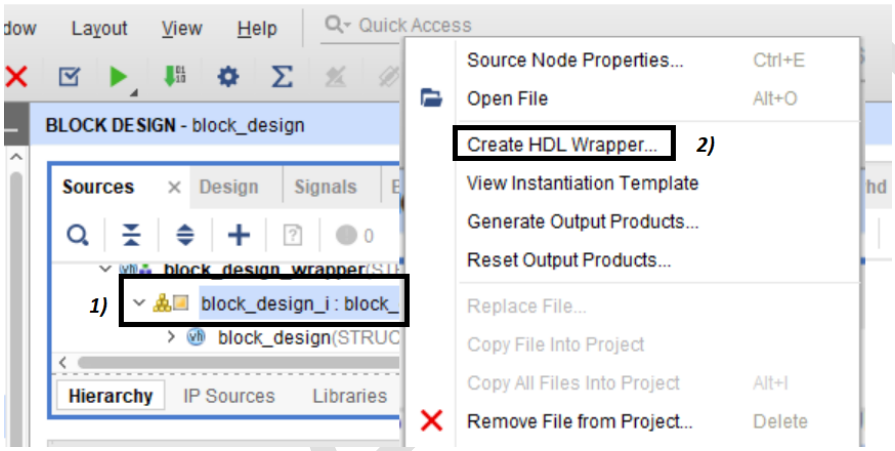


Figure 4. 2: Creating a hardware wrapper for the block design

Save the design.

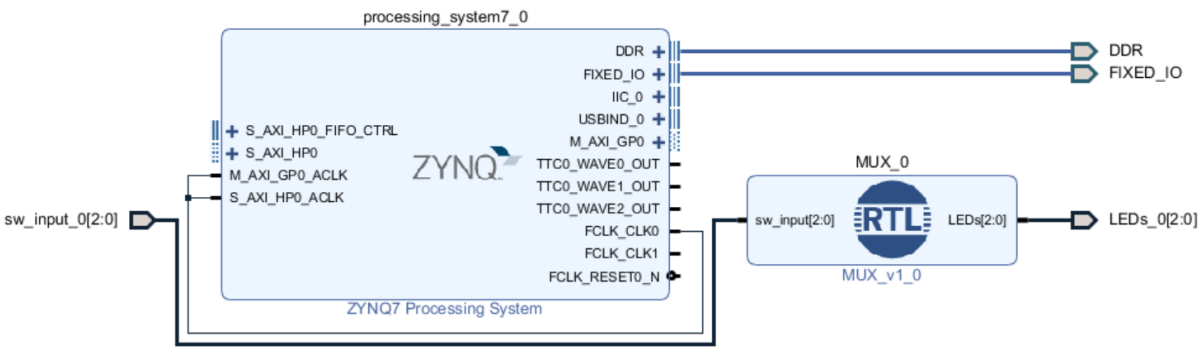
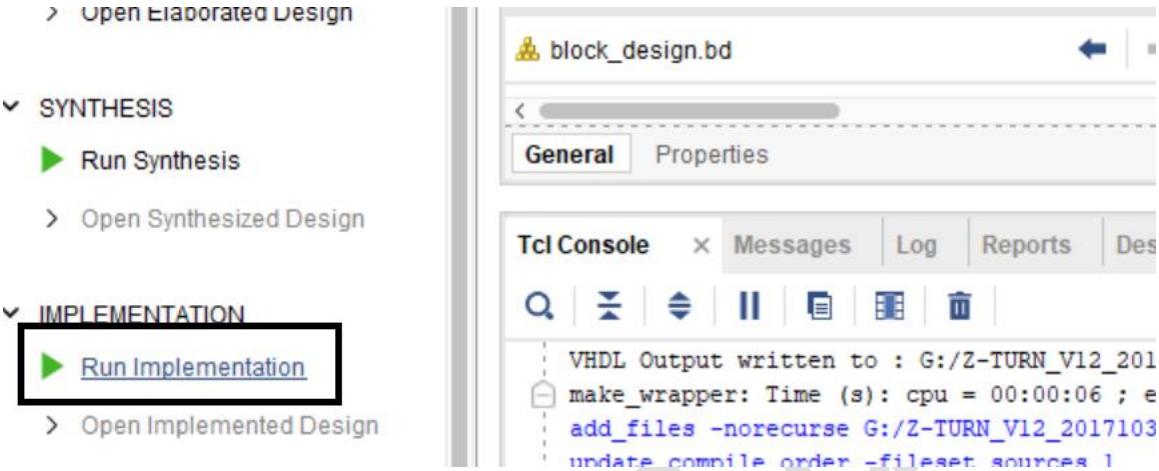


Figure 4. 3: The Full circuit diagram

Figure 4.2 shows the circuit diagram for this project. Note that ***FCL_CLK0*** is connected to ***M_AXI_GP0_ACLK*** and ***S_AXI_HP0_ACLK***. Note also that the MUX will be implemented in combinational logic, therefore no clock signal is required.

Click on ***Run Implementation***



If there are no errors then when prompted to open the implemented design, click on OK.

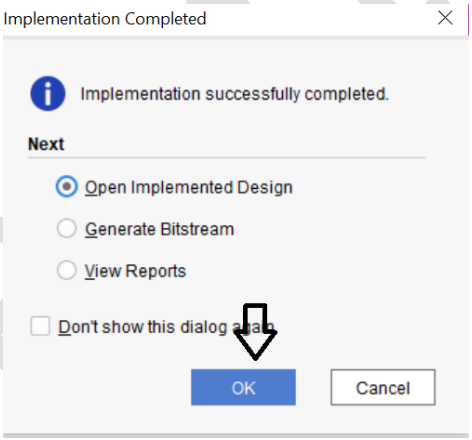


Figure 4. 4: Implementation Ready

Click on ***OK*** for the above window. Choose the ***I/O Ports*** tab and click on the arrow (>) of both ***LEDs_0*** and ***sw_inputs_0.***

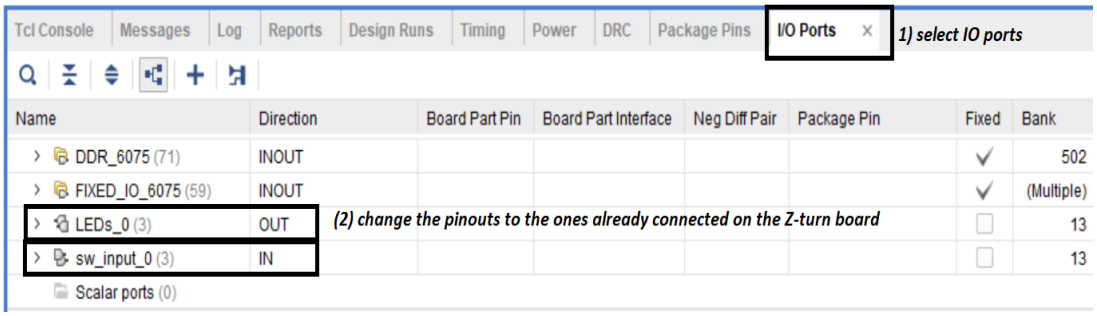


Figure 4. 5: I/O Tab

All ports (136)								
DDR_6075 (71)	INOUT					✓	502	(Multiple)*
FIXED_IO_6075 (59)	INOUT					✓	(Multiple)	(Multiple)*
LEDs_0 (3)	OUT					✓	34	LVCMOS33*
LEDs_0[2]	OUT				R14	✓	34	LVCMOS33*
LEDs_0[1]	OUT				Y16	✓	34	LVCMOS33*
LEDs_0[0]	OUT				Y17	✓	34	LVCMOS33*
sw_input_0 (3)	IN					✓	(Multiple)	LVCMOS33*
sw_input_0[2]	IN				J15	✓	35	LVCMOS33*
sw_input_0[1]	IN				G14	✓	35	LVCMOS33*
sw_input_0[0]	IN				T19	✓	34	LVCMOS33*
Scalar ports (0)								

Figure 4. 6: Pinouts of the system

To arrive at Figure 4.6, one has to do the changes illustrated step by step in Figures 4.7.

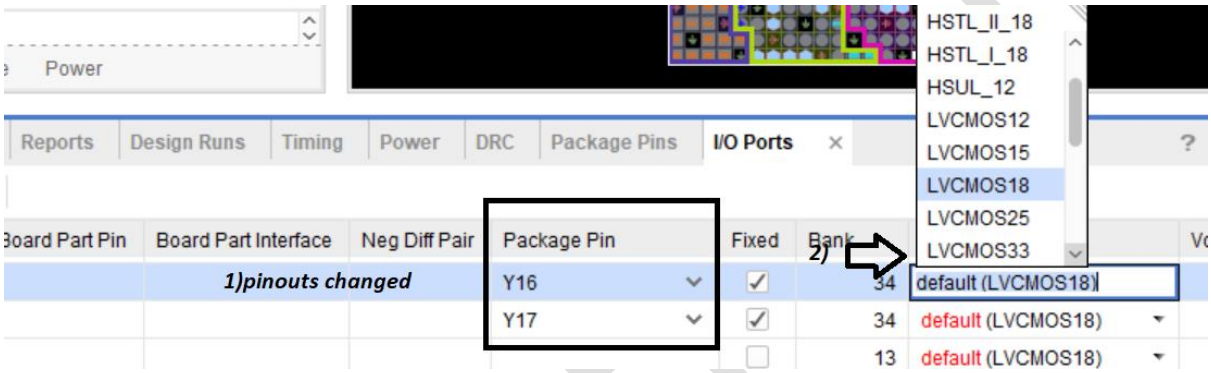


Figure 4. 7: Changing the operating voltage of the IOs and their pinouts

Figure 4.7 shows that the pinouts for the LEDs are changed to match the location of the LEDs on the Z-turn board. Apart from that one must change the operating voltage for both LEDs and switches to LVCMOS33. This will avoid errors later.

Since the constraints file was changed, Vivado asks to save the changes in a different constraints file. Give a name to the new constraints file.

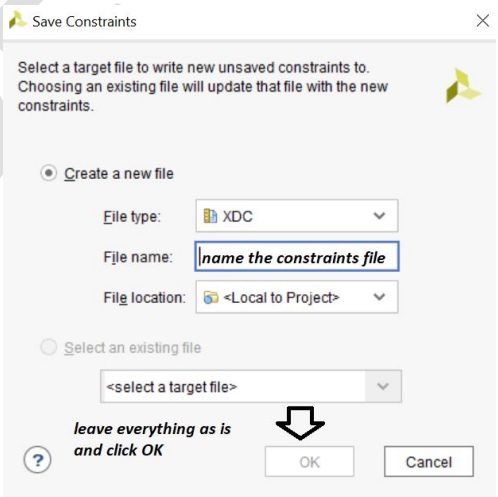


Figure 4. 8: Name the new constraints file

Double click on ***generate bitstream*** so that the .bit file is created. When the bitstream is created, one has to ***export the hardware including the bitstream file***, then launch ***SDK***.

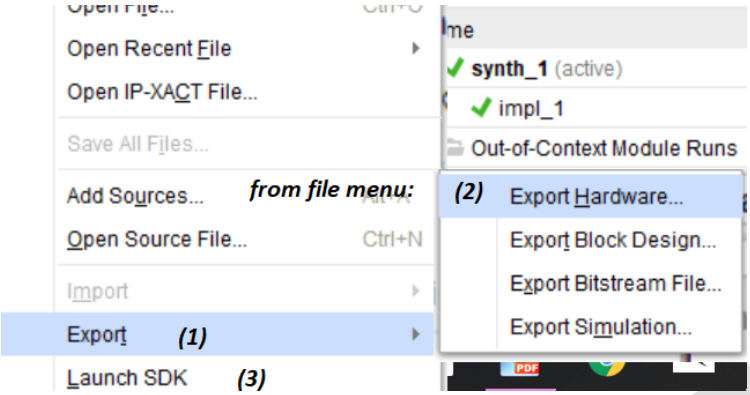


Figure 4. 9: Exporting the hardware and Launching SDK

Since the Zynq Processing System is not used in this project, there is no need to create a ***C*** code project. All one must do after the ***.bit*** file is generated, and the project is ***exported***, is create a ***FSBL*** project in ***SDK*** and create a boot image **from** the FSBL project. This is shown in Figure 4.10.

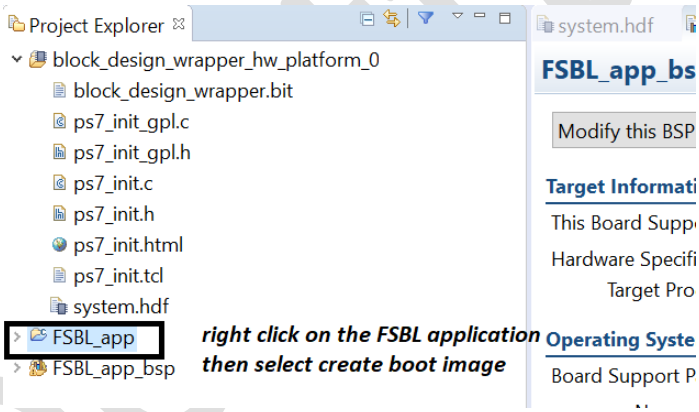
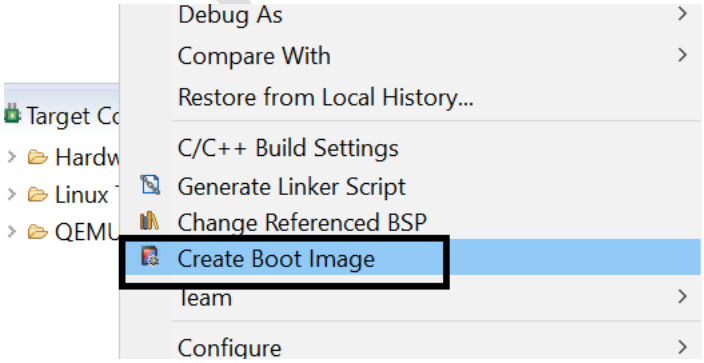


Figure 4. 10: FSBL project

The steps to create a First Stage Boot Loader project has been described in chapter 1 so there is no need to show how it is done here.



After the Boot image file is created, it can be found in the project → SDK → FSBL project →Bootimage folder.

Figure 4. 11: Create a Boot Image File

Using the DIP switches with the Processing System

In this chapter, the Processing System will monitor the state of the DIP switches which are connected to the Programmable Logic part of the Zynq 7. According to the combination of the state of the switches, the tri-colour RGB LEDs, which are also connected to the Programmable Logic will light to indicate which of the switches is active. The RGB LEDs are connected to pins R14, Y16 and Y17 while the switches are connected to J15, G14, T19 and R19. These were derived from the schematic diagram of the Z-turn board.

The following stages have been discussed in previous chapters so they will not be included again here.

1) Create a Vivado Project
2) ***DO NOT*** create a VHDL file
3) Click to create a block design
4) On the canvas click on the plus-sign (+) in the middle
5) Write ***Zynq*** in the field of the pop-up window then select the ***Zynq processing system*** from the list available
6) Right-click somewhere on the canvas and left-click on ***add IP***
7) Write AXI ***GPIO*** in the field provided in the pop-up window
8) Select the AXI GPIO from the selection list
9) ***Double-Left-click*** in the ***middle*** of one of the AXI GPIO blocks and the pop-up window in Figure 5.1 pops up.

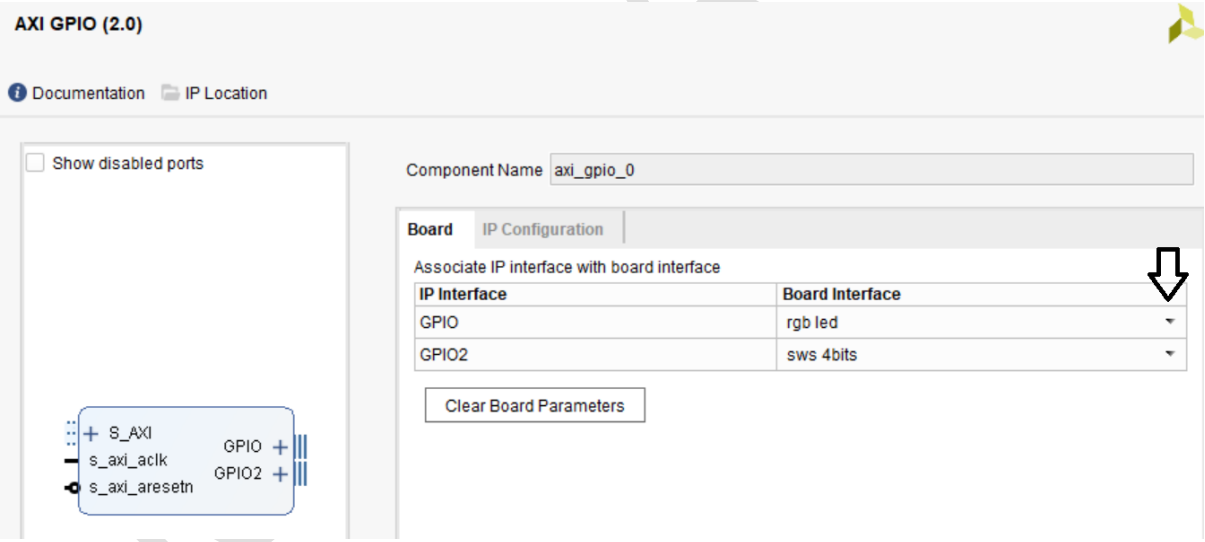


Figure 5. 1: Setup of AXI block

Figure 5.1 shows the AXI block named axi_gpio_0. This AXI block has *two channels* named ***GPIO*** and ***GPIO2***. Both channels are *32 bits* wide.

The drop down menu shows that the AXI block can be connected to the external peripherals according to the Z-turn board's support files which the user must download and install in the Vivado path so that Vivado would know which type of dev-board, one is using and therefore there are presets that could be utilized. This was discussed in chapter 1.

Since the preset configurations are going to be kept, then GPIO and GPIO2 will consist of three outputs and four inputs respectively.

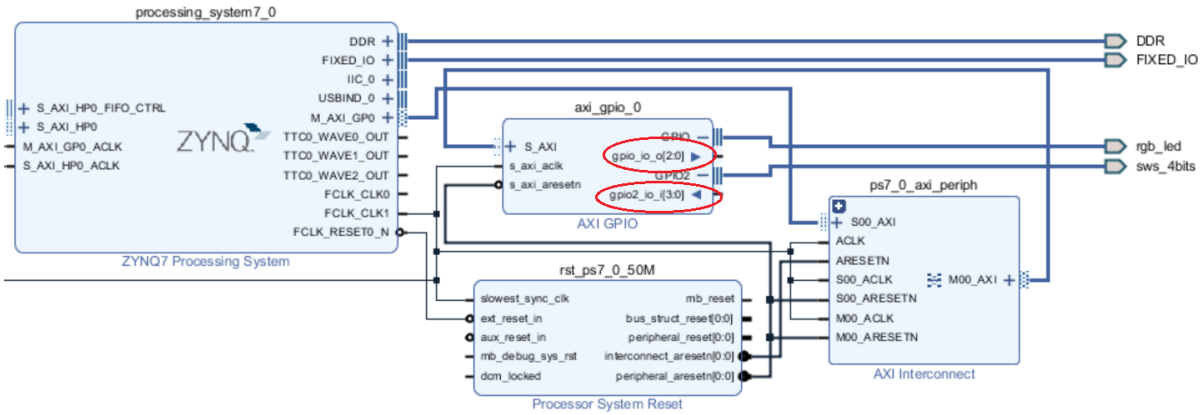


Figure 5. 2: AXI block channels configured as input and output

As shown in Figure 5.2, the channels within the AXI GPIO block assume their directions whether they are inputs or outputs automatically if the presets are used! note ***rgb_led*** and ***sws_4bits*** assigned to the pins according to the Board Support files!

Whenever there is an *AXI block*, one must include the *AXI interconnect block* together with the *Processor System Reset* block. This will make life easier at a later stage when compiling the project. So by right-clicking on the canvas and selecting the *ADD IP* for both cases, one will be able to include these two blocks in the design. Make sure to reduce the number of Master interfaces of the ACI interconnect block from 2 to 1 as shown in Figure 5.3.

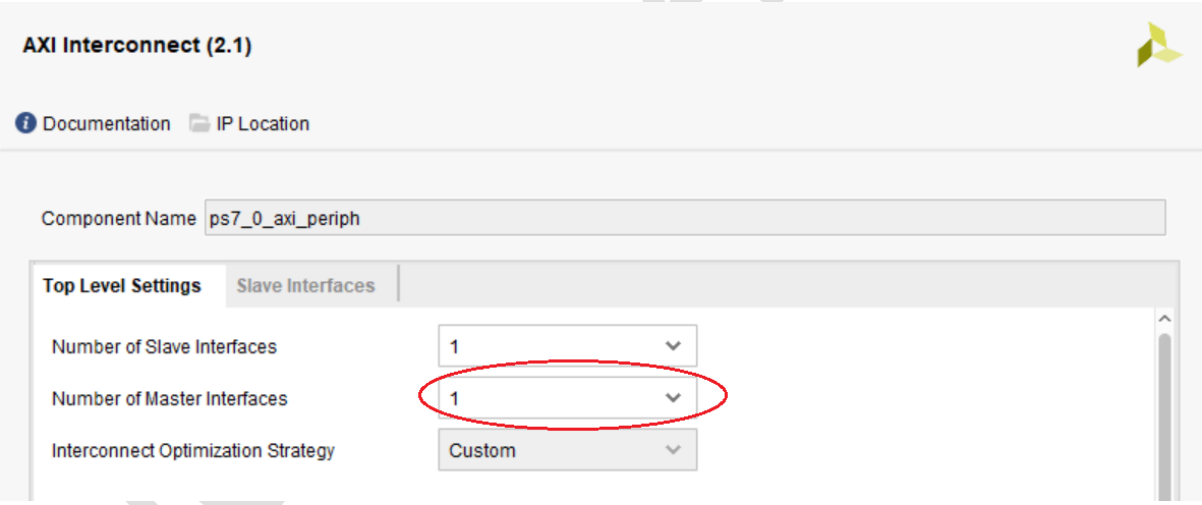


Figure 5. 3: Reduce the number of Master Interfaces in the AXI Interconnect Block

It is very important to note that, the user is not restricted to what is available on the dev-board bought! Because if **"custom"** is selected from the drop-down list, then it will become a free 32-bit GPIO which can be connected to any external peripheral.

10) After setting up the AXI blocks, one needs to click on ***Run Auto-Connection*** button on the top of the canvas. This will route all the blocks to interface the Zynq Processing System with the AXI GPIO blocks.

11) Now ***point the mouse to the pins*** of the AXI GPIO blocks and one by one, right-click and click on ***"make external***".
12) Do not forget to create a ***hardware wrapper*** by right-clicking on the block design and select ***"create hardware wrapper"*** from the list
13) After all the above is done, double-click on ***"Run implementation"***.
14) When prompted to open the implementation design, click on ***OK*** because from the implementation screen, ***one can designate the appropriate pin assignments*** according to the schematics of the dev-board. However, this time, since only the ***presets*** of the board are used, the pin assignments ***should be already correct***. It is not a bad idea if one would double-check that the pin assignments are correct!
15) At this point all that needs to be done is to generate the bitstream file.
16) After the bit stream is successfully generated, the project should be *exported* and the *bitstream included* as shown in Figure 5.3.
17) Then *SDK* could be launched from within the project. The *SDK* should be resident to the project itself so click on ***OK*** when prompted.

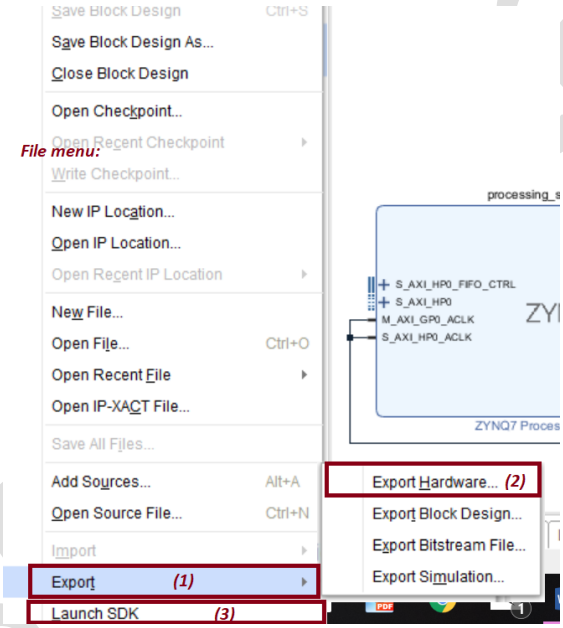


Figure 5. 4: Export Hardware include the bitstream file

Once SDK is opened it will immediately open a project for the hardware created in Vivado. At this point one needs to create a ***First Stage BootLoader Application*** from the file menu.

18) In SDK, create a FSBL application
19) Then create a hello-world application
20) Enable UART 1 from the Board Support Package of the Hello World code application by right clicking on the hello-world application project then select ***change the referenced BSP***

Points 18 to 20 have already been shown in previous chapters.

Now the software part

The focus of this chapter is on this part because the previous parts should by now be familiar with the learner. So first include the libraries for the AXI GPIO block. This is called ***gpio_v4_3*** and could be found in the path shown in Figure 5.5

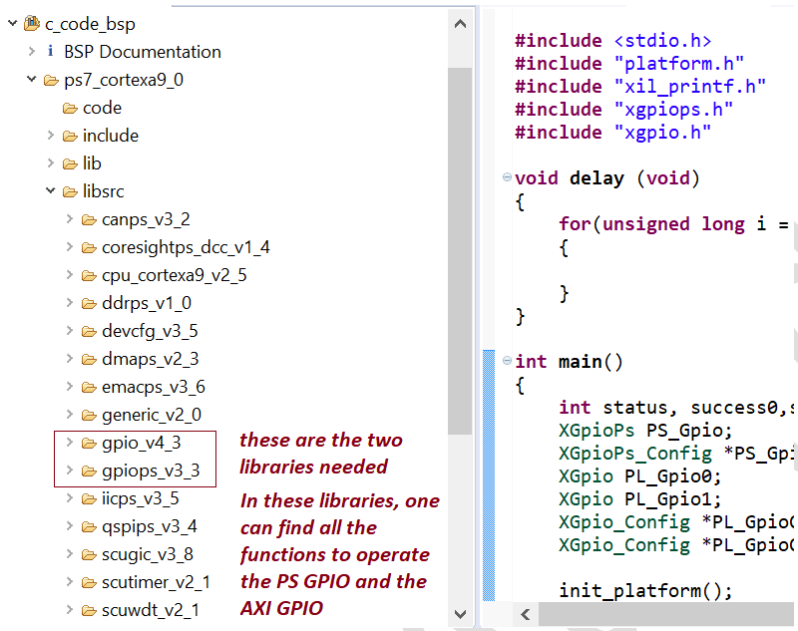


Figure 5. 5: Path to find the AXI GPIO library

All the functions that control the AXI GPIO block are listed in the library shown in Figure 5.5 above. For this project, the two LEDs connected to MIO 0 and MIO 9 will also be used. Incidentally these two LEDs are connected to pins **E6** and **B5**, however nothing should be done cause its already part of the constraints file due to the board support files.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xgpio.h"

void delay (void)
{
    for(unsigned long i = 0; i < 1000000;i++)
    {

    }
}
```

include these two header files

this is just a normal delay

The ***xgpiops.h*** header files are included in the C source file to have access to gpiops_v3_3 library or the MIO port pins. The ***xgpio.h*** is included to have access to the Programmable Logic pins via the AXI GPIO block.

When the processor enters the delay function, it till loop in the delay for 1 million times and then returns to the main program. That way, the main program is slowed so the LEDs can be seen blinking. Next initialize the MIO port and the AXI block as shown in the code on the next page.

```
/*Initialise PS GPIO driver*/
/*XGpio_Config *XGpioPs_LookupConfig(u16 DeviceId);*/
  ConfigPtr= XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
status= XGpioPs_CfgInitialize(&myPSGpio, ConfigPtr,ConfigPtr->BaseAddr);
    if (status != XST_SUCCESS)
          {
           print("cfg init err\n");
           return XST_FAILURE;
          }
```

Code Snippet 5. 1: Initializing the MIO port

The above function is found in ***xgpiops_sinit.c*** file. The argument ***u16 DeviceId*** is found in ***xgpiops_g.c*** file. The original function is the following:

XGpioPs_Config ***XGpioPs_LookupConfig**(u16 DeviceId)

XGpioPs_Config is the return type of this function so this function must be equated to a pointer that has the same attributes as *XGpioPs_Config*. This is done by declaring a pointer of the same type at the beginning of the main () and equate that pointer to the above function. So, declare the variable:

XGpioPs_Config *ConfigPtr;

Then equate it to the function:

ConfigPtr= XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);

After the LookupConfig(), one has to initialize the port. This is done by

s32 **XGpioPs_CfgInitialize**(XGpioPs *InstancePtr, XGpioPs_Config *ConfigPtr,u32 EffectiveAddr)

The above function returns a ***s32*** value so one must declare a variable of type ***s32*** at the beginning of the main () and equate it to this function. This is done below:

s32 status;

XGpioPs *InstancePtr must also be declared at the beginning of the main (), like so:

XGpioPs myPSGpio;

XGpioPs_Config *ConfigPtr has to be replaced with ***ConfigPtr*** like before and for ***u32 EffectiveAddr,*** one must write ***ConfigPtr -> BaseAddr.*** This was defined in ***xgpiops_hw.h*** file.

```
#define XGpioPs_WriteReg(BaseAddr, RegOffset, Data) \
        Xil_Out32((BaseAddr) + (u32)(RegOffset), (u32)(Data))
```

The ***IF*** statement that follows will check whether the previous function has been successful. If not, the program will stop there and nothing else happens.

Now to initialize the AXI GPIO block, the following function found in ***xgpio_sinit.c*** must be used.

int **XGpio_Initialize**(XGpio * InstancePtr, u16 DeviceId)

again, the above returns a variable or type ***int*** and therefore such a variable has to be declared at the beginning of the main (). The ***XGpio * InstancePtr*** must be replaced with an instance pointer that must be declared as well. This is shown in the following declarations:

XGpio myGpio;

And this should be written with an ***ampersand*** (**&**) sign in front of it.

int success;

The ***device ID*** should be copied from ***xgpio_g.c*** file. The complete statement is shown below:

```
/* Initialise the PL GPIO driver*/
    /*int XGpio_Initialize(XGpio *InstancePtr, u16 DeviceId);*/
    success=XGpio_Initialize(&myGpio,XPAR_AXI_GPIO_0_DEVICE_ID);
```

Just like for the initialization of the MIO port, one can use an IF statement to verify the success of the initialization. Here it is not included; however it is a good idea that it will be done.

From ***xgpiops.c*** file, use

void XGpioPs_SetDirectionPin(XGpioPs *InstancePtr, u32 Pin, u32 Direction)

The above function does not expect a return variable, so it does not have to be equated. For the instance pointer argument should be replaced with ***&myPSGpio***, the ***u32 Pin*** should be replaced with the pin number – in this case ***0*** since one of the LEDs is connected to bit 0 and the direction argument should be filled with ***1*** because it defines an *output* while ***0*** defines an *input*.

```
/*Now we need to set the port direction of each pin in the PS GPIO*/
 XGpioPs_SetDirectionPin(&myPSGpio,0,1); //MIO[0] set as output
 XGpioPs_SetDirectionPin(&myPSGpio,9,1); // MIO[9] set as output
 /*both MIO outputs above have an LED connected with them on the board*/
```

Now as an observation, even though the project works, however since the last argument is defined as a *32-bit* argument, one must write it in 32-bit form so it would be advisable to write the second function for MIO 9 as follows:

XGpioPs_SetDirectionPin(&myPSGpio,9,0x00000200); // MIO[9] set as output

Or to be even safer one should use the bank function and not the individual pin function as follows:

void XGpioPs_SetDirection(XGpioPs *InstancePtr, u8 Bank, u32 Direction)

where ***XGpioPs *InstancePtr*** is replaced with ***&myPSGpio*** as before, the ***u8 Bank*** will be replaced with ***0*** because it is bank 0 and ***u32 Direction*** will be replaced by 0x00000201

XGpioPs_SetDirection(&myPSGpio, 0, 0x00000201);

It is time to enable the outputs so from xgpiops.c file copy:

void XGpioPs_SetOutputEnablePin(XGpioPs *InstancePtr, u32 Pin, u32 OpEnable)

XGpioPs *InstancePtr is replaced with ***&myPSGpio***, ***u32 Pin*** is replaced with ***0 and 9*** respectively in different function calls, and ***u32 OpEnable*** is replaced with ***1*** declaring that the output is now enabled.

```
/*for the MIO outputs to work, i need to enable them by */
XGpioPs_SetOutputEnablePin(&myPSGpio,0,1); //enable MIO[0] pin
XGpioPs_SetOutputEnablePin(&myPSGpio,9,1); //enable MIO[9] pin
```

Now for the AXI GPIO block, one only has to set the direction of the individual channels as a whole and not individual pins. This is illustrated below:

```
/* Now i am going to set the direction of the PL GPIO*/
XGpio_SetDataDirection(&myGpio,2,0x0000000F); //bits 3:0 are connected to switches
XGpio_SetDataDirection(&myGpio,1,0x00000000);//RGB LEDs are connected to the
//lower 3 bits of this port
```

Code Snippet 5. 2: Setting the AXI GPIO direction

The function can be found in ***xgpio.c*** file.

void XGpio_SetDataDirection(XGpio *InstancePtr, **unsigned** Channel,u32 DirectionMask)

XGpio *InstancePtr is replaced with ***&myGPIO***, channel argument is replaced with either ***1 or 2***. ***1*** represents ***GPIO channel*** while ***2*** represents ***GPIO2 channel***. ***U32 DirectionMask*** determines the pin direction – in the AXI GPIO case, *logic 0* represents that pin is an *output* while *logic 1* means that the pin is an *input*. It is the reverse for the MIO port!

The code snippet 5.2 shows that channel 1 is declared as output while channel 2 is declared as input.

Now for the while (1) code. A while (1) statement defines an infinite loop. This means that the microcontroller will continue looping inside this loop forever. The code must check the state of the switch-bank and according to the state of each switch, it will light a combination of the RGB LEDs. To check the state of the switches one must read the whole channel and filter out the un-needed bits. This is called Bit-Masking where a bitwise-AND-function is done with individual bits of the channel. By ANDing with 0 all the bits that are not of interest will be discarded while when ANDing with 1 – the bits will be considered.

An if statement was used that selects the pattern of the LEDs according to the state of the switches.

The bank-read function was used to read the state of the bank as a whole shown below:

u32 **XGpioPs_Read**(XGpioPs *InstancePtr, u8 Bank)

it returns an ***unsigned 32-bit*** variable containing the state of the whole channel. Again ***XGpioPs *InstancePtr*** is replaced by &myGPIO while u8 Bank is replaced by the channel number, in this case 2 because that is where the switches are assigned. In the same statement the returned value from the read function is immediately ANDed with another variable to extract the state of the switches. It must be noted that, the switches are effectively connected to the least significant nibble of the channel.

The code lights the LEDs connected to the MIO port if the combination of the switches does not match any from the if statements, otherwise the MIO LEDs will be switched off while the LEDs on the Programmable Logic part will reflect the combinations of the switches' inputs. Code is show in the next page.

```
while(1)
{
print("press a switch\n\r");
delay();
sw_detection = (XGpio_DiscreteRead(&myGpio,2) & 0x0000000F);//read channel 2 cause switch
//are connected to channel 2 bits 3:0
/* Switches are connected as active low switches */
if(sw_detection == 0x00000001)
{
    XGpio_DiscreteWrite(&myGpio,1,0x0000000E); //LED will light with logic 0
//  XGpioPs_Write(&myPSGpio, 0,0x00000000); //switch off LEDs on PS bank 0
}
else if(sw_detection == 0x00000002)
{
    XGpio_DiscreteWrite(&myGpio,1,0x0000000D); //LED will light with logic 0
//  XGpioPs_Write(&myPSGpio, 0,0x00000000); //switch off LEDs on PS bank 0
}
else if (sw_detection == 0x00000004)
{
    XGpio_DiscreteWrite(&myGpio,1,0x0000000B); //LED will light with logic 0
    //XGpioPs_Write(&myPSGpio, 0,0x00000000); //switch off LEDs on PS bank 0
}

else
{
    XGpio_DiscreteWrite(&myGpio,1,0x0000000F); //switch off LEDs on PL side
//  XGpioPs_Write(&myPSGpio, 0,0x00000009); //write in bank 0 to affect MIO[9]
                                            // and MIO[0]
}
```

Interfacing with the Button Switch on the Z-turn board

The Z-turn board has 2 button switches. Both switches are active low. One of these button switches is connected to the ***Reset*** pin of the Zynq 7 while the second button switch is designated as ***USER*** button and therefore one could use it in his projects.

The procedure to create a Vivado project, how to create a block design and how to include the Zynq Processing System is already covered in previous chapters. Do not forget the create a hardware wrapper before generating the bitstream file. Figure 6.1 shows a typical system.

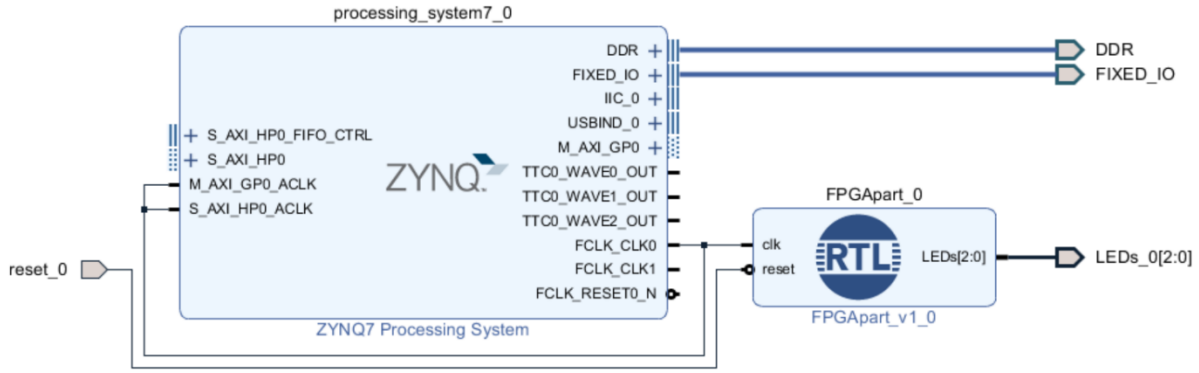


Figure 6. 1: Hardware System

The block diagram shows a separate VHDL module created in the Programmable Logic. This module is used to test whether the boot image file has been loaded in the Zynq 7 properly, so for this project, the VHDL module could be removed. The reset switch of the VHDL module is not the same reset switch mentioned in the introduction of this chapter.

Notice also that the button switch connected to MIO 50 is not seen in the diagram just like the LEDs connected to MIO 0 and MIO 9. These are default in the constraints file generated by the Board Support Files.

FIXED_IO_mio (54)	INOUT					✓	(Multiple)	(Multiple)*
FIXED_IO_mio[53]	INOUT				C11	✓	501	LVCMOS18
FIXED_IO_mio[52]	INOUT				C10	✓	501	LVCMOS18
FIXED_IO_mio[51]	INOUT				B9	✓	501	LVCMOS18
FIXED_IO_mio[50]	INOUT				B13	✓	501	LVCMOS18
FIXED_IO_mio[49]	INOUT				C12	✓	501	LVCMOS18
FIXED_IO_mio[48]	INOUT				B12	✓	501	LVCMOS18
FIXED_IO_mio[47]	INOUT				B14	✓	501	LVCMOS18
FIXED_IO_mio[46]	INOUT				D16	✓	501	LVCMOS18
FIXED_IO_mio[45]	INOUT				B15	✓	501	LVCMOS18
FIXED_IO_mio[44]	INOUT				F13	✓	501	LVCMOS18

Figure 6. 2: Part of the constraints file

Figure 6.2 shows part of the constraints file. If the VHDL module was not included in project, all that had to be done is just generate a bitstream file straight away, however since in this project, a VHDL module was also included, a new constraints file was created to accommodate the changes in the Programmable Logic part.

The ***USER*** button switch is connected to MIO 50. The pin is connected via a pull up resistor to 1V8. For this particular project, leave the default voltage of 1V8 in the IO settings in Vivado as shown in Figure

6.2. Generate a bitstream file, export the hardware including the bitstream file and Launch SDK from within the Vivado project.

In the following section, the software part and its intricacies will be discussed.

<u>The Software</u>

In SDK, the usual FSBL application has to be created as part of the project. After that create a new C project following the usual steps as described in previous chapters.

The ***gpiops_v3_3*** library will be used in this project. Therefore, include the ***xgpiops.h*** file as usual.

Initialize the MIO bank as shown in the following code. Detailed explanation of this code has already been covered in previous chapters.

```
PS_GpioConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
PSGpioStatus =XGpioPs_CfgInitialize(&PSGpio, PS_GpioConfigPtr,PS_GpioConfigPtr ->BaseAddr)
 if(PSGpioStatus != XST_SUCCESS)
 {
     return XST_FAILURE;
 }
XGpioPs_SetDirection(&PSGpio, 0,0xFFFFFFFF); // 1= output 0 = input
XGpioPs_SetOutputEnable(&PSGpio, 0,0xFFFFFFFF); //1 = enable
```

Code Snippet 6. 1: Initializing the MIO bank

Even though MIO 50 is on **bank 1**, this does not mean that any changes to the ***device-ID*** has to be made. It should remain the same ***XPAR_PS7_GPIO_0_DEVICE_ID***.

Code snippet 6.1 also shows that all MIO pins have been enabled as outputs. This is convenient to make sure that all the pins are enabled. However, it is advised to configure the pins needed as inputs separately, by using the following function:

void XGpioPs_SetDirectionPin(XGpioPs *InstancePtr, u32 Pin, u32 Direction)

where ***XGpioPs *InstancePtr*** is replaced by the name of the instance in this case ***&PSGpio, u32 Pin*** must be replaced with the pin number, in this case ***50*** (for MIO 50) and the direction should be set to ***0*** as it must be configured as *<u>input</u>*.

```
XGpioPs_SetDirectionPin(&PSGpio, 50, 0); //this sets MIO 50 to input
```

The author tried using the function where all the bank is configured with one function using the

XGpioPs_SetDirection(&PSGpio, 1,0xFFFBFFFF); // only MIO 50 is set as input in bank 1

But for some reason, it did not work!

Now, since the focus of this chapter is to learn how to use the button switch on MIO 50, the ***C*** code does a simple task to demonstrate its use. It waits for a button press then blinks both LEDs on MIO 0 and MIO 9 at the same time, once, then waits for another switch press.

To read from the port, the same concept must be adopted. Use the

u32 **XGpioPs_ReadPin**(XGpioPs *InstancePtr, u32 Pin)

function. This function returns a u32 value. ***XGpioPs *InstancePtr*** should be declared as shown in previous chapters while ***u32 Pin*** should be replaced with the pin number - in this case ***50*** because it is connected to MIO 50.

```
 sw_MIO50 = XGpioPs_ReadPin(&PSgpio, 50);
if(sw_MIO50 != 1)
{

    XGpioPs_WritePin(&PSgpio, 0, 1);
}
else
{

    XGpioPs_WritePin(&PSgpio, 0, 0);
}
```

Code Snippet 6. 2: detecting the Button Switch

The return value of the read-pin function is stored in a variable of type u32. Since the switch is active low, when the switch is idle (not pressed), the function returns a 1 because of the pull-up feature of the circuit. Once the switch is pressed, it connects MIO 50 to ground and therefore the function returns a 0. This is monitored by the ***IF*** statement. Note that the LEDs light when a logic 0 is at the output of the pin.

Processing System Dual AXI block control

In this chapter, two AXI GPIO blocks will be controlled from the Processing System. This is the maximum number of AXI GPIO blocks one can use with the Processing System because the ***xgpio_v4_4*** library functions within SDK, only cater for two AXI GPIO blocks. However, it must be said that each AXI GPIO block has a total of 64 IOs and therefore there is more than enough IOs with two AXI GPIO blocks!

In the meantime, one can use as many AXI GPIO blocks as the application needs, with the Programmable Logic part of the Zynq 7. However, again this number is limited with the number of physical IOs the Zynq 7 has, but in case of internal connections, one can add as many AXI GPIO blocks as needed. The final circuit is shown in Figure 7.1 below.

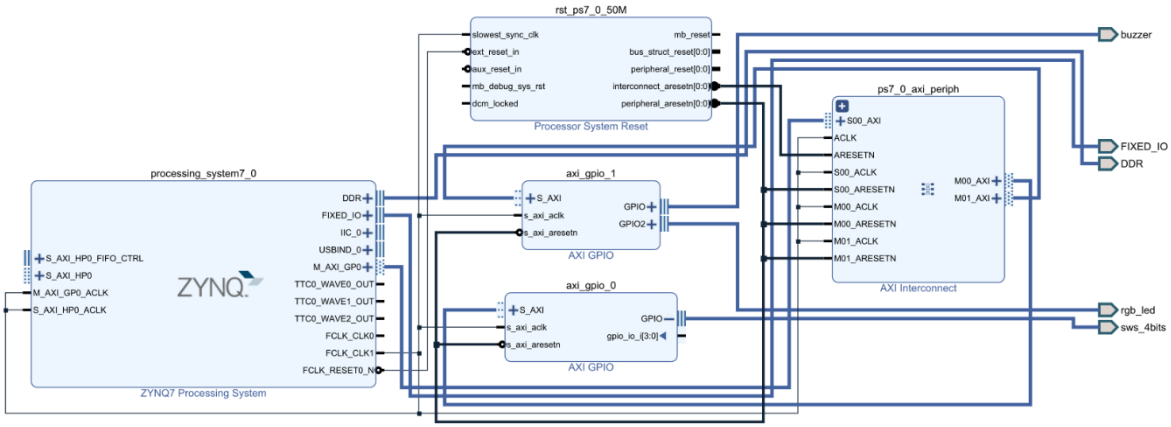


Figure 7. 1: final Block Design

The process to create a project and a block design has already been discussed in previous chapters so these will not be mentioned here anymore. Also, when using AXI GPIO blocks, one needs to include the *AXI interconnect block* and the Processor Reset Block, both shown in Figure 7.1. Again, these have been explained in previous chapters and therefore they will not be mentioned here.

One way to use two AXI GPIO blocks is to configure one of the AXI block to accept inputs while the second AXI GPIO block can be configured to be all made up of outputs. That way, one can have up to 64 inputs and up to 64 outputs! Do we need more?!

In the project described in this chapter AXI_GPIO_0 block's channel 1, is connected to the DIP switches and therefore from the 64 inputs, only 4 are used. On the other hand, AXI_GPIO_1 is configured as output block. The two available channels within the AXI block are used, channel 1 is connected to the onboard piezo buzzer while channel 2 is connected to the LEDs. Both peripherals reside on the Programmable Logic side. To implement these settings, one must go through the following steps:

1) Double Click on the AXI Block shown in Figure 7.2
2) For channel 1 within AXI_GPIO_0, click on the drop-down menu and select sws_4bits as shown in Figure 7.3
3) Now double click again on AXI_GPIO_1 and select the LEDs and buzzer as shown in Figure 7.4.

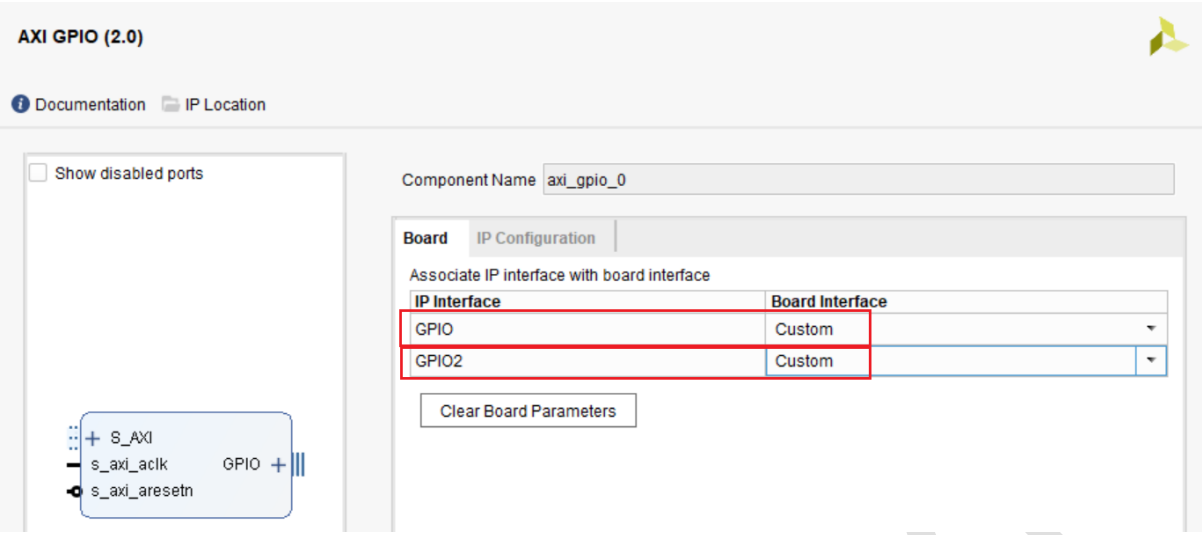


Figure 7. 2: Double Click on the AXI block

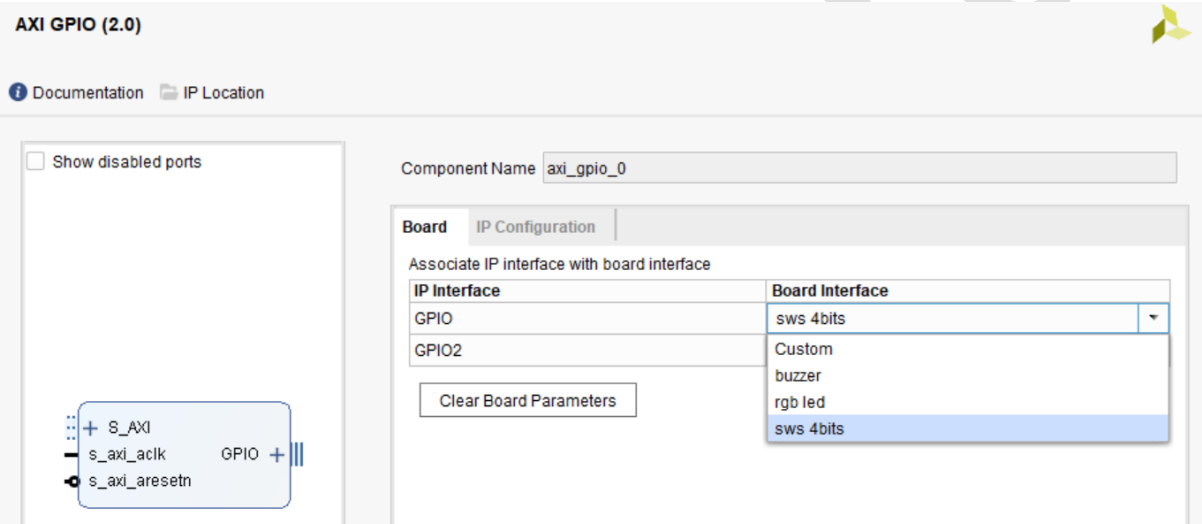


Figure 7. 3: Selecting the DIP switches

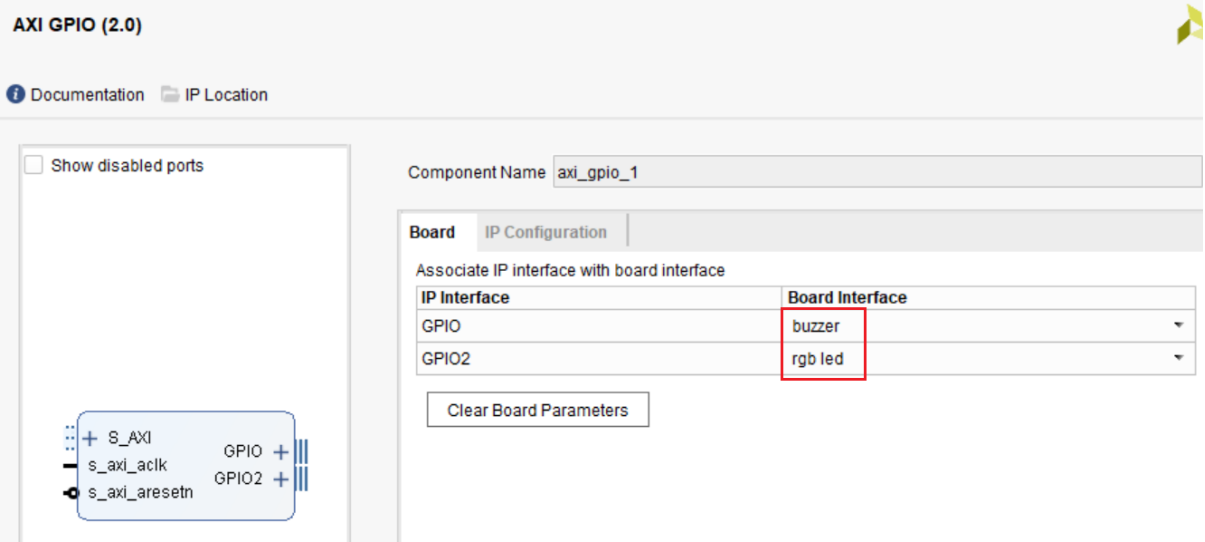


Figure 7. 4: Configuring the second AXI block

The next thing is to *assign an address* to both the AXI GPIO blocks to be memory mapped, shown in Figure 7.5 below:

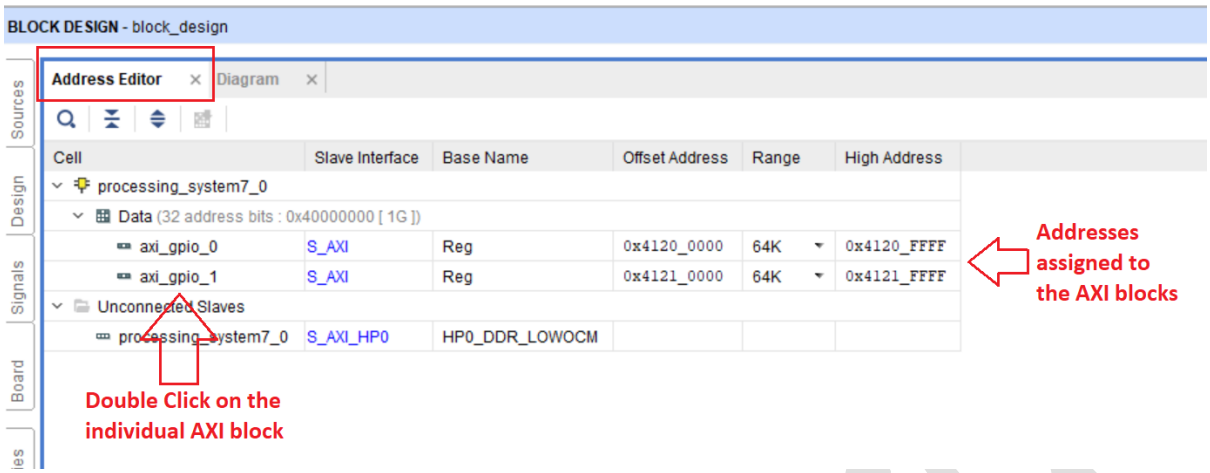


Figure 7. 5: Assigning Addresses to the AXI blocks

The above is achieved if one ***right-clicks*** on the individual AXI GPIO block, and from the menu select ***Assign Address***. The addresses will be automatically given to the respective AXI GPIO block. This step is also shown in one of the previous chapters.

Save the block design and create a Hardware Wrapper. Then synthesize the model.

Open the Synthesized design and check the assigned pinouts of the DIP switches, the piezo buzzer and the LEDs. Due to the board support files, these should have the right pin assignments.

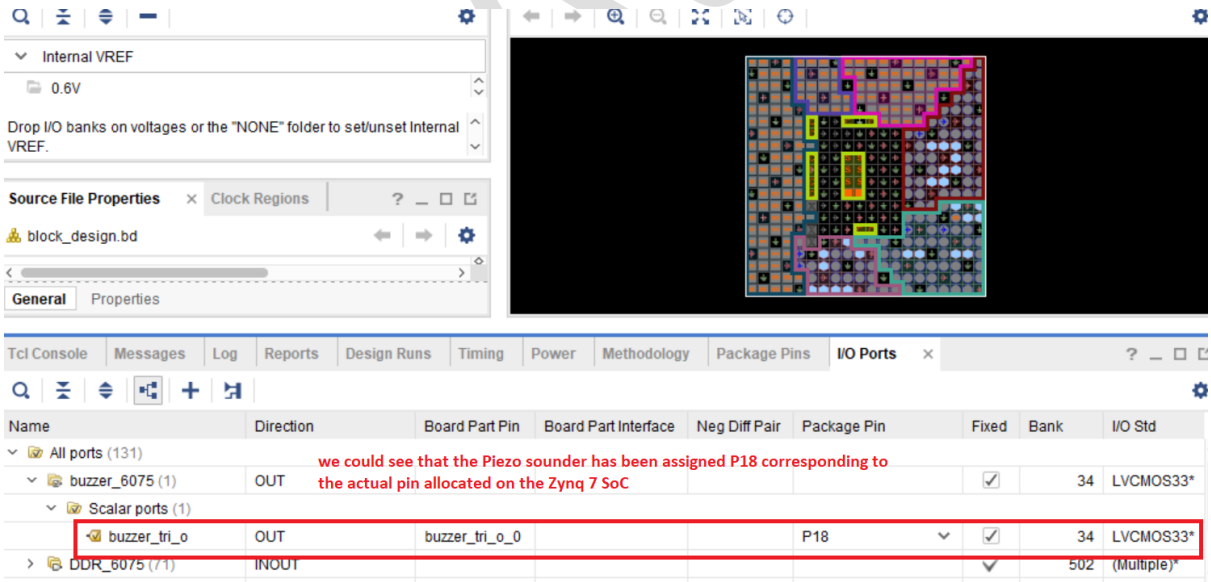


Figure 7. 6: Pin assignment of the Buzzer

Note that the working voltage is changed to 3V3 and the box under the column called *Fixed* is ticked. Also note the name of the pinout, together with the assigned pin number P18 which is taken from the Board Support Files and therefore nothing should be changed from this tab. The above also holds true for the switches and the LEDs. These are explained better in Figure 7.7 and Figure 7.8 on the next page.

sws_4bits_tri_i (4)	IN					✓	(multiple)	LVCMOS33*
sws_4bits_tri_i[3]	IN	sws_4bits_tr...	these are the pins connected to the DIP switches	J15		✓	35	LVCMOS33*
sws_4bits_tri_i[2]	IN	sws_4bits_tr...		G14		✓	35	LVCMOS33*
sws_4bits_tri_i[1]	IN	sws_4bits_tr...		T19		✓	34	LVCMOS33*
sws_4bits_tri_i[0]	IN	sws_4bits_tr...		R19		✓	34	LVCMOS33*

note the working voltage is 3V3

Figure 7. 7: Pins assigned to the DIP switches

rgb_led_tri_o (3)	OUT					✓	34	LVCMOS33*
rgb_led_tri_o[2]	OUT	rgb_led_tri_o...	here we can see the RGB LEDs pinouts	Y17		✓	34	LVCMOS33*
rgb_led_tri_o[1]	OUT	rgb_led_tri_o...		Y16		✓	34	LVCMOS33*
rgb_led_tri_o[0]	OUT	rgb_led_tri_o...		R14		✓	34	LVCMOS33*
Scalar ports (0)								

Figure 7. 8: Pinouts where the LEDs are connected

Note that the LEDs and the Buzzer cannot form part of the same channel within the same AXI GPIO block because Vivado does not allow this to happen. So, to accommodate both the RGB LEDs and the Buzzer, two channels within the same AXI GPIO block must be used.

After saving to a new constraints file, it is time to generate a bitstream file, export the hardware (including the bitstream) and launch SDK from within the project.

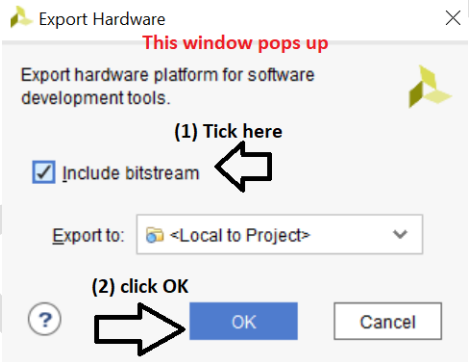


Figure 7. 9: Include the Bitstream File when exporting the hardware

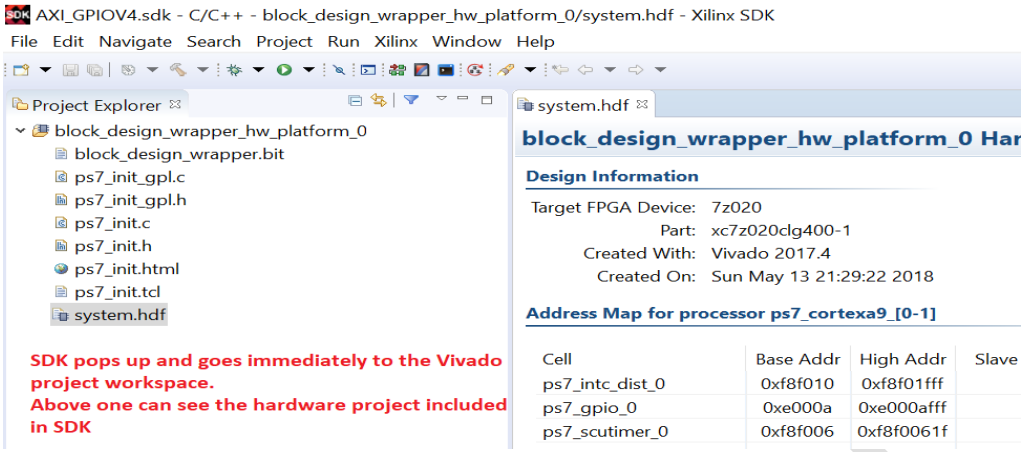


Figure 7. 10: SDK project linked to the Vivado project

Create a First Stage Boot Loader (FSBL) project, then create a C project from where control of the AXI GPIO blocks will be done. This is what will be learnt new in this chapter!

Again, choose the Hello World project from the list as shown in Figure 7.11.

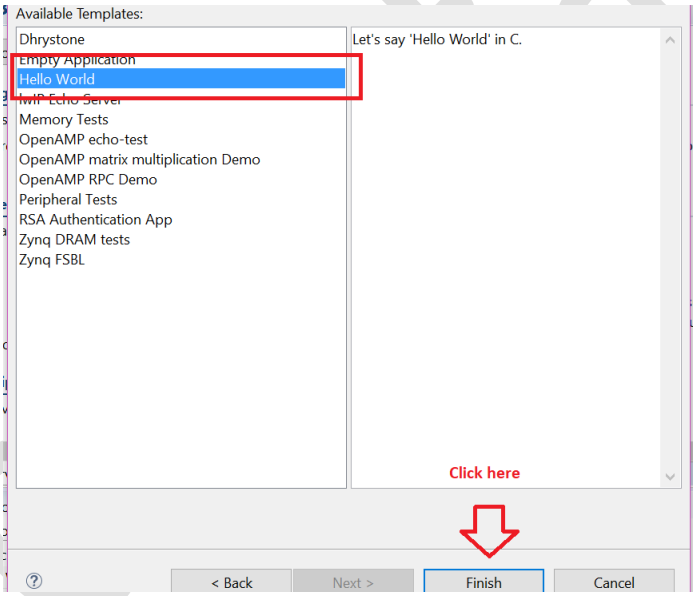


Figure 7. 11: Choosing the Hello World project from list

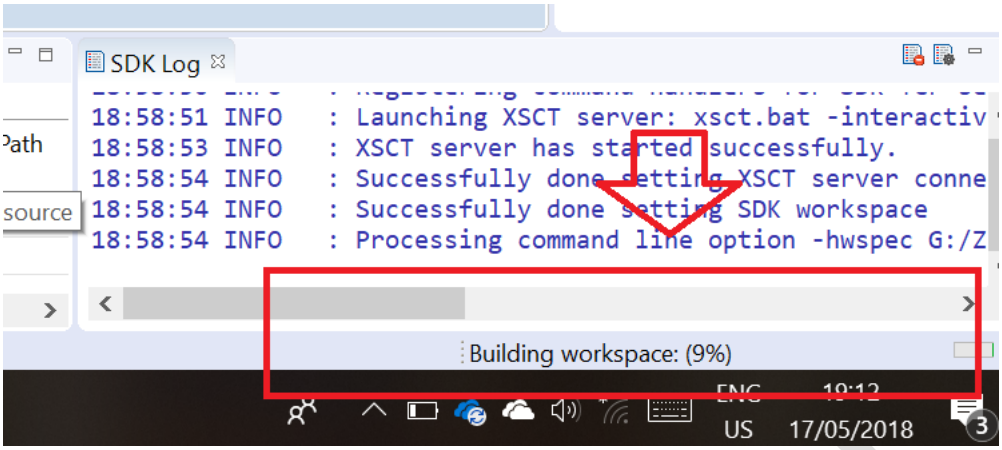


Figure 7. 12: Wait for SDK to finish compiling

Figure 7.12 is very important! One must wait for the SDK workspace to finish building! This could be checked from the bottom-right-corner of the screen.

If the UART is needed by the C application, one must modify the Board Support Package as shown in Figure 7.13.

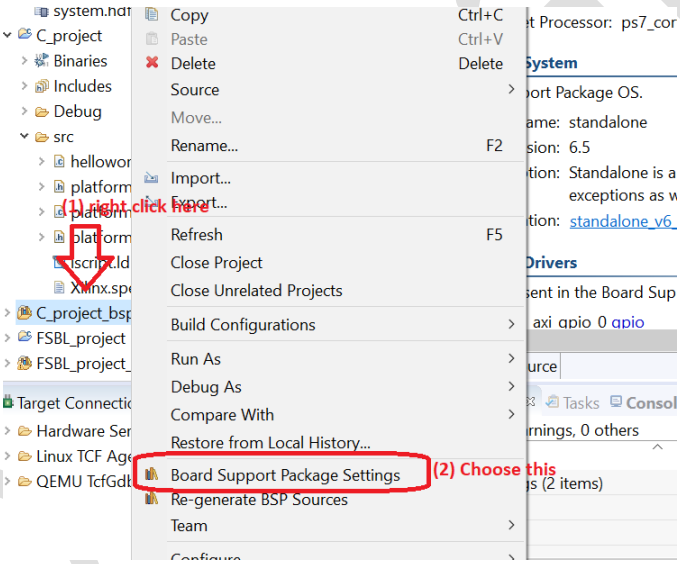


Figure 7. 13: Opening the Board Support Package

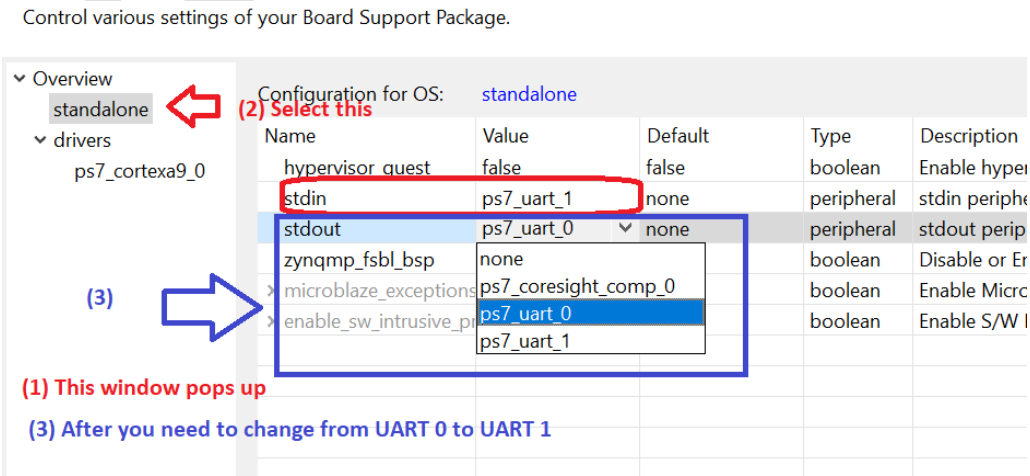


Figure 7. 14: Change to UART 1

Wait for SDK to compile the project.

The Software for this project

Open the *Hello_World.c* file located in the src folder in the C project. Include the *xgpiops.h* and *xgpio.h* header files using the #include "*xxxxxxx.h*" directive at the beginning of the C file.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xgpio.h"
```

As shown in previous chapters, now open the *gpio_v4_3* folder. This contains the library of functions that could be used with the AXI GPIO blocks. To use the MIO port, one has to open the *gpiops_v3_3* folder. Now from these two folders one can extract or copy the functions to configure both the MIO bank and the AXI GPIO blocks.

```
//initialise the PS GPIO
PS_GpioConfigPtr= XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
status = XGpioPs_CfgInitialize(&PS_Gpio, PS_GpioConfigPtr,PS_GpioConfigPtr -> BaseAddr);
if(status != XST_SUCCESS)
{
    return XST_FAILURE;
}

//set whether the PS port pins are to act as inputs or outputs 1 = output 0 = input
XGpioPs_SetDirection(&PS_Gpio,0,0x00000201); //MIO_0 & MIO_9 are set as outputs
//enable the individual pins of the same bank
XGpioPs_SetOutputEnable(&PS_Gpio, 0, 0x00000201);
```

Code Snippet 7. 1: Initializing the MIO Bank

```
//initialise the AXI GPIO 0 block
PL_GpioConfigPtr0 = XGpio_LookupConfig(XPAR_AXI_GPIO_0_DEVICE_ID);
success0 = XGpio_Initialize(&PL_Gpio0, XPAR_AXI_GPIO_0_DEVICE_ID);
if(success0 != XST_SUCCESS)
{
    return XST_FAILURE;
}

//initialize the AXI GPIO 1 block
PL_GpioConfigPtr1 = XGpio_LookupConfig(XPAR_AXI_GPIO_1_DEVICE_ID);
success1 = XGpio_Initialize(&PL_Gpio1,XPAR_AXI_GPIO_1_DEVICE_ID);
if(success1 != XST_SUCCESS)
{
    return XST_FAILURE;
}

//In this application we need both channels of AXI GPIO1 to be set as outputs
XGpio_SetDataDirection(&PL_Gpio1,1,0x00000000); //piezo connected with channel 1 of AXI 1
XGpio_SetDataDirection(&PL_Gpio1, 2,0x00000000);//LEDs are connected to channel 2 of AXI 1

//In this application we have a separate AXI block for the inputs. Only channel 1 is used
XGpio_SetDataDirection(&PL_Gpio0, 1,0x0000000F);//switches are connected to channel 1 of AXI 0
```

Code Snippet 7. 2: Initializing the two AXI blocks

Looking at *Code Snippet 7.2*, one should notice that there are two instances of AXI GPIO. These are named as *PL_Gpio0* and *PL_Gpio1*. These are the names of the AXI GPIO blocks!

Within AXI_GPIO1 block, two channels are used. These are named as channel 1 and channel 2. *Two different channels* were needed, one to drive the piezo buzzer while the second channel was used to drive the RGB LEDs. This is because the *Board Support information* was used in this project. A workaround to be more efficient with the IO pins, is to change the channel to ***custom*** when *configuring the AXI GPIO block within Vivado*, opt for *4 outputs* and these should be enough to control each LED and the buzzer. However, in this project, the main focus is to show how to control two AXI GPIO blocks from the Processing System.

All the variables and pointers used in above functions must be declared at the beginning of the main () as shown in Code Snippet 7.3.

```
int status, success0,success1,sw_positions;
XGpioPs PS_Gpio; // PS GPIO pointer pointing to the bank where the IOs are
XGpioPs_Config *PS_GpioConfigPtr; //this is the configuration pointer for the PS GPIO
XGpio PL_Gpio0;//this is the first AXI GPIO 0 pointer
XGpio PL_Gpio1;//this the second pointer for AXI GPIO 1
XGpio_Config *PL_GpioConfigPtr0; //these are the configuration pointers for
XGpio_Config *PL_GpioConfigPtr1; //both FPGA AXI blocks
```

Code Snippet 7. 3: Declaring the pointers and variables

```
while(1)
{
 print("check switches\n\r");

 //read the position of the switches and mask
 sw_positions = (XGpio_DiscreteRead(&PL_Gpio0, 1) & 0x0000000F);

 switch(sw_positions)
 {
 case 0x00000000:
         XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
         XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x0000000F); //switch off PL LEDs
         XGpioPs_Write(&PS_Gpio, 0, 0x00000201);//MIO_0 & MIO_9 are off
         break;
 case 0x00000001:
     XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000001); //switch on piezo
     XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x0000000F); //switch off PL LEDs
     XGpioPs_Write(&PS_Gpio, 0, 0x00000201);//MIO_0 & MIO_9 are off
     break;
 case 0x00000002:
     XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
     XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x00000001); //switch on PL LEDs
     XGpioPs_Write(&PS_Gpio, 0, 0x00000201);//MIO_0 & MIO_9 are off
     break;
```

```
 case 0x00000003:
        XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
        XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x00000002); //switch on PL LEDs
        XGpioPs_Write(&PS_Gpio, 0, 0x00000201);//MIO_0 & MIO_9 are off
        break;
 case 0x00000004:
        XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
        XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x00000003); //switch on PL LEDs
        XGpioPs_Write(&PS_Gpio, 0, 0x00000201);//MIO_0 & MIO_9 are off
        break;
 case 0x00000005:
        XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
        XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x00000004); //switch on PL LEDs
        XGpioPs_Write(&PS_Gpio, 0, 0x00000201);//MIO_0 & MIO_9 are off
        break;
 case 0x00000006:
        XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
        XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x00000005); //switch on PL LEDs
        XGpioPs_Write(&PS_Gpio, 0, 0x00000201);//MIO_0 & MIO_9 are off
        break;

case 0x00000007:
        XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
        XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x00000006); //switch on PL LEDs
        XGpioPs_Write(&PS_Gpio, 0, 0x00000201);//MIO_0 & MIO_9 are off
        break;
case 0x00000008:
        XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
        XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x00000007); //switch on PL LEDs
        XGpioPs_Write(&PS_Gpio, 0, 0x00000200);//MIO_0 on only
        break;
case 0x00000009:
        XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
        XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x00000007); //switch on PL LEDs
        XGpioPs_Write(&PS_Gpio, 0, 0x00000001);//MIO_9 on only
        break;
case 0x0000000A:
            XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000001); //switch on piezo
            XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x00000000); //switch on PL LEDs
            XGpioPs_Write(&PS_Gpio, 0, 0x00000200);//MIO_0 on only
            break;
case 0x0000000B:
            XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
            XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x0000000E); //switch on PL LEDs
            XGpioPs_Write(&PS_Gpio, 0, 0x00000001);//MIO_0 and MIO_9 on only
            break;
```

```
case 0x0000000C:
                XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000001); //switch off piezo
                XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x0000000E); //switch on PL LEDs
                XGpioPs_Write(&PS_Gpio, 0, 0x00000200);//MIO_0 and MIO_9 on only
                break;
case 0x0000000D:
                XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
                XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x0000000C); //switch on PL LEDs
                XGpioPs_Write(&PS_Gpio, 0, 0x00000001);//MIO_0 and MIO_9 on only
                break;
case 0x0000000E:
                XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
                XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x00000003); //switch on PL LEDs
                XGpioPs_Write(&PS_Gpio, 0, 0x00000200);//MIO_0 and MIO_9 on only
                break;
case 0x0000000F:
                XGpio_DiscreteWrite(&PL_Gpio1, 1, 0x00000000); //switch off piezo
                XGpio_DiscreteWrite(&PL_Gpio1, 2, 0x00000000); //switch on PL LEDs
                XGpioPs_Write(&PS_Gpio, 0, 0x00000000);//MIO_0 and MIO_9 on only
                break;
```

Code Snippet 7. 4: Main Code

Code Snippet 7.4 detects the positions of the 4 DIP switches and according to their relative positions, a combination of RGB LEDs and the on-board buzzer are used to determine how the DIP switches are placed. A switch-case statement is used this time because it is more efficient than the if-else statement for this application.

All that is needed now is to save the .C file and this will start compilation. If no errors are found, then create a Boot image file, copy it to SD card, transfer the SD card to the Z-turn board. Power the Z-turn board, and change the DIP switches' positions repeatedly and notice the combination of the RGB LEDs together with the Buzzer. Enjoy!

XADC – The Analogue to Digital Converter Block within the 7 Series FPGAs and the Zynq 7

ADC stands for Analogue to Digital Conversion. This means that an analogue voltage is sampled, and this analogue voltage is represented by a decimal number. The decimal number is not infinite because it is restricted by the number of bits. Most common microcontrollers have 10-bit ADC peripherals however the Zynq 7 has a 12 -bit ADC and therefore the input voltage can be represented by a decimal number from 0 to 4095. The recommended reference voltage by Xilinx for the analogue inputs is 1.25V, however on the Z-turn board, the external reference voltage XADC_VCC is 1.8V obtained via an inductor which suppresses further any noise on XADC_VCC supply rail. Strictly speaking, this means that the analogue signal applied to any one of the analogue inputs should not exceed 1.8V! However, it is recommended by the author, not to exceed the analogue input voltage by more than 1V.

The Zynq 7 SoC has internal parameters that could be sampled via the internal XADC. These might need to be monitored if the Zynq 7 is part of a critical system to make sure that the Zynq 7 is operating within its parameters. These internal parameters include but is not limited to die-temperature, Auxiliary Vcc which is the reference voltage for the auxiliary XADC channels, etc, etc. One can check Xilinx UG480 for more information on this. Therefore, as a first XADC project, it would be ideal to check these internal parameters, and in subsequent projects, one will endeavour to explain more complex projects using XADC block.

Apart from checking the internal parameters, the Zynq 7 has one 12-bit channel which is able to sample at an impressive rate of 1 Mega Sample Per Second (1MSPS). This is called Vp_0/Vn_0 in the Xilinx literature such as the UG480, however in the Z-turn board's cape IO schematic, these are marked as XADC_INP0 and XADC_INN respectively. Using the board support files, one does not have to worry about the pinouts because they will be automatically assigned by Vivado, however one has to be careful when reading the schematics by MYIR because they are a bit confusing! In fact as a reference point, one should check the orientation of the 1.8V from the header pins and also the 3V3 header pin. This check will help to determine the orientation of Vp_0 / Vn_0 relative to the schematics offered by MYIR.

One other thing that might confuse novices is that Vp and Vn are referred to as two separate channels. This depends on how one sees it, because if the a differential voltage is applied between Vp and Vn, then one might still regard it as a single channel. On the other hand, most of the analogue voltages such as those from analogue sensors are referenced to ground (0V) and therefore Vn must be connected directly to analogue 0V of the signal which might also be the same ground of the digital system….and again the channel would be regarded as single as well!

Apart from this dedicated external 1MSPS channel, the Zynq 7 has another 16 analogue to digital channels referred to as auxiliary channels. In UG480, these are named as VAUXP[0] and VAUXN[0] for channel 0, VAUXP[1] / VAUXN[1] for channel 1, etc. The analogue pins are shared with digital pins and therefore these particular pins are multi-functional. They are also differential type and therefore one can apply two separate voltages on *AuxVp_**n*** / *AuxVn_**n***, and the internal op-amp will do the subtraction between the input voltages. The second ***n*** (highlighted in ***italic-bold***) present the ***number*** of the auxiliary analogue input pair. These channels sample at a maximum rate of 250 kilo-Samples per Second, and they are four times slower

than the dedicated ADC input. Having said that, it still has a relatively high sampling rate. From Chapter 9 onwards, this book will discuss the external analogue channels in more detail, but for now let's move on to see how to sample the internal parameters of the Zynq 7.

Sampling Zynq 7 internal parameters

In this project, the internal parameters of the Zynq 7 are sampled and read from the Processing System. These are then output on LEDs to validate the system's operation by making sure that the digital result is changing with every change in the input analogue voltage. For this test, the dev-board designed by the author was used on top of the Cape IO board by MYIR. This dev-baord was created to compliment the Z-turn board. It has two 10KΩ preset-potentiometers and two 5kΩ preset-pots, together with four more DIP switches, four push-to-make switches, a single seven-segment display and a set of 18 SMT LEDs. All of these components are completely isolated from the Zynq 7.

As always, start by creating a new Vivado project. For this project there is no need to create a VHDL file and therefore skip those steps. When Vivado IDE is opened on the project, one can create a block design and add the Zynq Processing System. Since the system is going to output the decimal equivalent of the analogue input signal on LEDs, one can include the AXI interconnect block and AXI GPIO block as part of the schematics.

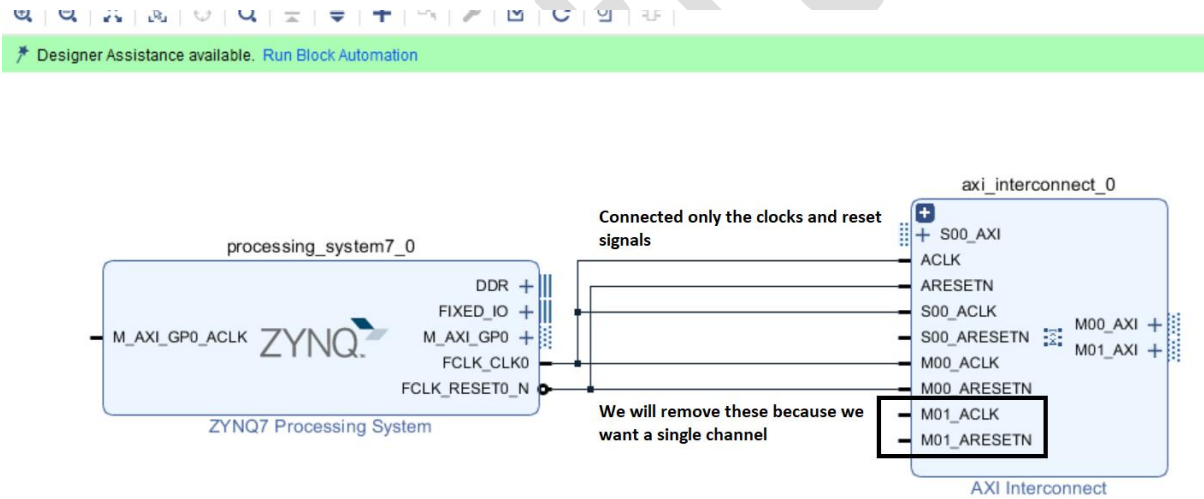


Figure 8. 1: configure the AXI interconnect block

The AXI interconnect block is configured to accept two AXI slave blocks (even though they are called M01…see Figure 8.1), in this project only one is needed so double click on the AXI interconnect block and the configuration window pops up shown in Figure 8.2.

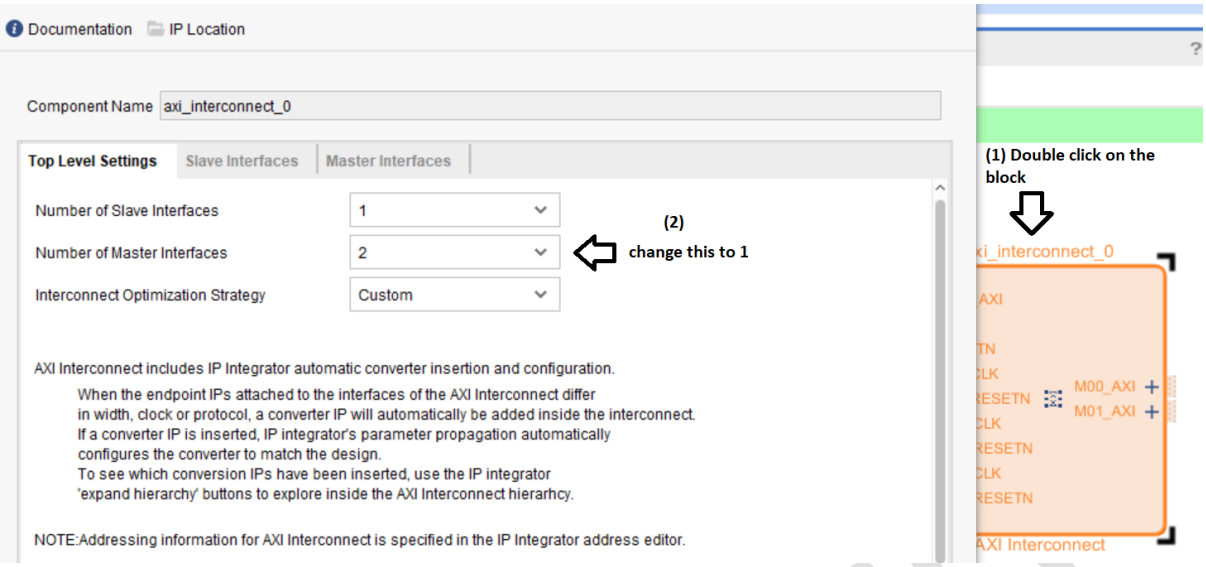


Figure 8. 2: AXI interconnect config window

Change the number of Master interfaces to 1 and leave the page as it is. Click on OK. This is how the diagram looks like now:

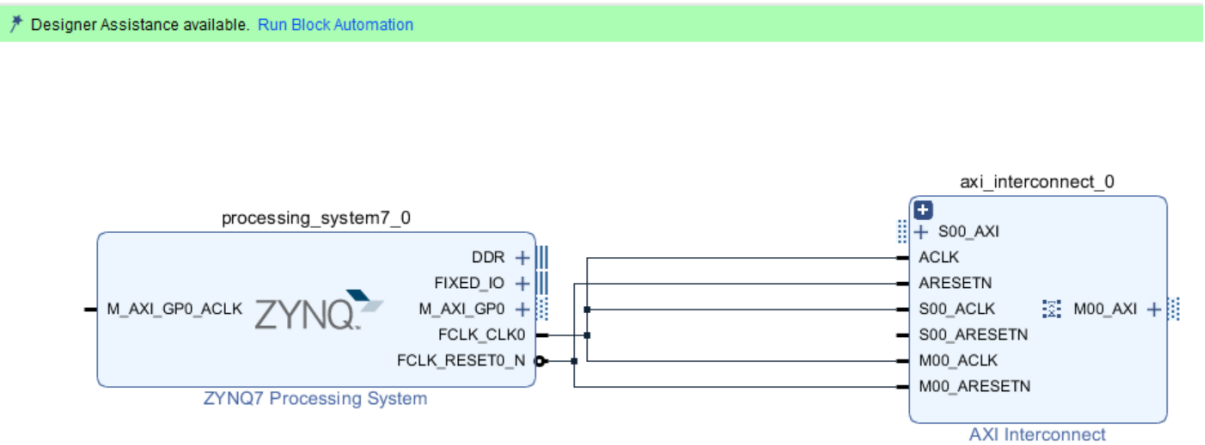


Figure 8. 3 Zynq 7 connected to AXI interconnect

then include the XADC block:

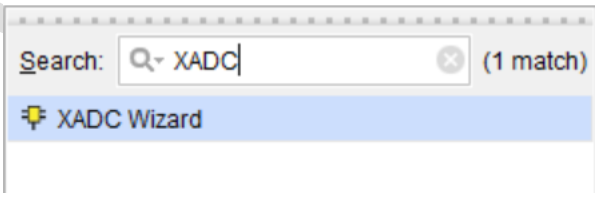


Figure 8. 4: Adding the XADC block

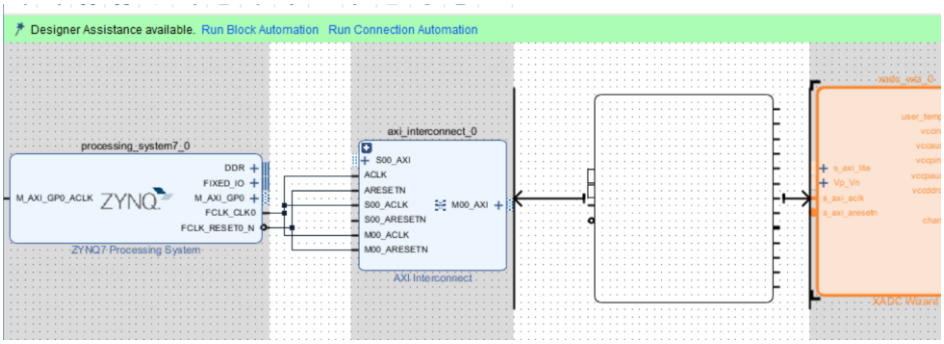


Figure 8. 5: blocks in the block design can be moved

Figure 8.5 shows that blocks can be moved even though with some restrictions. One can left click on the object block and hover the mouse over the canvas for the object block to move along with the mouse.

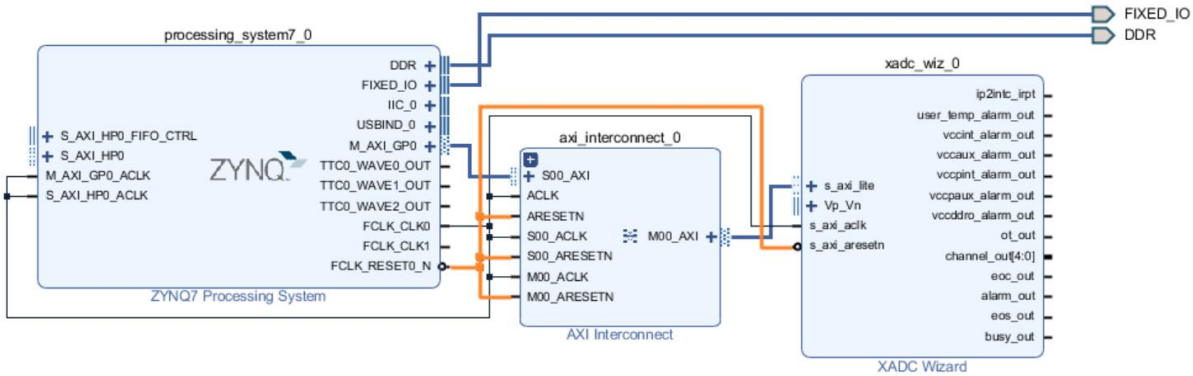


Figure 8. 6: connect the resets

Connect the reset pins of all the blocks together. A better solution is to use the Processing System Reset block as shown in previous chapters.

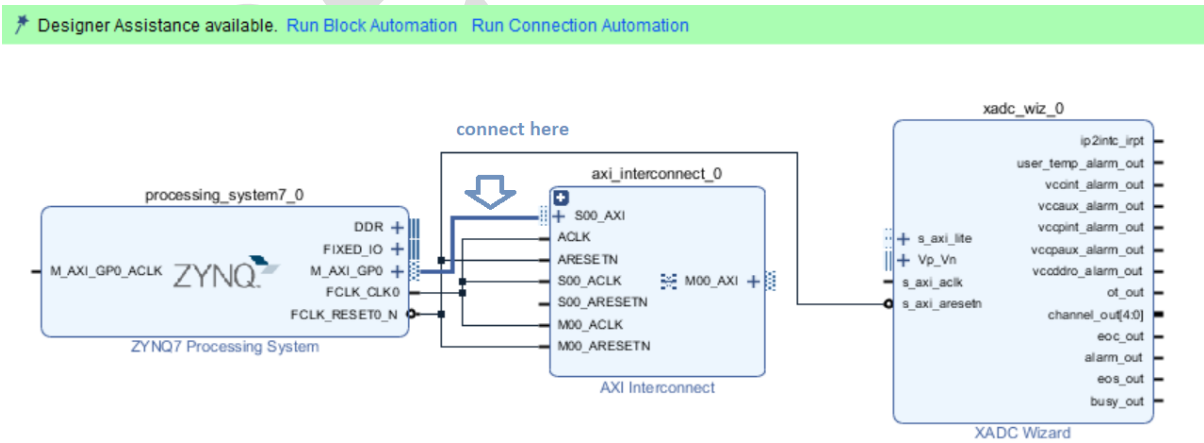


Figure 8. 7: Connect the data bus 1

Figure 8.7 show how the data from the Processing System is connected to the AXI interconnect block.

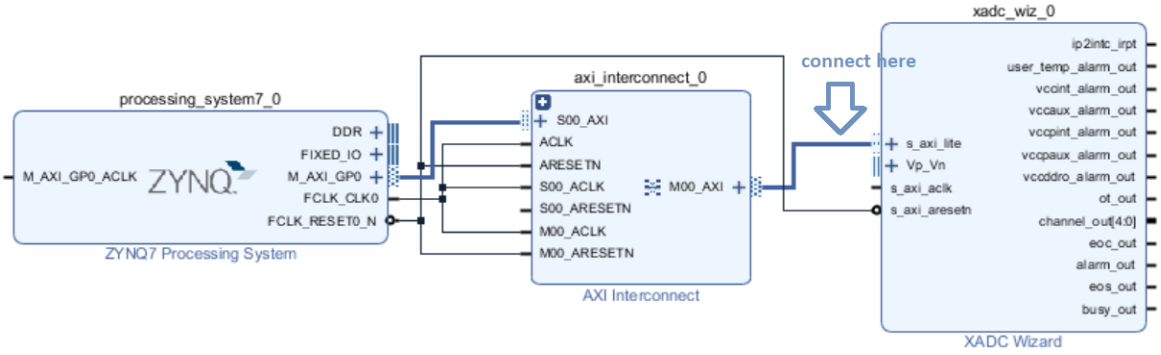


Figure 8. 8: Connect data bus 2

Figure 8.8 shows how the interconnect block is connected to the XADC block.

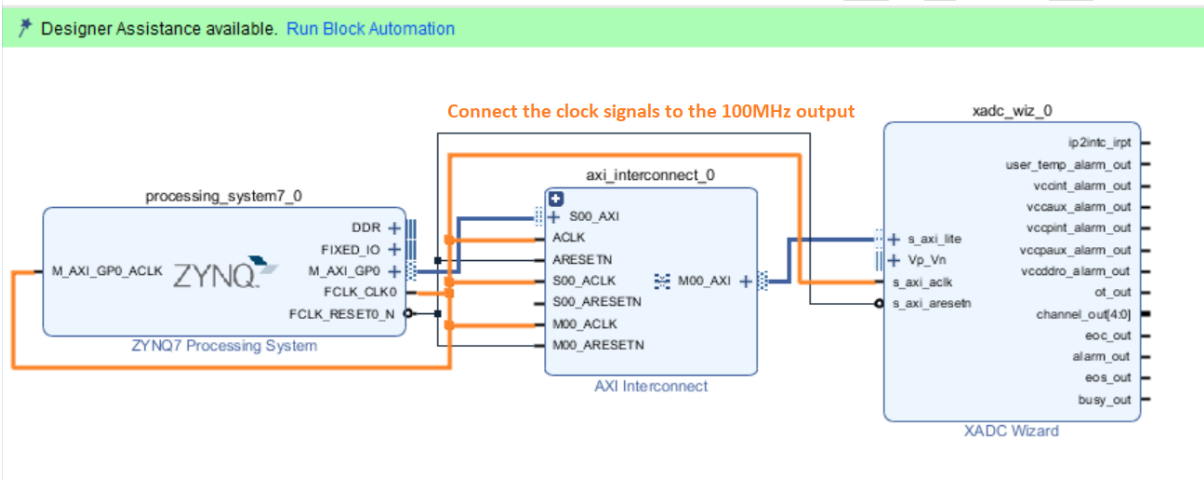


Figure 8. 9: Connecting the 100 MHz clock

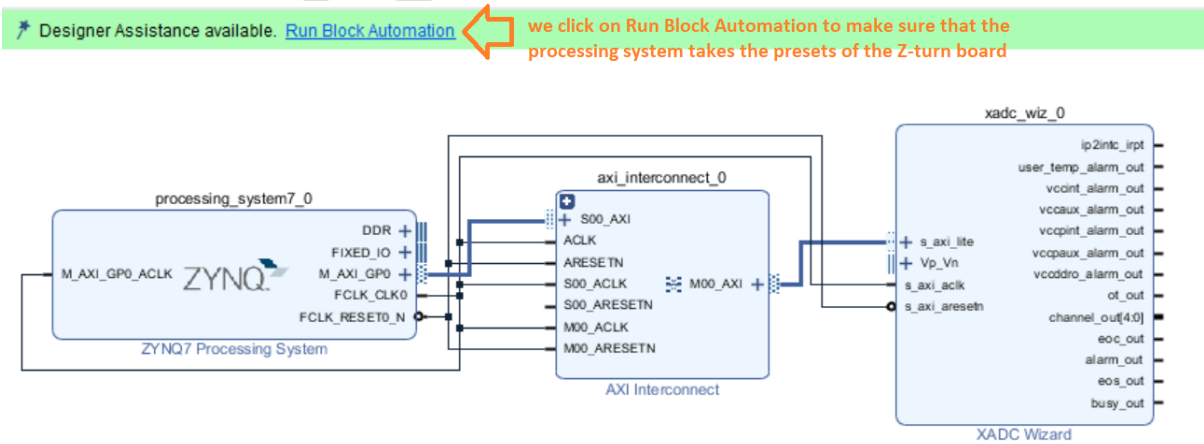


Figure 8. 10: Run Block Automation

Click on *Run Connection Automation* so that the Zynq Processing System takes the pre-set configuration assigned in the Board Support Files.

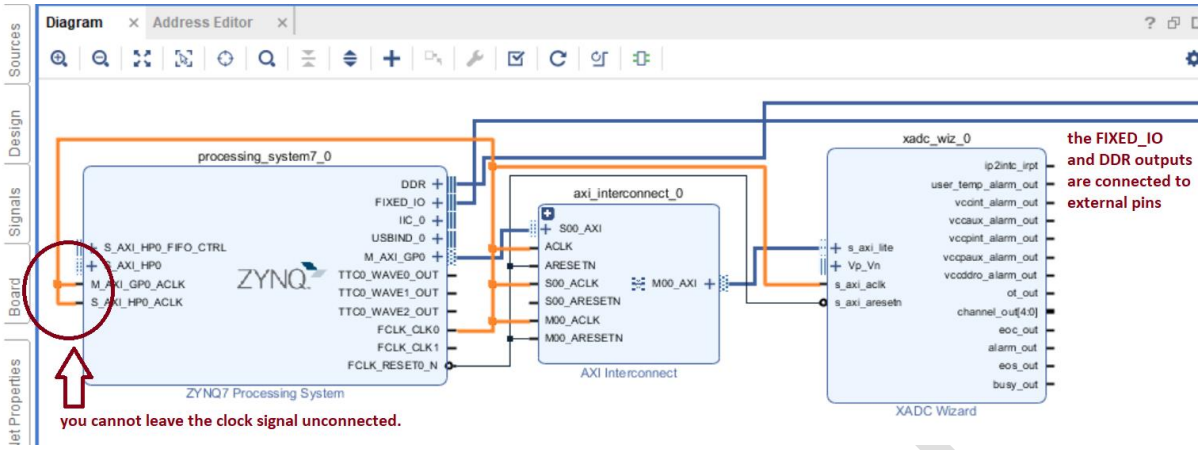


Figure 8. 11: Full Block diagram of the Zynq Processing System

Note that in Figure 8.11 the M_AXI_GP0_ACLK and S_AXI_HP0_ACLK are connected. These two clocks must be connected to the same clock signal as the whole system otherwise Vivado will generate and error.

Configuring the XADC to sample internal parameters

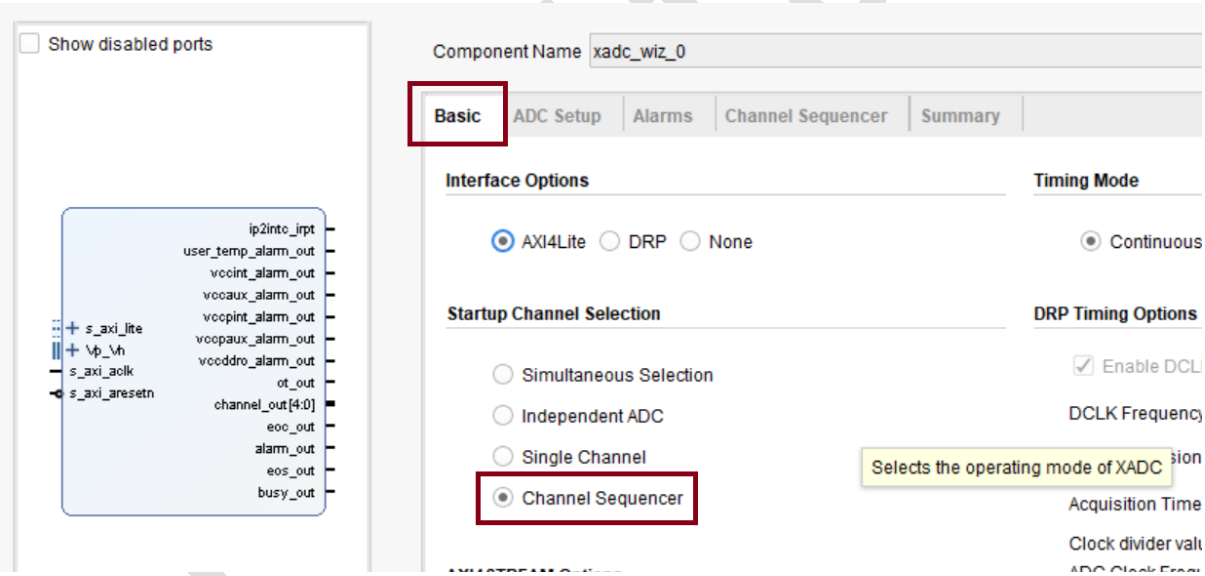


Figure 8. 12: XADC basic tab

Double-click on XADC wizard and **change** the setting from ***single channel*** to ***channel sequencer***. The channel sequencer is selected so that the XADC block will hop from one parameter to the next. The ADC result will be stored in respective status registers and therefore when one would like an ADC result of a particular parameter, the latest result will be retrieved.

Note at this point that the result resides between bit 4 and bit 16 of the status register and therefore this must be shifted to the right by four places to obtain the right binary weights of the bits.

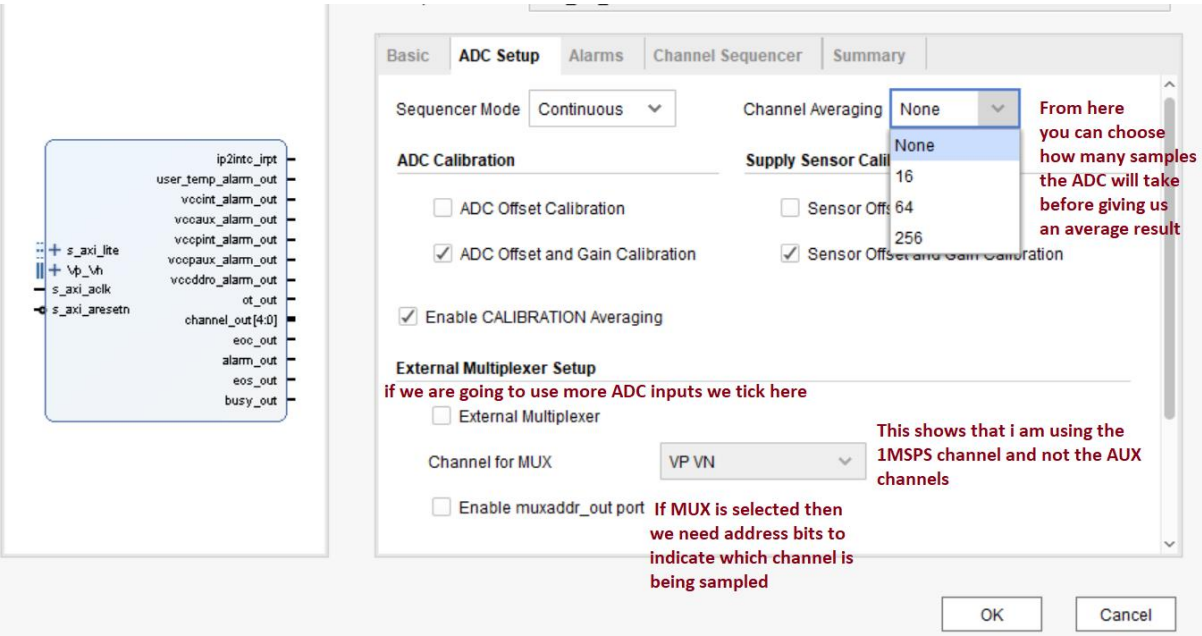


Figure 8. 13: ADC setup tab

For the ADC setup page, it would be a good idea to configure the XADC block to sample at least 16 times before the result is available to the user. Leave the rest of the settings as they are.

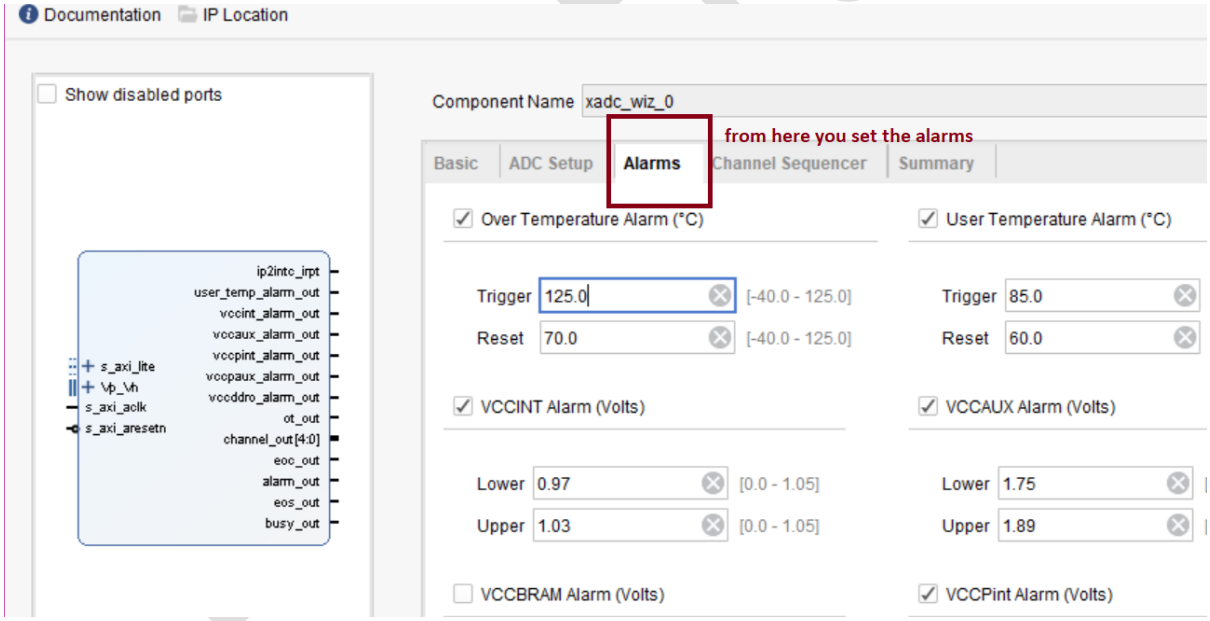


Figure 8. 14: ADC Alarms Tab

The Alarms could be switched off my removing the tick from the boxes.

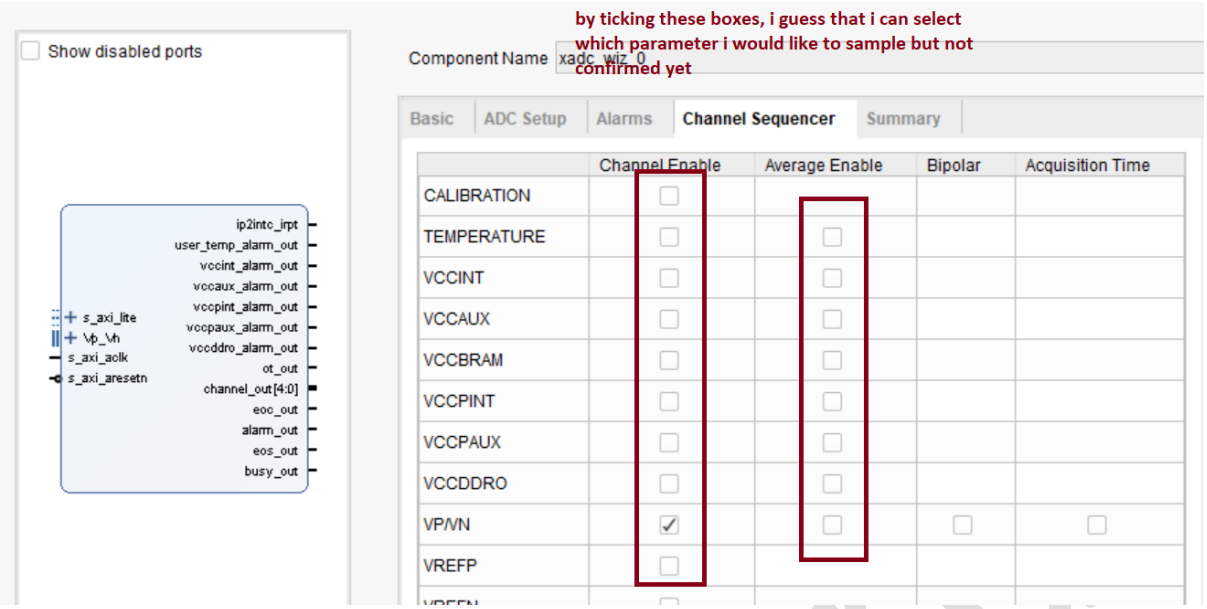


Figure 8. 15: ADC Channel Sequencer Tab

Figure 8.15 shows a list of ADC channels that one can include in the sequencer settings. At that time the author was just testing so to correct the statements done in Figure 8.15, the internal parameters that must be sampled, could be ticked in the list and these will be enabled. However, it must also be said that even though in Figure 8.15 the internal parameters were not selected, they could still be sampled from the processing system.

Create a hardware wrapper.

After the hardware wrapper, the hardware is exported by File→ Export → Export Hardware. Then launch SDK from the ***File menu***.

The Software

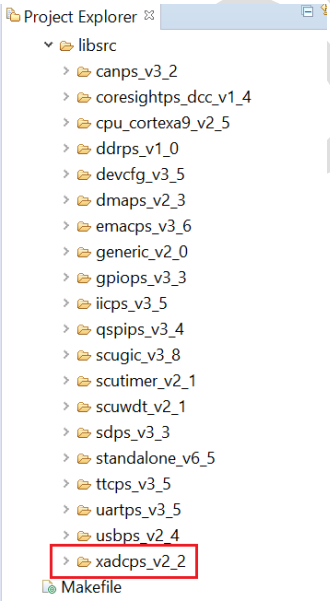


Figure 8. 16: List of libraries in SDK

After launching SDK, do not forget the create a FSBL application and a Hello World application. The XADC library is located at the very bottom of the list of folders as shown Figure 8.16. Double click on it to find all the support files for this XADC block.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xadcps.h"
#include "xil_types.h"
```

include these two header files

Figure 8. 17: Include Directives

Initialize the XADC peripheral using the following instructions:

From: ***xadcps_sinit.c*** get the lookup function:

XAdcPs_Config *XAdcPs_LookupConfig(u16 DeviceId)

This function returns an ***XAdcPS_Config*** type and therefore one must declare it at the beginning of the main function.

XAdcPs_Config *XADC_ConfigPtr;

The ***u16 DeviceId*** is listed in ***xadcps_g.c*** and in the parameter-list it should be replaced by: ***XPAR_PS7_XADC_0_DEVICE_ID***. So, the complete statement should look like:

XADC_ConfigPtr=XAdcPs_LookupConfig(XPAR_PS7_XADC_0_DEVICE_ID);

Another part of the initialization process is the configuration function that is found in ***xadcps.c*** file. This looks like this:

```
int XAdcPs_CfgInitialize(XAdcPs *InstancePtr, XAdcPs_Config *ConfigPtr,
                         u32 EffectiveAddr)
```

therefore, it returns an integer type and one must replace the instance pointer with ***&XAdcPs***-instance-pointer-name. The above statement is written as:

XADCstatus=XAdcPs_CfgInitialize(&XADCblock,XADC_ConfigPtr,XADC_Config Ptr->BaseAddress);

To avoid unnecessary warnings, check whether the configuration initialization was successful or not by:

```
If (XADCstatus!= XST_SUCCESS)
    {
        return XST_FAILURE;
    }
```

The statement-listing up till now is:

```
main()

int XADCstatus;
//this variable will hold the return value of the XAdcPs_cfgInitialize()

XAdcPs XADCblock; //this is the XAdcPs-instance name

XAdcPs_Config *XADC_ConfigPtr; //XAdc_config pointer
init_platform(); //initialize according to Z-turn board parameters

 XADC_ConfigPtr=XAdcPs_LookupConfig(XPAR_PS7_XADC_0_DEVICE_ID);
 /* get the name of the XADC block*/

 /* configure the XADC peripheral */
 XADCstatus = XAdcPs_CfgInitialize(&XADCblock, XADC_ConfigPtr,XADC_ConfigPtr->BaseAddress)
 if(XADCstatus!= XST_SUCCESS)
 {
     return XST_FAILURE;
 }
```

Code Snippet 8. 1: Initializing the XADC block in the C application

There are some statements that are unique to the XADC initialization. This is one of them:

The **self-test function** will check whether there are any problems with the XADC by resetting the device, then writes a value in the Alarm Threshold Register, then resets the device again. This function is found in ***xadcps_selftest.c*** and returns a value of type ***int***.

int XAdcPs_SelfTest(XAdcPs *InstancePtr);

selfteststatus = XAdcPs_SelfTest(&XADCblock);

selfteststatus is a variable of type ***int*** that stores the returned value of the self-test function.

if (selfteststatus != XST_SUCCESS)

{

return XST_FAILURE;

}

this will suppress the warning that we are not using the *int* variable

Next, stop the channel sequencer by selecting the mode to be as single channel mode. The function resides in ***xadcps.c*** and is:

void XAdcPs_SetSequencerMode(XAdcPs *InstancePtr, u8 SequencerMode)

To select a <u>*mode*</u> for the parameter ***u8 SequencerMode***, there is a list in ***xadcps.h*** file.

#define XADCPS_SEQ_MODE_SINGCHAN 3 /< Single channel -No Sequencing */**

```
*   #define XADCPS_SEQ_MODE_SAFE        0  < Default Safe Mode >
    #define XADCPS_SEQ_MODE_ONEPASS      1  < Onepass through Sequencer >
    #define XADCPS_SEQ_MODE_CONTINPASS 2  < Continuous Cycling Sequencer >
    #define XADCPS_SEQ_MODE_SINGCHAN   3  < Single channel -No Sequencing >
    #define XADCPS_SEQ_MODE_SIMUL_SAMPLING 4  < Simultaneous sampling>
    #define XADCPS_SEQ_MODE_INDEPENDENT    8  < Independent mode > */
```

Code Snippet 8. 2: Definintion statements found in xadcps.h file

The next thing is to disable the alarms by using the function

void XAdcPs_SetAlarmEnables(XAdcPs *InstancePtr, u16 AlmEnableMask)

found in ***xadcps.c*** file.

XAdcPs_SetAlarmEnables(&XADCblock, 0x0000); //0 = disables alarm ; 1 = enables alarm

Restart the sequencer and make it sample internal parameters such as Zynq 7 temperature, VCCINT etc.

XAdcPs_SetSequencerMode(&XADCblock, XADCPS_SEQ_MODE_SAFE);

Select the channels to sample:

The function is found in ***xadcps.c*** file.

int XAdcPs_SetSeqChEnables(XAdcPs *InstancePtr, u32 ChEnableMask)

and the parameters are found in ***xadcps_hw.h*** file.

Important Note:

Each function carries with it a description and in the description, there will be a note indicating from where the parameters can be selected giving the name of the header file and also how the parameters are named. An example follows:

```
/*****************************************************************************/
/**
*
* This function enables the specified channels in the ADC Channel Selection
* Sequencer Registers. The sequencer must be disabled before writing to these
* regsiters.
*
* @param    InstancePtr is a pointer to the XAdcPs instance.
* @param    ChEnableMask is the bit mask of all the channels to be enabled.
*       Use XADCPS_SEQ_CH__* defined in xadcps_hw.h to specify the Channel
*       numbers. Bit masks of 1 will be enabled and bit mask of 0 will
*       be disabled.
*       The ChEnableMask is a 32 bit mask that is written to the two
*       16 bit ADC Channel Selection Sequencer Registers.
*
* @return
*       - XST_SUCCESS if the given values were written successfully to
*       the ADC Channel Selection Sequencer Registers.
*       - XST_FAILURE if the channel sequencer is enabled.
*
* @note     None
```

There are some ready-made ***macros*** that one can use that convert raw ADC data into the quantity one would like to measure such as the internal temperature of the processor etc. These macros are found in ***xadcps.c*** file.

u16 XAdcPs_GetAdcData(XAdcPs *InstancePtr, u8 Channel);

```
/*****************************************************************************/
/**
*
* Get the ADC converted data for the specified channel.
*
* @param    InstancePtr is a pointer to the XAdcPs instance.
* @param    Channel is the channel number. Use the XADCPS_CH_* defined in
*       the file xadcps.h.
*       The valid channels are
*       - 0 to 6
*       - 13 to 31
*
* @return   A 16-bit value representing the ADC converted data for the
*       specified channel. The XADC Monitor/ADC device guarantees
*       a 10 bit resolution for the ADC converted data and data is the
*       10 MSB bits of the 16 data read from the device.
*
* @note     The channels 7,8,9 are used for calibration of the device and
*       hence there is no associated data with this channel.
*
*****************************************************************************/
u16 XAdcPs_GetAdcData(XAdcPs *InstancePtr, u8 Channel)
```

Code Snippet 8. 3: ADC result function

Declare a variable of type ***16 bit unsigned (u16)*** at the beginning of the main function. Then use it to store the return variable of the ***XAdcPs_GetAdcData()***.

The returned data from this function is just a decimal number from 0 to 4095 since it is 12 bits wide. This number must be processed again to reflect the analogue quantity it is monitoring. There are two macros that use 32-bit unsigned numbers, and therefore the returned number from the above function will be stored in a 32-bit number not in a u16 data type!!

The instance pointer is always the one declared somewhere above:

&XADCblock and the ***u8 Channel*** is taken from ***xadcps.h*** Underneath there is a list of u8 channels:

```
/************************** Constant Definitions ***************************/


/**
 * @name Indexes for the different channels.
 * @{
 */
#define XADCPS_CH_TEMP        0x0  /**< On Chip Temperature */
#define XADCPS_CH_VCCINT      0x1  /**< VCCINT */
#define XADCPS_CH_VCCAUX      0x2  /**< VCCAUX */
#define XADCPS_CH_VPVN        0x3  /**< VP/VN Dedicated analog inputs */
#define XADCPS_CH_VREFP       0x4  /**< VREFP */
#define XADCPS_CH_VREFN       0x5  /**< VREFN */
#define XADCPS_CH_VBRAM       0x6  /**< On-chip VBRAM Data Reg, 7 series */
#define XADCPS_CH_SUPPLY_CALIB  0x07 /**< Supply Calib Data Reg */
#define XADCPS_CH_ADC_CALIB 0x08 /**< ADC Offset Channel Reg */
#define XADCPS_CH_GAINERR_CALIB 0x09 /**< Gain Error Channel Reg  */
#define XADCPS_CH_VCCPINT   0x0D /**< On-chip PS VCCPINT Channel , Zynq */
#define XADCPS_CH_VCCPAUX   0x0E /**< On-chip PS VCCPAUX Channel , Zynq */
#define XADCPS_CH_VCCPDRO   0x0F /**< On-chip PS VCCPDRO Channel , Zynq */
#define XADCPS_CH_AUX_MIN    16 /**< Channel number for 1st Aux Channel */
#define XADCPS_CH_AUX_MAX    31 /**< Channel number for Last Aux channel */
```

Code Snippet 8. 4: ADC Channel List

The macro that converts the 12-bit ADC data into temperature resides in ***xadcps.h***. From the comments that accompany it, it was determined that it returns a ***float*** type.

```
* This macro converts XADC Raw Data to Temperature(centigrades).
*
* @param    AdcData is the Raw ADC Data from XADC.
*
* @return   The Temperature in centigrades.
*
* @note     C-Style signature:
*       float XAdcPs_RawToTemperature(u32 AdcData);  //returns a float data type
*
*****************************************************************************/
#define XAdcPs_RawToTemperature(AdcData)                    \
    ((((float)(AdcData)/65536.0f)/0.00198421639f ) - 273.15f)

/*****************************************************************************/
```

Code Snippet 8. 5: Raw to Temperature Macro

Note that AdcData is of type ***u32*** while the returned data from the previous function XAdcGetAdcData() is of type u16. However, this might not impose a problem since the data will be stored in the correct word position within the 32-bit word.

Because the above macro accepts ***u32*** data, the ***print()*** already written in the hello-world program had to be changed to ***printf()*** because the 32-bit raw data is not supported in the print() function. On the other-hand, the ***printf()*** supports ***%lu*** which means ***unsigned long data*** type.

The whole software program is listed below:

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xadcps.h"
#include "xil_types.h"


void delay (void);

int main()
{
    int XADCstatus;
    //this variable will hold the return value of the XAdcPs_cfgInitialize()
    int selfteststatus;
    // this varialbe holds the return value of the self test function
    int SeqEnable;
    /*this variable holds the return value for the seq_enables ()*/

    XAdcPs XADCblock; //this is the XAdcPs-instance name

    XAdcPs_Config *XADC_ConfigPtr; //XAdc_config pointer
    init_platform(); //initialize according to Z-turn board parameters

    u32 rawCPU_temp,rawintVCC,rawBRAMvoltage,rawAUXVCC;
    float CPUtemp,intVCC,BRAMvoltage,AUX_VCC;

XADC_ConfigPtr=XAdcPs_LookupConfig(XPAR_PS7_XADC_0_DEVICE_ID);
/* get the name of the XADC block*/

/* configure the XADC peripheral */
XADCstatus = XAdcPs_CfgInitialize(&XADCblock, XADC_ConfigPtr,XADC_ConfigPtr->BaseAddress);
if(XADCstatus!= XST_SUCCESS)
{
    return XST_FAILURE;
}

/*Run a self test by resetting the device, write a name in the Alarm Threshold
 * Register and resetting the device again*/
selfteststatus =  XAdcPs_SelfTest(&XADCblock);
if(selfteststatus != XST_SUCCESS)
{
    return XST_FAILURE;
}
```

```
while(1)
     {
      rawCPU_temp = XAdcPs_GetAdcData(&XADCblock, XADCPS_CH_TEMP);
      CPUtemp = XAdcPs_RawToTemperature(rawCPU_temp);
      printf("raw temp data: %lu while temp in degree Celcius:  %f\n\r",rawCPU_temp,CPUtemp);

      rawintVCC = XAdcPs_GetAdcData(&XADCblock, XADCPS_CH_VCCINT);
      intVCC = XAdcPs_RawToVoltage(rawintVCC);
      printf("raw intVCC data: %lu while internal VCC: %f\n\r",rawintVCC,intVCC);

      rawBRAMvoltage = XAdcPs_GetAdcData(&XADCblock, XADCPS_CH_VBRAM);
      BRAMvoltage = XAdcPs_RawToVoltage(rawBRAMvoltage);
          printf("raw BRAM voltage: %lu while BRAM voltage: %f\n\r",rawBRAMvoltage,BRAMvoltage);

          rawAUXVCC = XAdcPs_GetAdcData(&XADCblock, XADCPS_CH_VCCAUX);
          AUX_VCC = XAdcPs_RawToTemperature(rawAUXVCC);
          printf("raw Aux VCC: %lu while Aux VCC: %f\n\r",rawAUXVCC,AUX_VCC);
      printf("\r\n");
      printf("\r\n");
          delay();

     }

    cleanup_platform();
    return 0;
}

void delay (void)
{
        for(unsigned i = 0; i < 100000000; i++)
        {
                //do nothing
        }
}
```

It must be mentioned here that the raw data from the XADC should be shifted to the right by 4 places. This is not shown in this program however, it will be pointed out in the future programs.

Chapter 9 Sampling External ADCs from the Processing System

The previous chapter dealt with sampling the internal parameters of the System on Chip. As promised, this chapter will show how to monitor external analogue inputs from the Processing System. Later, the same will be done, but this time from the Programmable Logic.

So, create a project as shown in previous chapters. Do not include any VHDL modules. Once the Vivado project is opened, click on *Create a Block Design*. In the block design include the Zynq Processing System, the AXI interconnect and the XADC wizard.

The AXI interconnect should be configured to have one master output as shown in Figure 9.1 below.

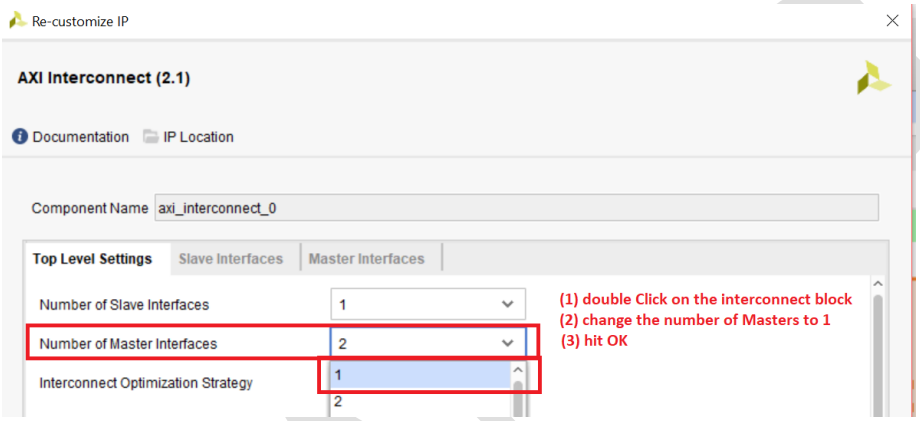


Figure 9. 1: Configure the AXI interconnect block

This time, the Processing System Reset Block will be added in the design. This will help reduce the warning and errors.

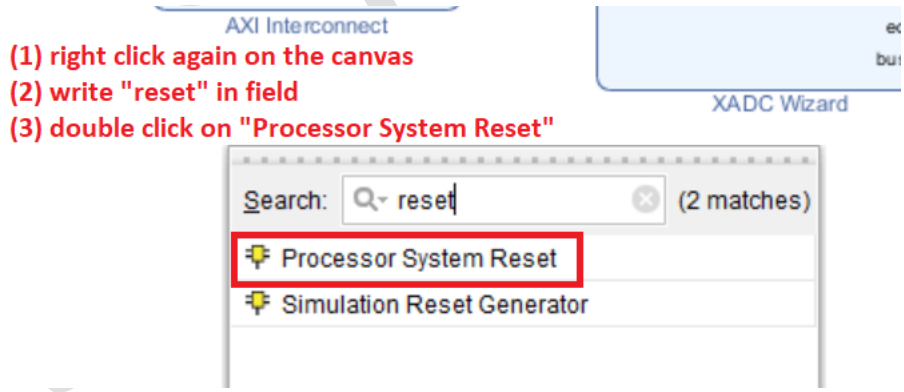


Figure 9. 2: Including the Processing System Reset Block

Now connect the resets of the AXI interconnect to the dedicated reset on the Processing System Reset block as shown in Figure 9.3 on the next page.

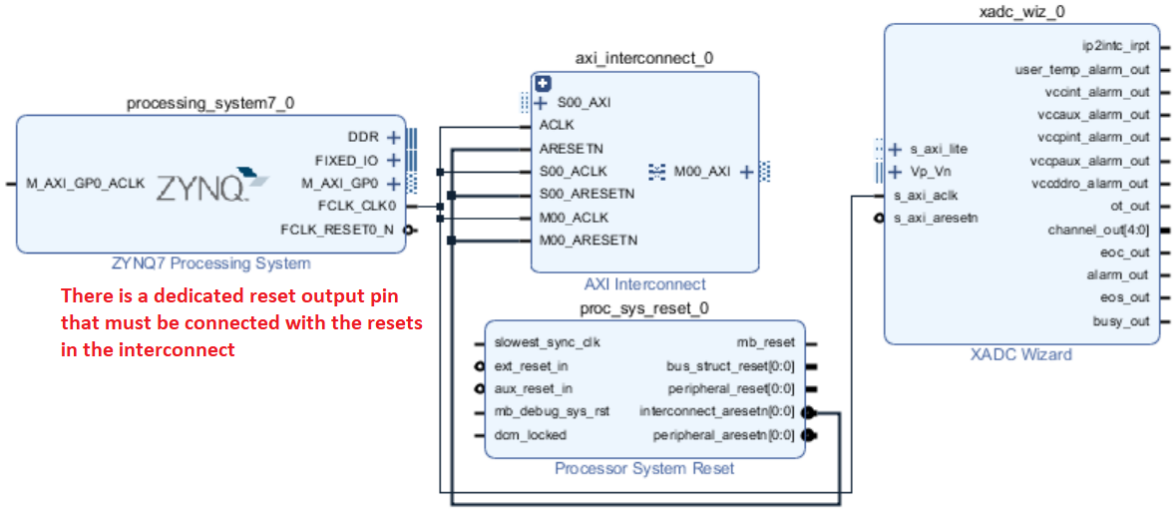


Figure 9. 3: Connecting the Reset of the AXI interconnect block

Now, connect the reset of the XADC wizard block to the Processing System Reset block as shown in Figure 9.4 below.

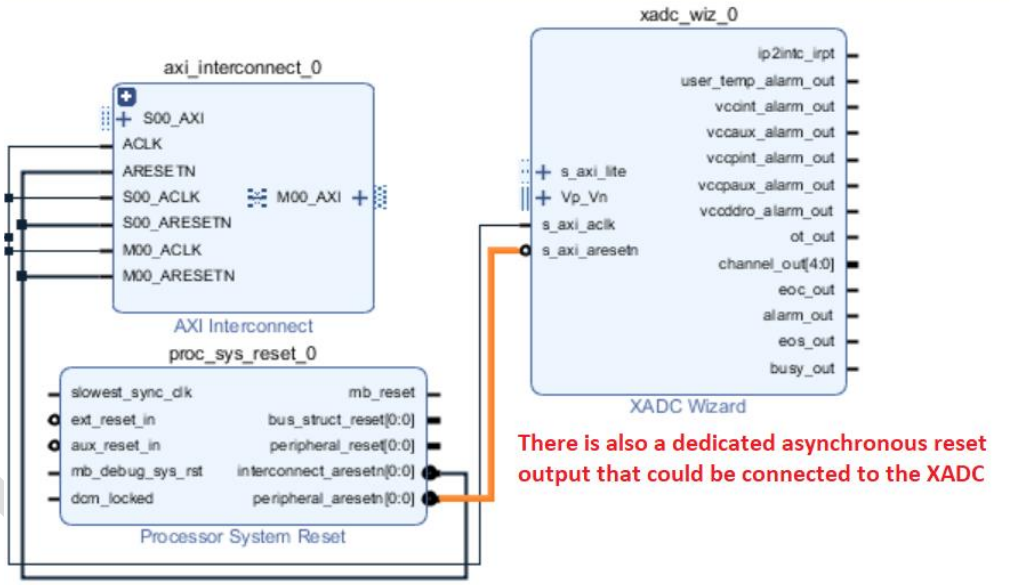


Figure 9. 4: Connecting the Reset of the XADC

Now connect the Reset of the Processing System to the Reset block as shown in Figure 9.5 on the next page.

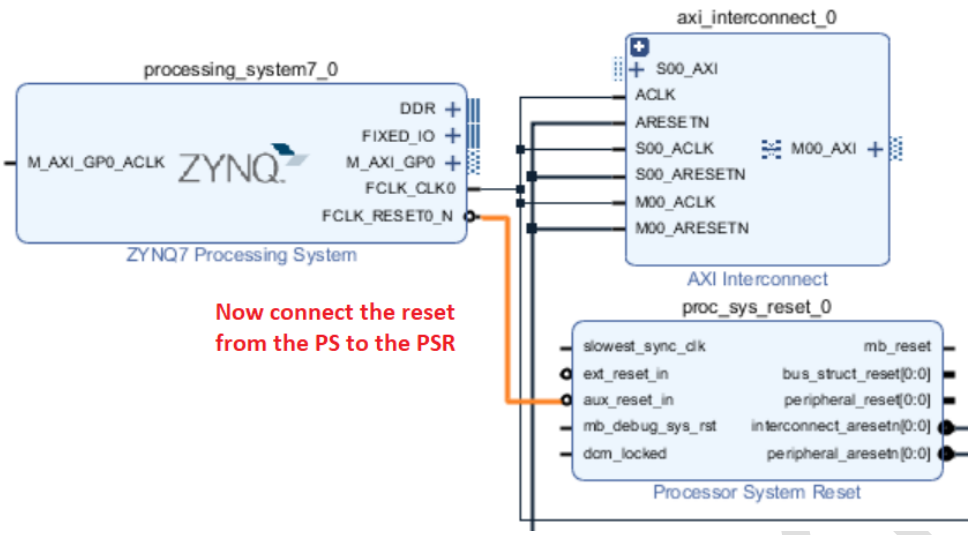
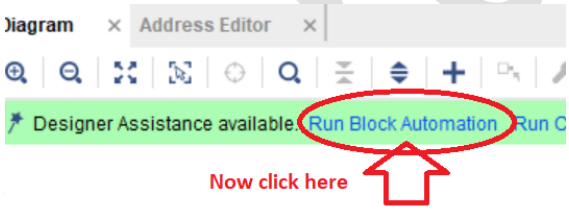
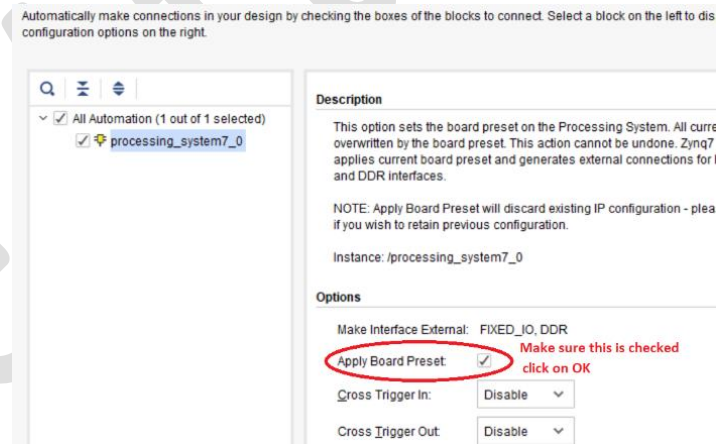


Figure 9. 5: Connecting the Reset of the Processing System

Click on *Run Block Automation.*



As always make sure the tick on Apply Board Preset box is present.



Now connect all the clock signals to the 100 MHz clock output from the Processing System.

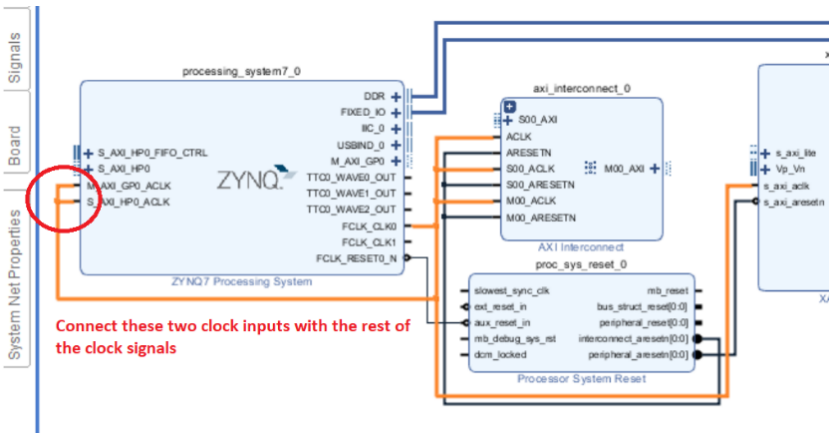


Figure 9. 6: Connecting the clocks

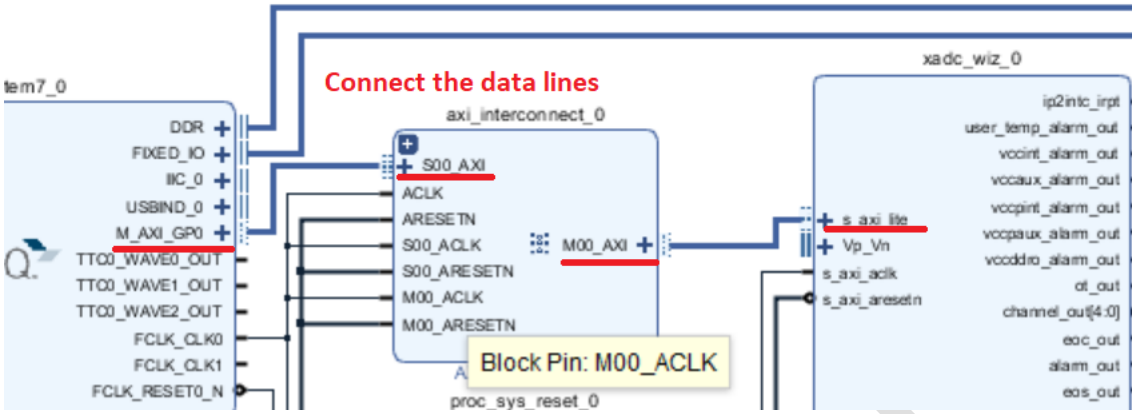


Figure 9. 7: Connecting the Data lines

Figure 9.7 shows how to connect the data lines between the Zynq Processing System, the AXI interconnect block and the XADC wizard.

The next thing to do is to configure the XADC wizard, so double click on the XADC block and follow the instructions in the Figures underneath.

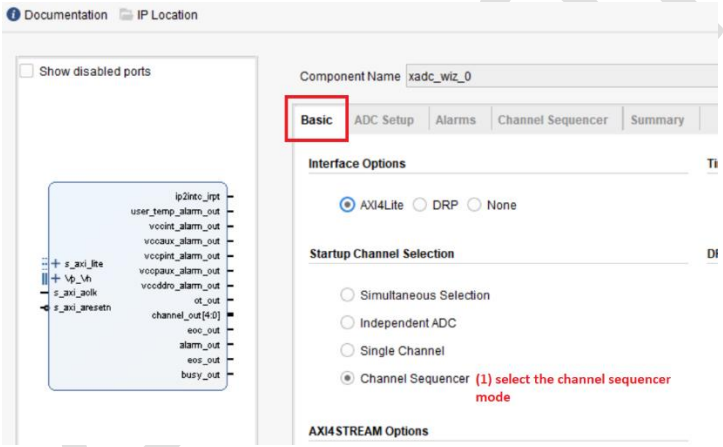


Figure 9. 8: Select Channel Sequencer

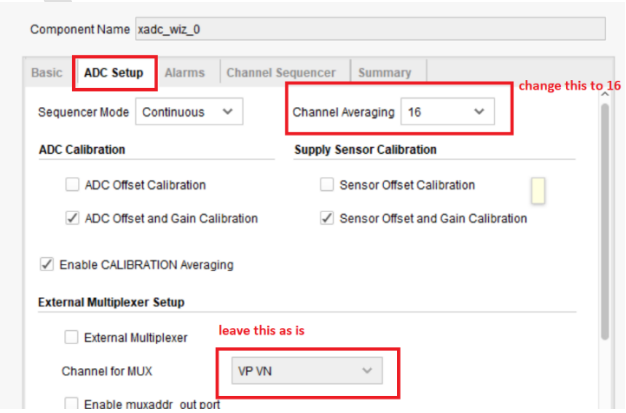


Figure 9. 9: Enable an average of 16

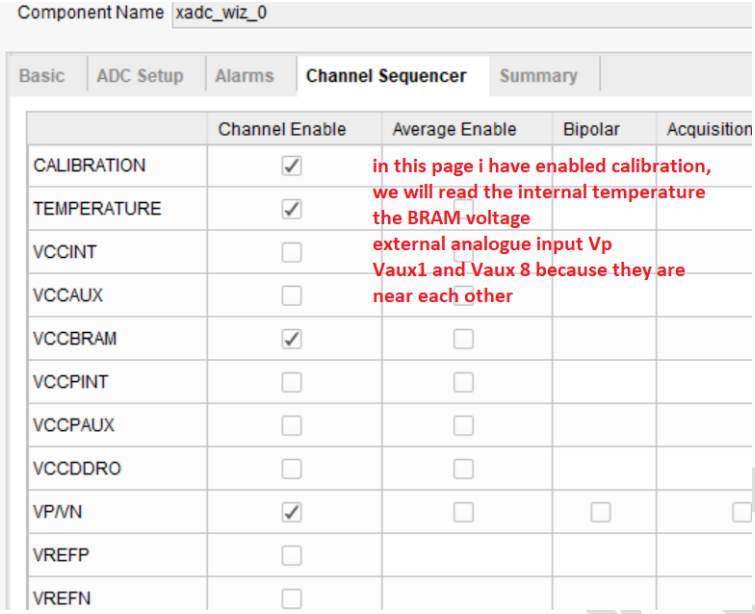
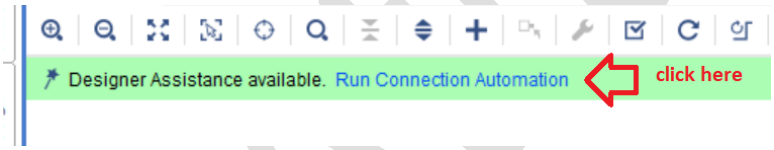


Figure 9. 10: Selecting the Channels

Figure 9.10 shows that the channels of interest must be selected from the list. The Auxiliary pins are not shown in Figure 9.10 but they are selected. Now close the XADC wizard configuration. Click on *Run Block Automation*.



For the window of Figure 9.11, click on OK.

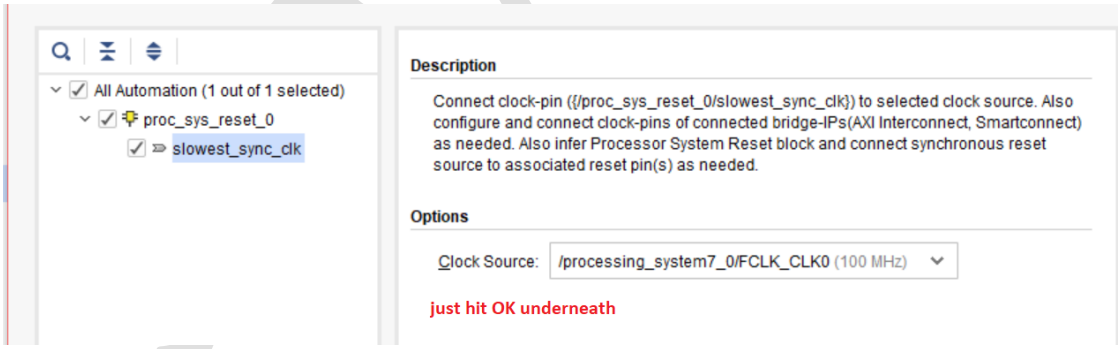
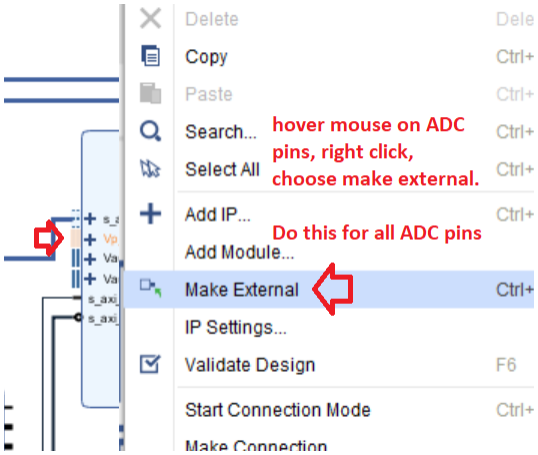


Figure 9. 11: Warning Box



Now, the Vp/Vn pins must be connected to their respective external pins, to do so, hover the mouse on Vp/Vn pins on the XADC wizard block, right click and then select Make External as shown in Figure 9.12.

Figure 9. 12: Connect Vp/Vn to their external pins

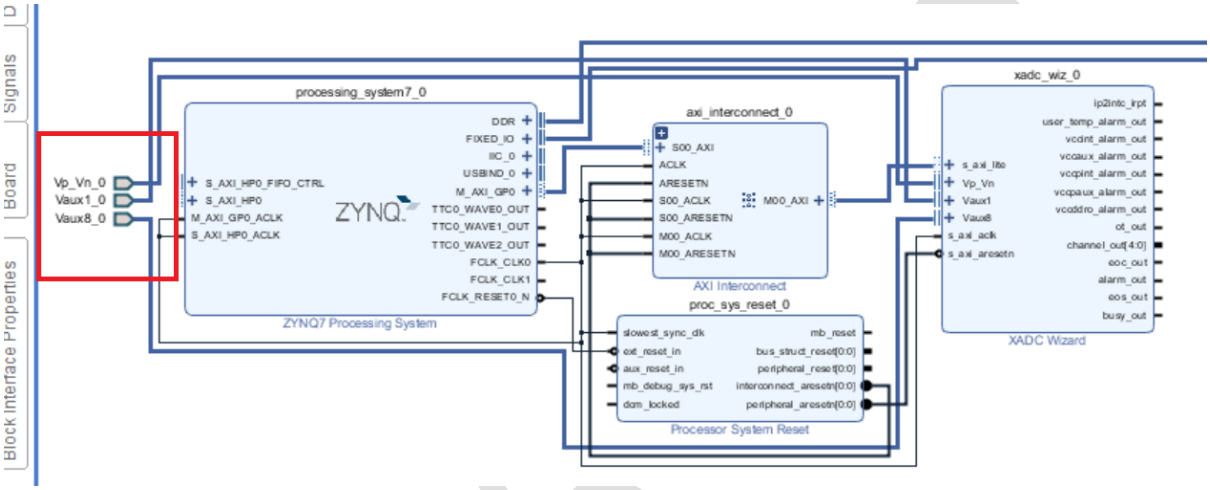
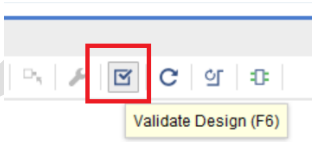


Figure 9. 13: Showing the External Pins of the ADCs

Note in Figure 9.13 that the auxiliary channels 1 and 8 have been enabled together with the dedicated Vp/Vn ADC. It would be a good idea if the design is validated.



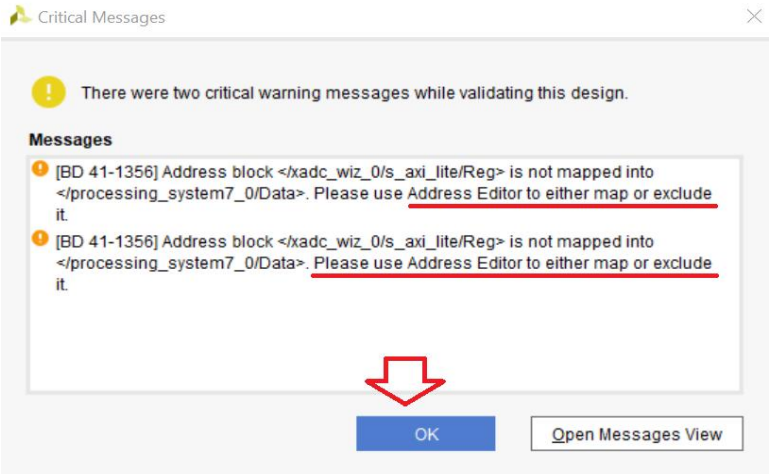


Figure 9. 14: Critical Warnings

If the warning in Figure 9.14 are displayed, then follow their instructions and use the address Editor to assign memory locations to the XADC block. This is illustrated in Figure 9.15 on the next page.

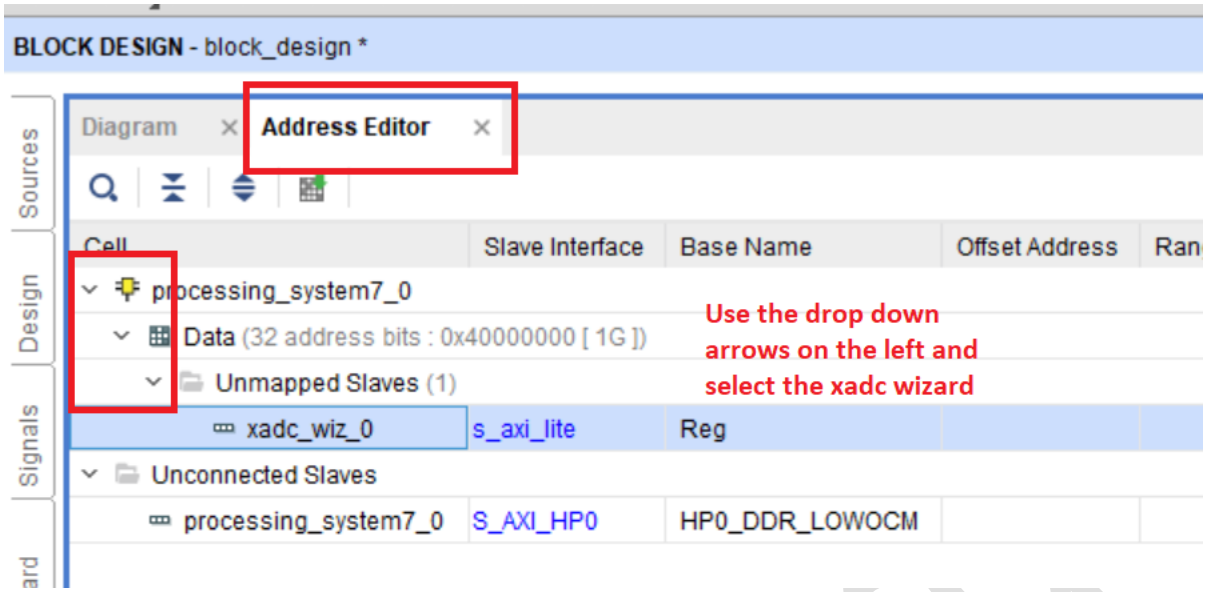


Figure 9. 15: Assigning an address to the XADC block

Figure 9. 16: Choose the Assign Address from list

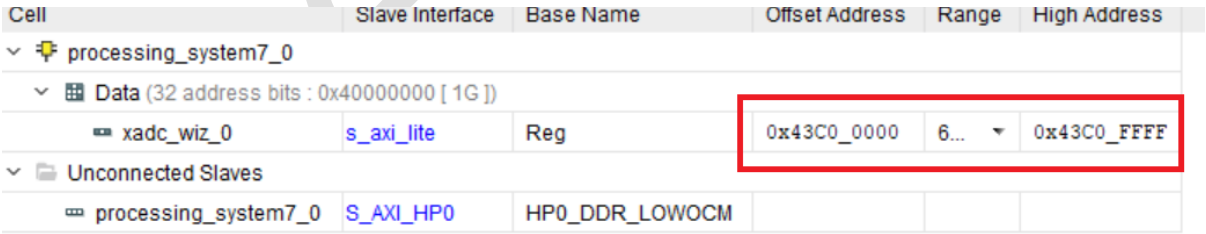


Figure 9. 17: The XADC block is assigned an address

Now it is a good idea to validate the design from the Block Design Menu.

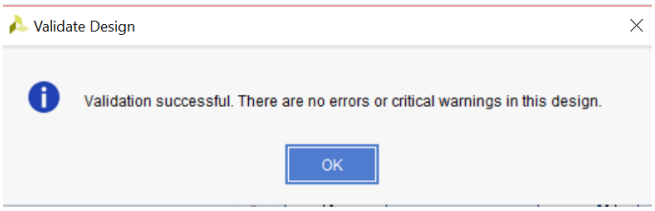


Figure 9. 18: Validation Successful message

Now create a Hardware Wrapper.

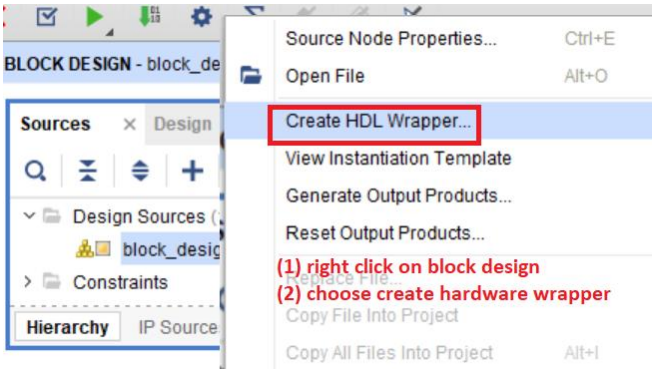


Figure 9. 19: Steps to Create a Hardware Wrapper

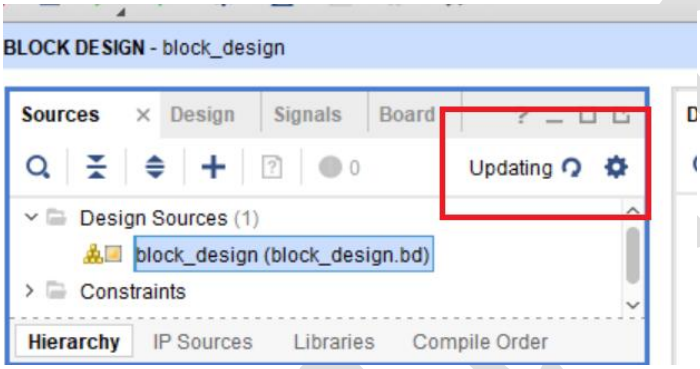


Figure 9. 20: Making sure that updates are done

Figure 9.20 shows an important step. If the updates are not done and the synthesis begins, Vivado will generate an error. So, make sure that the *Updating* message disappears before clicking to synthesize and implementation.

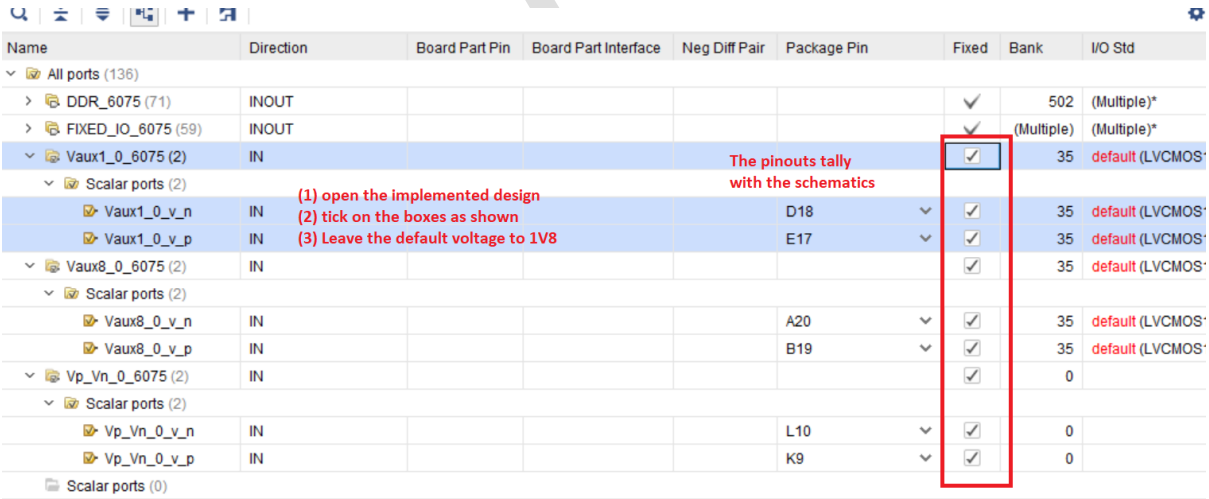


Figure 9. 21: Pinouts

Figure 9.21 is just a check to see that the pinouts assigned automatically by Vivado tally with the schematics. Make sure that the Fixed boxes are ticked. Now click on Generate Bitstream File and wait. If there are no errors and the bitstream file is generated successfully, export the hardware by File → Export → Export Hardware.

When it is done exporting the hardware, Launch SDK from within the Vivado project.

The Software

In SDK, create a new FSBL project and a new Hello World project. These two steps have been shown in previous chapters. For this project, the ADC readings will be shown on a serial terminal so the following BSP adjustments must be made. This has been shown before in previous chapters.

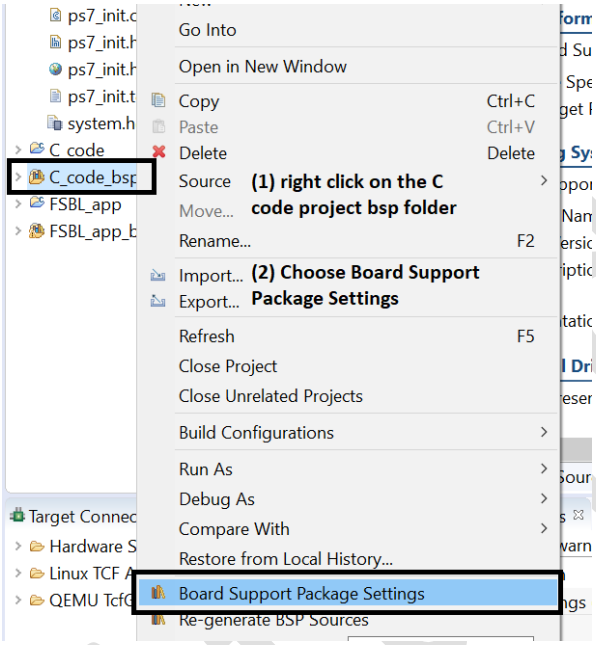


Figure 9. 22: Open the BSP Setting

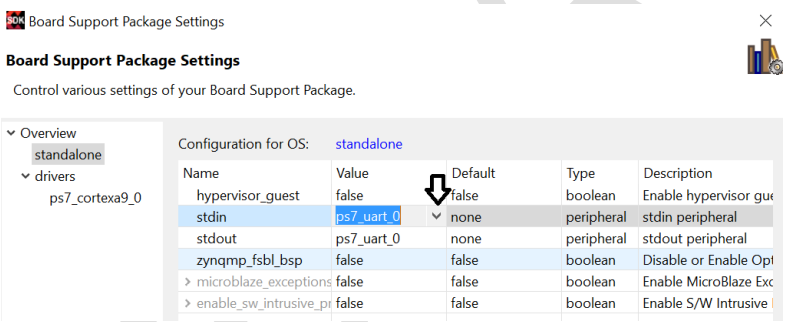
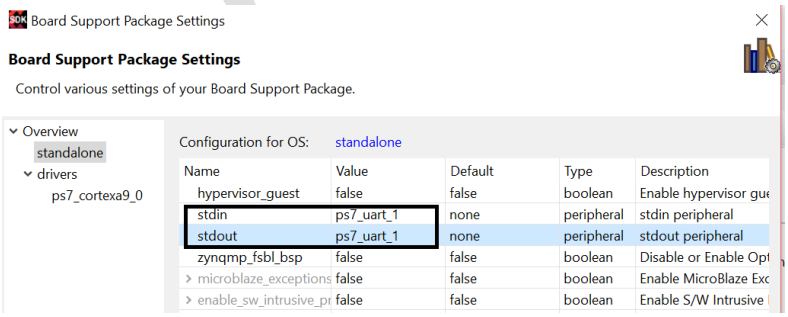


Figure 9.23 shows the steps to enable UART 1 instead of UART 0. This is because the Z-turn board has an interface chip for serial data communication connected to the pins of UART 1.

Figure 9. 23: Change to UART 1



Click on OK underneath and wait for the project to compile again.

Now open the HelloWorld.c file.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xadcps.h"
```

Figure 9. 24: Include the XADC library

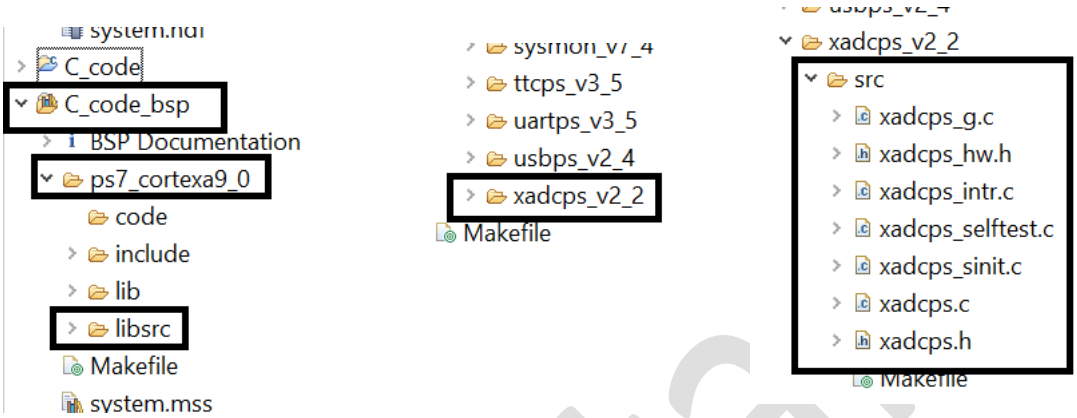


Figure 9. 25: Location of the XADC Library

Figure 9.25 shows the location of the XADC library within the C project environment. It also shows the functions associated with the XADC block.

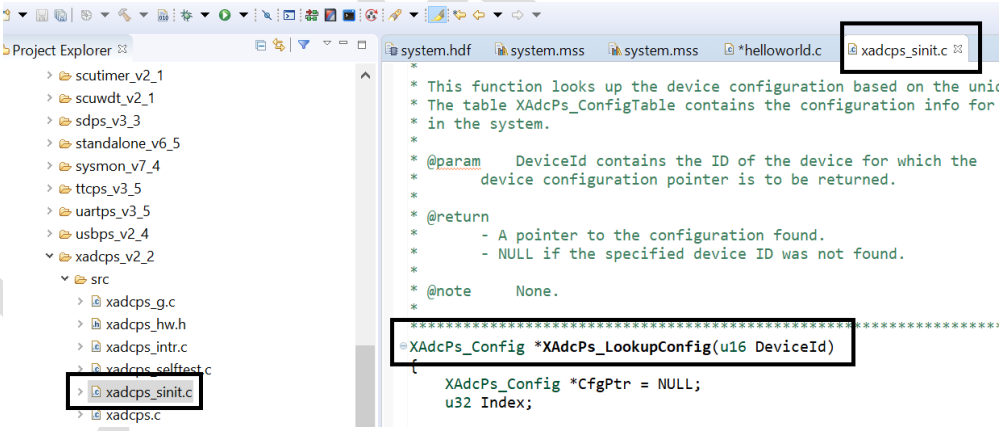


Figure 9. 26: Location of the Lookup function

From the project tree double click on ***xadcps_sinit.c*** file. Copy the lookup function name in the helloworld.c file.

```
XAdcPs_Config *XAdcPs_LookupConfig(u16 DeviceId)
```

Now this function returns an ***XAdcPs_Config*** type. Therefore, one must declare a variable at the beginning of the main function and equate this statement to the variable. Also, a parameter of type *u16 DeviceId* should be passed to this function.

This parameter is obtained from:

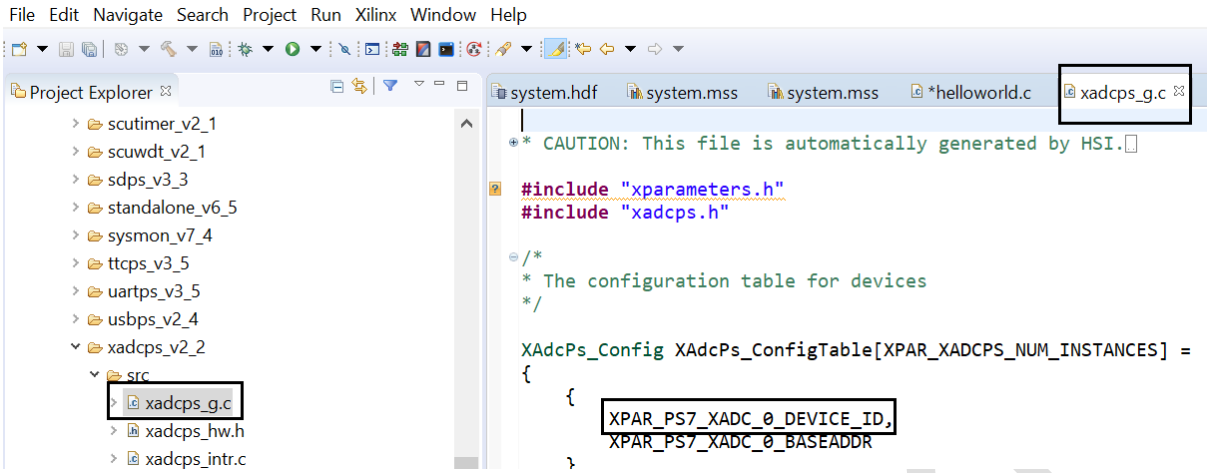


Figure 9. 27: Location of the u16 Device ID

So, the proper statement should be:

XADC_configPtr = XAdcPs_LookupConfig(XPAR_PS7_XADC_0_DEVICE_ID);

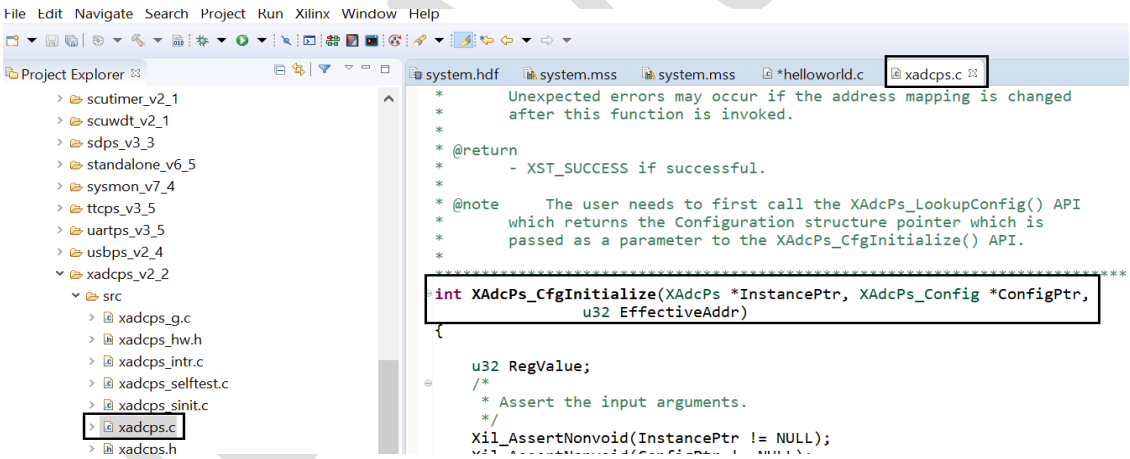


Figure 9. 28: Location of the Initialization Function

Also, part of the initialization is the function in Figure 9.28.

It returns an **int** type, therefore this must be declared as a variable at the beginning of the main function. Apart from that, there is an ***instance-pointer*** of type ***XAdcPs***. This must also be declared on top of the main function. The complete statement should look like Code Snippet 9.1:

```
main()

int init_success;
XAdcPs XADCperipheral;
XAdcPs_Config *XADC_configPtr;
init_platform();

/* Initialise the XADC*/
XADC_configPtr = XAdcPs_LookupConfig(XPAR_PS7_XADC_0_DEVICE_ID);
init_success=XAdcPs_CfgInitialize(&XADCperipheral,XADC_configPtr,XADC_configPtr->BaseAddre
if(init_success != XST_SUCCESS)
{
    return XST_FAILURE;
}
```

Variable declarations here

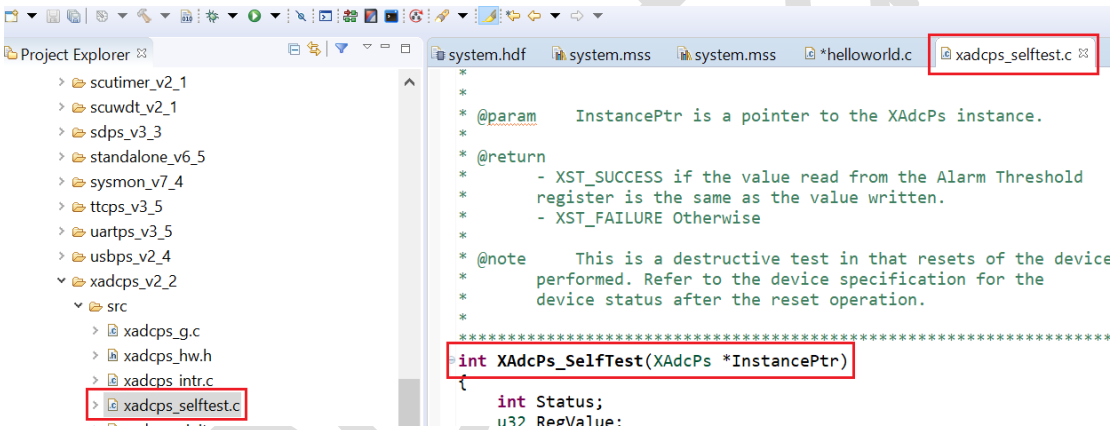
Complete initialization

Code Snippet 9. 1: Complete Initialization statement

Notice the instance-pointer: ***&XADCperipheral***

Self-Test

The self-test function is used to reset the XADC and to check whether the XADC is healthy. This will return a variable of type ***int*** and therefore this must be equated to another variable that should be declared at the beginning of the main function. This function resides in:



Code Snippet 9. 2: Location of the Self-Test Function

Code Snippet 9. 3: Writing the Self-test function in code

```
int init_success,STstatus;
XAdcPs XADCperipheral;
XAdcPs_Config *XADC_configPtr;
init_platform();

/* Initialise the XADC*/
XADC_configPtr = XAdcPs_LookupConfig(XPAR_PS7_XAI
init_success=XAdcPs_CfgInitialize(&XADCperiphera
if(init_success != XST_SUCCESS)
{
    return XST_FAILURE;
}

/*Self test. This should also reset the XADC*/
STstatus=XAdcPs_SelfTest(&XADCperipheral);
if(STstatus != XST_SUCCESS)
{
    return XST_SUCCESS;
}
```

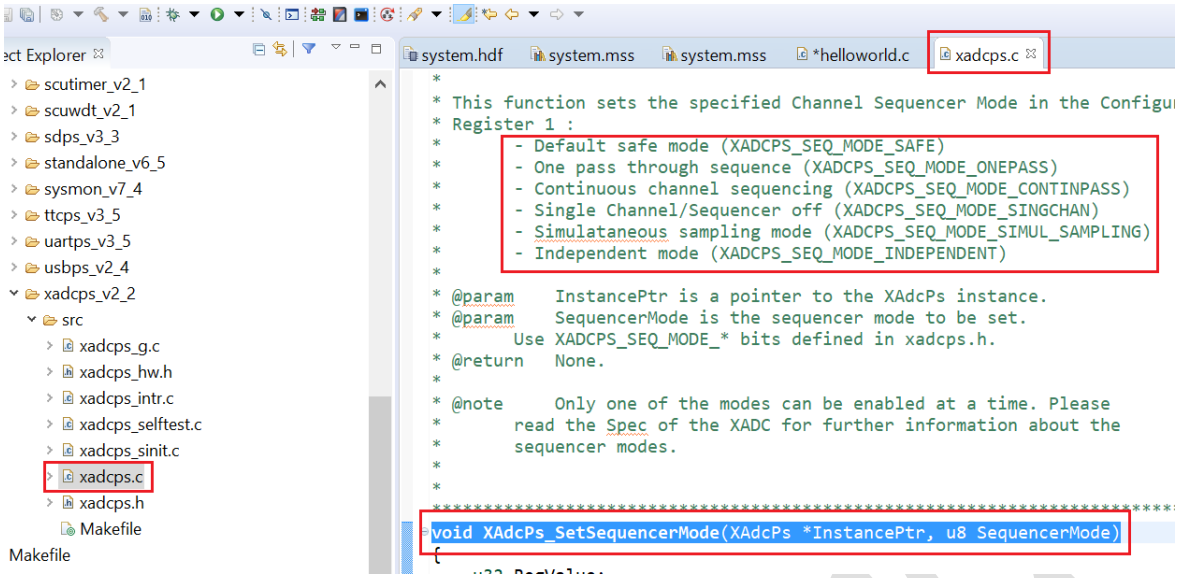


Figure 9. 29: Location of the Sequence Function

Figure 9.29 shows the location of the sequencer function. The XADC has to be stopped so that the configuration registers could be written to, to configure the XADC block. The function resides in ***xadcps.c.*** The parameters are listed in the comments list as illustrated in Figure 9.29.

```
/* now we stop the sequencer*/
XAdcPs_SetSequencerMode(&XADCperipheral, XADCPS_SEQ_MODE_SINGCHAN);
```

Code Snippet 9. 4: Stopping the XADC

As the comments in Figure 9.29 show, the XADC sequencer is stopped if the parameter passed to the SetSequence() is *SINGCHAN*.

The XADC should be put into safe mode so that the configuration registers could be changed:

```
/* we put the XADC in safe mode so that we can change the configuration registers*/
XAdcPs_SetSequencerMode(&XADCperipheral, XADCPS_SEQ_MODE_SAFE);
```

Code Snippet 9. 5: XADC in Safe Mode

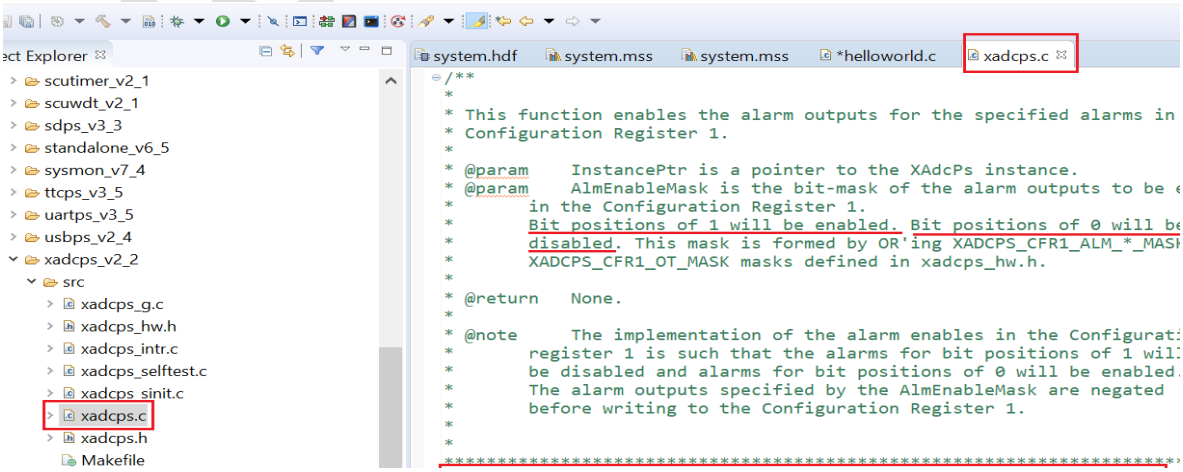


Figure 9. 30: Location of the Alarms Function

The first configuration is to disable the alarms. 0 will disable the alarms while 1 will enable the alarms.

```
/* Disable the alarms*/
XAdcPs_SetAlarmEnables(&XADCperipheral,0x0000);
```

Code Snippet 9. 6: Alarms Function

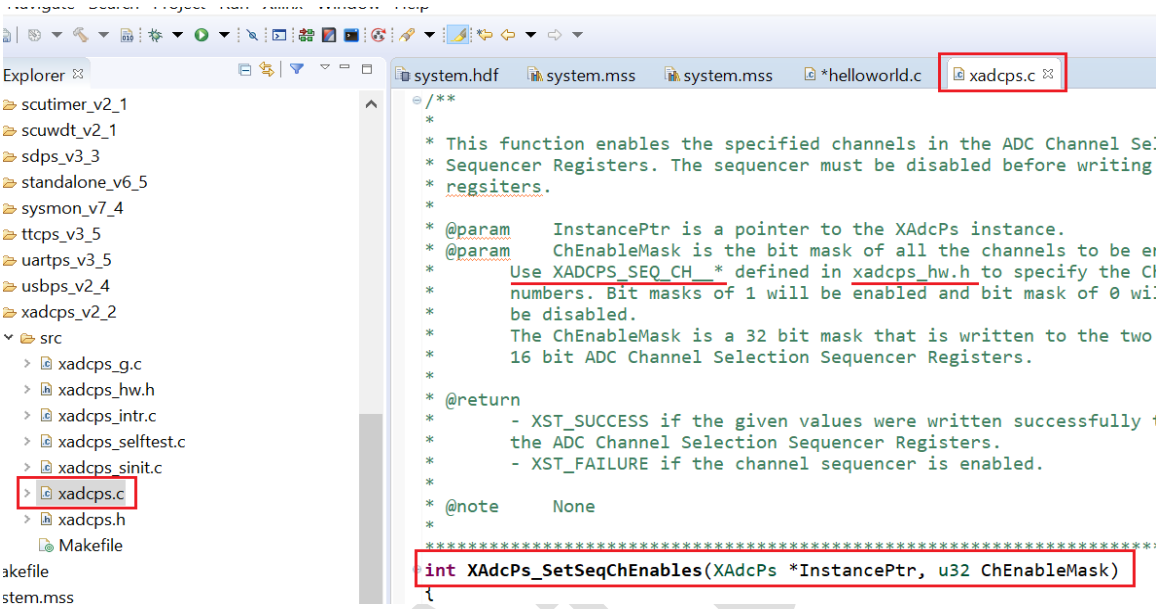


Figure 9. 31: Location of Channel Enable Function

Figure 9.31 shows the location where the function to enable the individual channels is located. The comments give a hint on how to identify the parameters that could be passed to this function and their location. This function returns a value of type ***int*** and therefore this must be declared again at the beginning of the main(). The parameters are listed in the Figure 9.32 and can be found in ***xadcps_hw.h*** file.

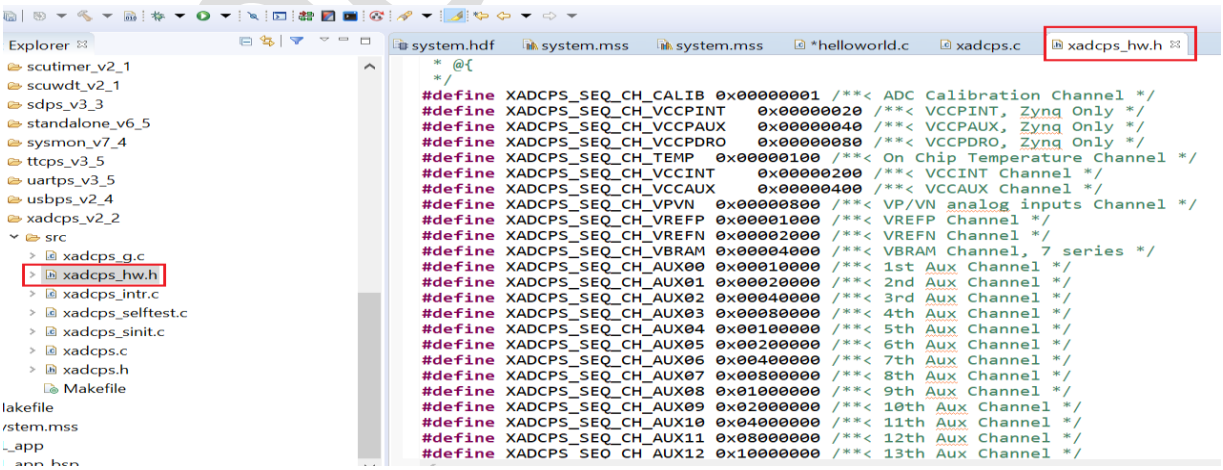


Figure 9. 32: Channel Parameter List

The **channels' enable function** should tally with the hardware we enabled in the XADC wizard which is part of the hardware-block-design. This is shown in Figure 9.33 again, on the next page, so that the student will not get confused.

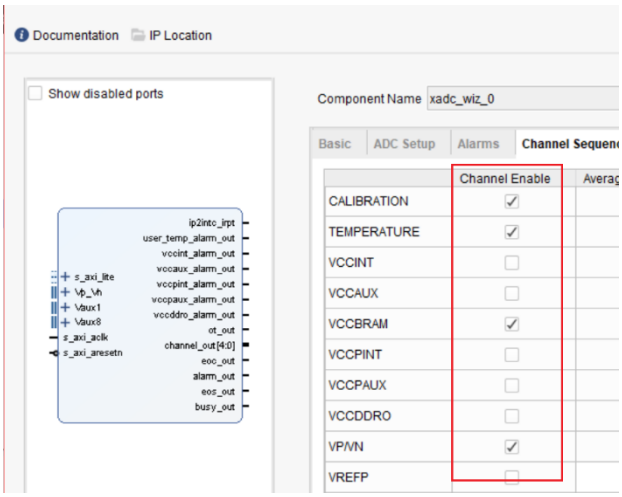


Figure 9. 33: ADC channels

Again, the AUXVp01/AUXVn01 and Vn8AUXVp08/AUXVn08 are not shown in Figure 9.33, however they are included so make sure that in the function parameters, this will be included!

```
/*now we will enable the channels we would like to monitor
 * The enabled channels should tally with what we have enabled in the
 * XADC wizard forming part of our hardware.*/
ch_status=XAdcPs_SetSeqChEnables(&XADCperipheral, XADCPS_SEQ_CH_CALIB |
                 XADCPS_SEQ_CH_TEMP | XADCPS_SEQ_CH_VPVN | XADCPS_SEQ_CH_VBRAM |
                 XADCPS_SEQ_CH_AUX01 | XADCPS_SEQ_CH_AUX08);

if(ch_status != XST_SUCCESS)
{
    return XST_FAILURE;
}
```

Code Snippet 9. 7: Syntax to Enable the ADC channels

This time, the Processing System will sample both internal parameters and also external ADC channels in the same program! Now, the sequence by which the channels will be sampled has to be configured as wel. The function that takes care of this resides in ***xadcps.c*** file. The parameters can be found in ***xadcps_hw.h*** file.

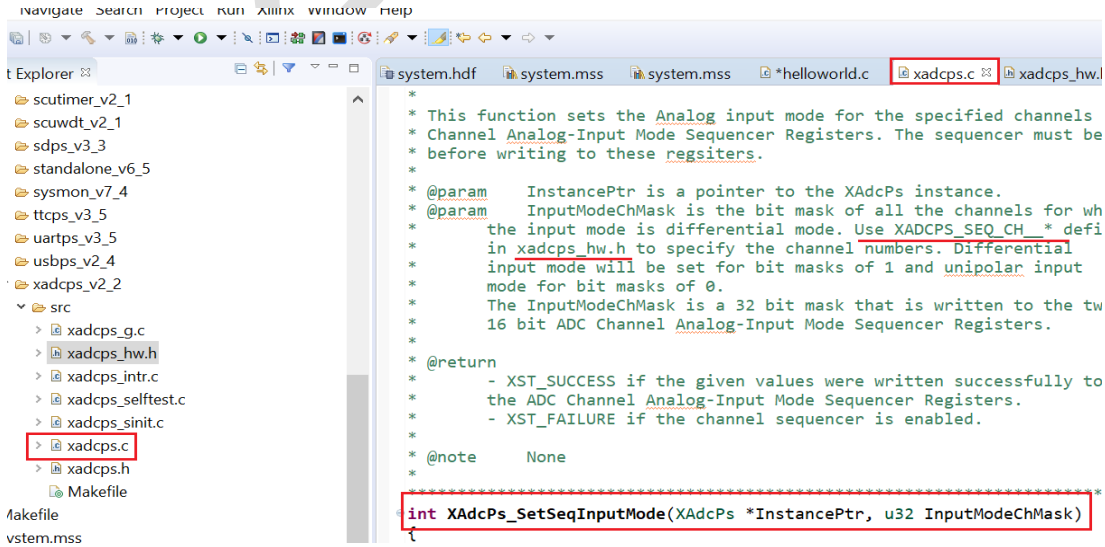


Figure 9. 34: Location of the Sampling Sequence Function

It is a good idea to copy the parameters of the channel enables function in the function shown in Figure 9.34.

```
/*Now we need to set which channels will be sampled in sequence*/
 SeqModeStatus=XAdcPs_SetSeqInputMode(&XADCperipheral, XADCPS_SEQ_CH_CALIB |
          XADCPS_SEQ_CH_TEMP | XADCPS_SEQ_CH_VPVN | XADCPS_SEQ_CH_VBRAM |
          XADCPS_SEQ_CH_AUX01 | XADCPS_SEQ_CH_AUX08);
 if(SeqModeStatus != XST_SUCCESS)
 {
     return XST_FAILURE;
 }
```

Code Snippet 9. 8: Sampling Sequence

The sampling will start from the first parameter. Once its finished and it stores its digital equivalent in the respective status register, the XADC samples the next channel according to the list of parameters shown in Code Snippet 9.8. It continues to sample the channels one after the other until all the channels are sampled. The XADC will start all over again if the next function is included.

```
/* Before starting to sample data we will set the XADC to continuously
 * sample the channels*/
XAdcPs_SetSequencerMode(&XADCperipheral, XADCPS_SEQ_MODE_CONTINPASS);
```

Code Snippet 9. 9: Function to sample continuously

The *get data()* is used to get the 12-bit decimal equivalent of the quantity you are monitoring. Its location is shown in Figure 9.35.

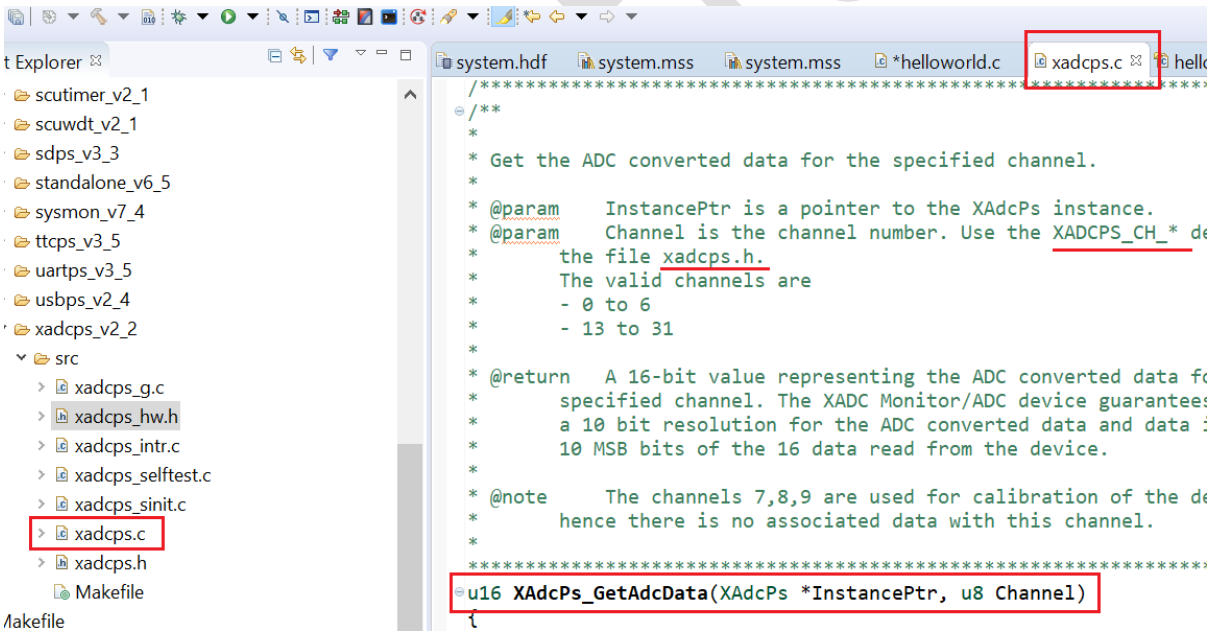


Figure 9. 35: Location of the GetADCData function

The parameters for the above function are stored in ***xadcps.h*** and are shown Figure 9.36.

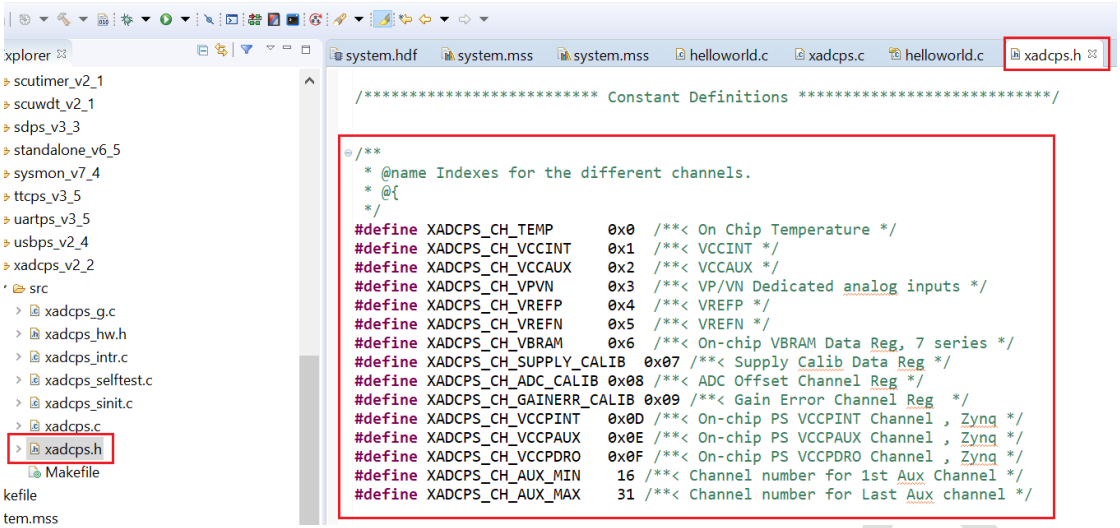


Figure 9. 36: Location of the parameters for the GetADCData Function

One other thing that needs to be clarified is the fact that get_data() is returning a 16 bit variable, however it has to be stored in a 32 bit variable. This is because the built-in macro converts the 12-bit data from the ADC into either voltage or temperature and the macro itself takes care to do the conversion, which is hidden from the user.

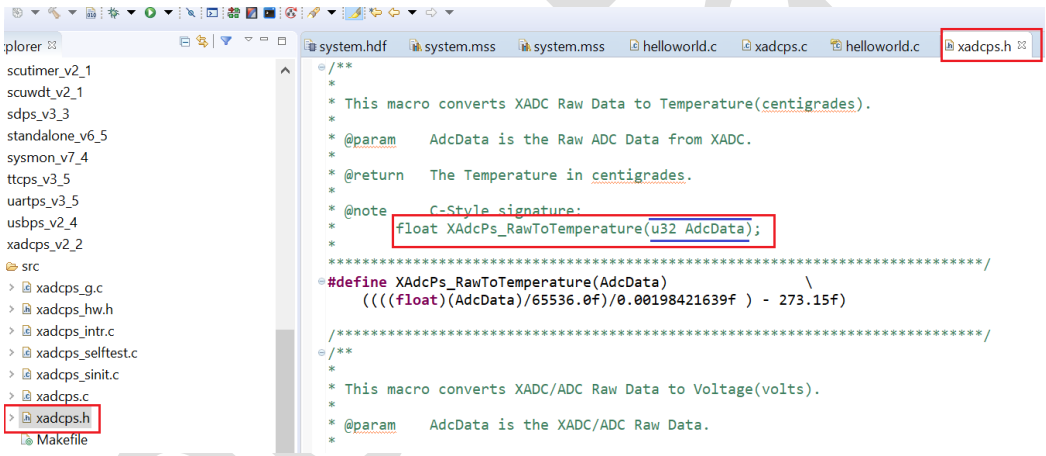


Figure 9. 37: Location of the built-in macro that converts raw data into temperature

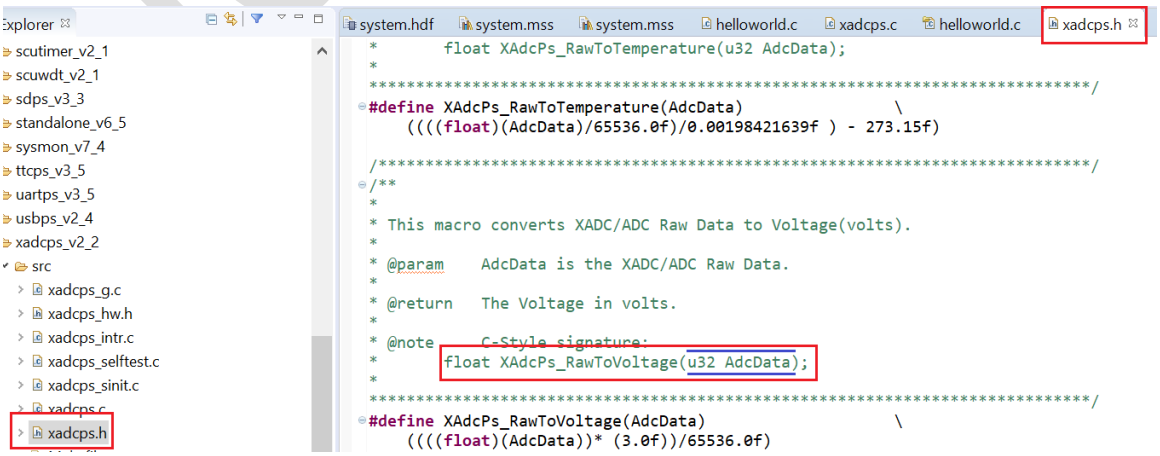


Figure 9. 38: Location of the built-in macro that converts raw data into voltage

The parameter list provided in the header file did not include all the auxiliary channel definitions and therefore the author had to include them manually. UG480, states that auxiliary channel 0 has an address of 16. The addresses continue to increment such that channel 15 has an address of 31. Figure 9.39 shows the definition statements that where included by the author in the header file. The list is shown in the red box.

```
#define XADCPS_CH_TEMP        0x0  /**< On chip Temperature */
#define XADCPS_CH_VCCINT      0x1  /**< VCCINT */
#define XADCPS_CH_VCCAUX      0x2  /**< VCCAUX */
#define XADCPS_CH_VPVN        0x3  /**< VP/VN Dedicated analog inputs */
#define XADCPS_CH_VREFP       0x4  /**< VREFP */
#define XADCPS_CH_VREFN       0x5  /**< VREFN */
#define XADCPS_CH_VBRAM       0x6  /**< On-chip VBRAM Data Reg, 7 series */
#define XADCPS_CH_SUPPLY_CALIB  0x07 /**< Supply Calib Data Reg */
#define XADCPS_CH_ADC_CALIB 0x08 /**< ADC Offset Channel Reg */
#define XADCPS_CH_GAINERR_CALIB 0x09 /**< Gain Error Channel Reg  */
#define XADCPS_CH_VCCPINT    0x0D /**< On-chip PS VCCPINT Channel , Zynq */
#define XADCPS_CH_VCCPAUX    0x0E /**< On-chip PS VCCPAUX Channel , Zynq */
#define XADCPS_CH_VCCPDRO    0x0F /**< On-chip PS VCCPDRO Channel , Zynq */
#define XADCPS_CH_AUX_MIN     16 /**< Channel number for  Aux00 Channel */

#define XADCPS_CH_AUX01       17 /**< Channel number for Aux01 Channel */
#define XADCPS_CH_AUX02       18 /**< Channel number for Aux02 Channel */
#define XADCPS_CH_AUX03       19 /**< Channel number for Aux03 Channel */
#define XADCPS_CH_AUX04       20 /**< Channel number for Aux04 Channel */
#define XADCPS_CH_AUX05       21 /**< Channel number for Aux05 Channel */
#define XADCPS_CH_AUX06       22 /**< Channel number for Aux06 Channel */
#define XADCPS_CH_AUX07       23 /**< Channel number for Aux07 Channel */
#define XADCPS_CH_AUX08       24 /**< Channel number for Aux08 Channel */

#define XADCPS_CH_AUX_MAX     31 /**< Channel number for Last Aux channel */
```

Figure 9. 39: Adding the channel numbers manually

Since the definition statements where included manually by the author, it is imperative to either click on Save all or make sure that the header file is saved before saving the actual C file.

The following is the get_data() together with the macros in the while (1) loop:

```
while(1)
{
    ADC_valueTemp=XAdcPs_GetAdcData(&XADCperipheral, XADCPS_CH_TEMP);
    temp=XAdcPs_RawToTemperature(ADC_valueTemp);

    ADC_valueVp=XAdcPs_GetAdcData(&XADCperipheral, XADCPS_CH_VPVN);
    Vp=XAdcPs_RawToVoltage(ADC_valueVp);

    ADC_valueVBRAM=XAdcPs_GetAdcData(&XADCperipheral, XADCPS_CH_VBRAM);
    VBRAM=XAdcPs_RawToVoltage(ADC_valueVBRAM);

    ADC_valueAux01=XAdcPs_GetAdcData(&XADCperipheral, XADCPS_CH_AUX01);
    Aux01=XAdcPs_RawToVoltage(ADC_valueAux01);

    ADC_valueAux08=XAdcPs_GetAdcData(&XADCperipheral, XADCPS_CH_AUX08);
    Aux08=XAdcPs_RawToVoltage(ADC_valueAux08);

    printf("Internal temperature: %f\n\r",temp);
    printf("Vp voltage: %f\n\r",Vp);
    printf("VBRAM voltage: %f\n\r",VBRAM);
    printf("Aux01 voltage: %f\n\r",Aux01);
    printf("Aux08 voltage: %f\n\r",Aux08);
    delay();
```

Code Snippet 9. 10: Sampling the ADC channels

Print() must be changed to printf() statement because this will generate an error of too many parameters. Create a boot image file and copy it to SD card. What follows is the whole code:

```
/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 *this project will attempt to control multiple ADC inputs
 *such as some of the internal parameters, Vp and also Aux1
 * ------------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ------------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xadcps.h"

void delay (void);

int main()
{
    int init_success,STstatus,ch_status,SeqModeStatus;
    u32 ADC_valueTemp,ADC_valueVp,ADC_valueVBRAM,ADC_valueAux01,ADC_valueAux08;
    float temp,Vp,VBRAM,Aux01,Aux08;
    XAdcPs XADCperipheral;
    XAdcPs_Config *XADC_configPtr;
    init_platform();

    /* Initialise the XADC*/
    XADC_configPtr = XAdcPs_LookupConfig(XPAR_PS7_XADC_0_DEVICE_ID);
    init_success=XAdcPs_CfgInitialize(&XADCperipheral,XADC_configPtr,XADC_configPtr->BaseA
    if(init_success != XST_SUCCESS)
    {
        return XST_FAILURE;
    }

    /*Self test. This should also reset the XADC*/
    STstatus=XAdcPs_SelfTest(&XADCperipheral);
    if(STstatus != XST_SUCCESS)
    {
        return XST_SUCCESS;
    }

/*Self test. This should also reset the XADC*/
STstatus=XAdcPs_SelfTest(&XADCperipheral);
if(STstatus != XST_SUCCESS)
{
    return XST_SUCCESS;
}

/* now we stop the sequencer*/
XAdcPs_SetSequencerMode(&XADCperipheral, XADCPS_SEQ_MODE_SINGCHAN);

/* we put the XADC in safe mode so that we can change the configuration registers*/
XAdcPs_SetSequencerMode(&XADCperipheral, XADCPS_SEQ_MODE_SAFE);

/* Disable the alarms*/
XAdcPs_SetAlarmEnables(&XADCperipheral,0x0000);

/*now we will enable the channels we would like to monitor
 * The enabled channels should tally with what we have enabled in the
 * XADC wizard forming part of our hardware.*/
ch_status=XAdcPs_SetSeqChEnables(&XADCperipheral, XADCPS_SEQ_CH_CALIB |
                XADCPS_SEQ_CH_TEMP | XADCPS_SEQ_CH_VPVN | XADCPS_SEQ_CH_VBRAM |
                XADCPS_SEQ_CH_AUX01 | XADCPS_SEQ_CH_AUX08);
```

```
if(ch_status != XST_SUCCESS)
{
    return XST_FAILURE;
}

/*Now we need to set which channels will be sampled in sequence*/
SeqModeStatus=XAdcPs_SetSeqInputMode(&XADCperipheral, XADCPS_SEQ_CH_CALIB |
         XADCPS_SEQ_CH_TEMP | XADCPS_SEQ_CH_VPVN | XADCPS_SEQ_CH_VBRAM |
         XADCPS_SEQ_CH_AUX01 | XADCPS_SEQ_CH_AUX08);
if(SeqModeStatus != XST_SUCCESS)
{
    return XST_FAILURE;
}

/* Before starting to sample data we will set the XADC to continuously
 * sample the channels*/
XAdcPs_SetSequencerMode(&XADCperipheral, XADCPS_SEQ_MODE_CONTINPASS);


while(1)
{
    ADC_valueTemp=XAdcPs_GetAdcData(&XADCperipheral, XADCPS_CH_TEMP);
    temp=XAdcPs_RawToTemperature(ADC_valueTemp);

    ADC_valueVp=XAdcPs_GetAdcData(&XADCperipheral, XADCPS_CH_VPVN);
    Vp=XAdcPs_RawToVoltage(ADC_valueVp);

    ADC_valueVBRAM=XAdcPs_GetAdcData(&XADCperipheral, XADCPS_CH_VBRAM);
    VBRAM=XAdcPs_RawToVoltage(ADC_valueVBRAM);

    ADC_valueAux01=XAdcPs_GetAdcData(&XADCperipheral, XADCPS_CH_AUX01);
    Aux01=XAdcPs_RawToVoltage(ADC_valueAux01);

    ADC_valueAux08=XAdcPs_GetAdcData(&XADCperipheral, XADCPS_CH_AUX08);
    Aux08=XAdcPs_RawToVoltage(ADC_valueAux08);

    printf("Internal temperature: %f\n\r",temp);
    printf("Vp voltage: %f\n\r",Vp);
    printf("VBRAM voltage: %f\n\r",VBRAM);
    printf("Aux01 voltage: %f\n\r",Aux01);
    printf("Aux08 voltage: %f\n\r",Aux08);
    delay();

}

    cleanup_platform();
    return 0;
}

void delay (void)
{
    for(unsigned i=0;i<10000000;i++)
    {

    }
}
```

So, the focus of this chapter was to show how to sample internal parameters, the dedicated ADC channel and two auxiliary ADC channels from the Processing System. During the discussion, the short comings encountered by the author were highlighted and their workaround explained. It must be said that the voltage and temperature macros offered by Xilinx are not so reliable and one should write his/her own functions to convert to temperature and voltage. In the next chapters, the XADC will be sampled from the Programmable Logic part.

Monitoring two ADC channels with data simultaneously shared between the PS and PL parts of the Zynq 7

Introduction

In this chapter, two external analogue inputs will be monitored by both the ***Processing System*** part and the ***Programmable Logic*** part of the Zynq 7. It will be shown how easy it is to use the XADC block simply because the advanced hardware included in the XADC makes life so much easier for the design engineer!

Creating a Vivado Project

There are sections in chapters 1 and 2 that explain in detail how to create a Vivado project both for the PL part and also for the PS part, so this part of the document will be skipped.

After creating the project and also creating a VHDL source file as part of the project, one should wait for Vivado to update as shown in Figure 10.1 below:

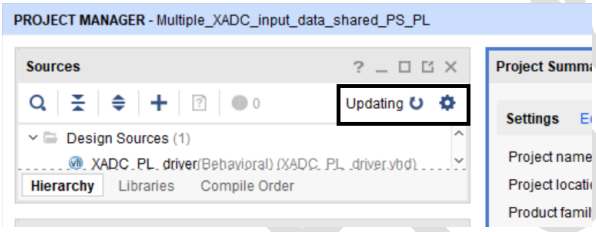


Figure 10. 1: Wait for Vivado to Update

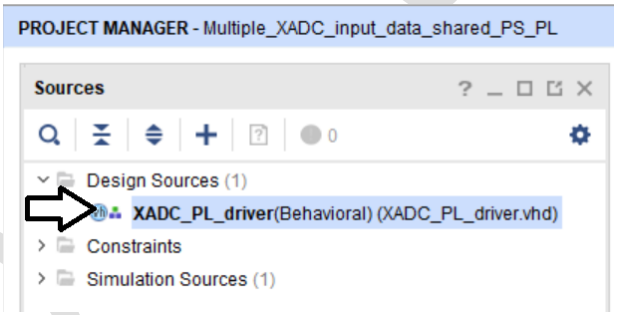


Figure 10. 2: Location of the VHDL Module

Figure 10.2 shows the location of the VHDL module within the project. Double click on it to edit it.

The XADC module will be configured to run in *continuous mode*, therefore it will output the ADC result in 12-bit digital form, together with the corresponding *channel-address*. Since the main objective of this exercise is to learn how to configure the XADC block and interface it with both PS and PL parts of the Zynq 7, the VHDL module will output the raw 16-bit result directly to the LEDs. It is known that the XADC result resides between bit 4 and bit 15 and therefore some form of processing is needed to obtain the actual value as a decimal number! The Zynq Processing system will do its own processing (shifting to the right by 4 bits) on the XADC data while the VHDL module will do its own separate processing. Code snippet 10.1 shows the VHDL code to implement a simple multiplexer because the objective of this

project is to make sure that the XADC data is available for both PS and PL at the same time! Also note how simple concept used to implement shifting of data in VHDL!

```
--use UNISIM.VComponents.all;

entity XADC_PL_driver is
    Port (
            SevenSeg : out STD_LOGIC_VECTOR (6 downto 0);
            channeladdr_out : out std_logic_vector(6 downto 0);
            ADCresult_in : in std_logic_vector(15 downto 0);
            ADCresult_out : out STD_LOGIC_VECTOR (11 downto 0);
            ADCchannel : in STD_LOGIC_VECTOR (4 downto 0));
end XADC_PL_driver;

architecture Behavioral of XADC_PL_driver is

signal internalADCresult : std_logic_vector(11 downto 0);

begin
channeladdr_out <= "00" & ADCchannel;

ADCresult_out <= ADCresult_in (15 downto 4) when ADCchannel = "00011" else
                 ADCresult_in (15 downto 4) when ADCchannel = "11000" else
                 "000000000000";
```

Code Snippet 10. 1 VHDL module

Note: One can change the names of the IO terminals of the VHDL module any time.

A simple method to output digital results to an output port, one can use a multiplexer (MUX).

By using a multiplexer, there is no issue of timing constraints or worrying that the XADC block is not in sync with the VHDL module because the multiplexer used will use the channel address of the analogue input as the select bits to select which digital data it will be output.

For this experiment a custom-made development board designed by the author was connected to the cape IO board by MYIR. It was fully isolated from both the inputs and the outputs, to make sure that the pins of the Zynq 7 will never get damaged. This dev-board extension had 18 LEDs driven by opto-transistors, 4 pre-set pots, 4 push-to-make switches, 4 slide switches and a single seven segment display.

Save the VHDL module and create a block design.

Note: One could start from the block diagram and then write the VHDL code after. There is no priority!

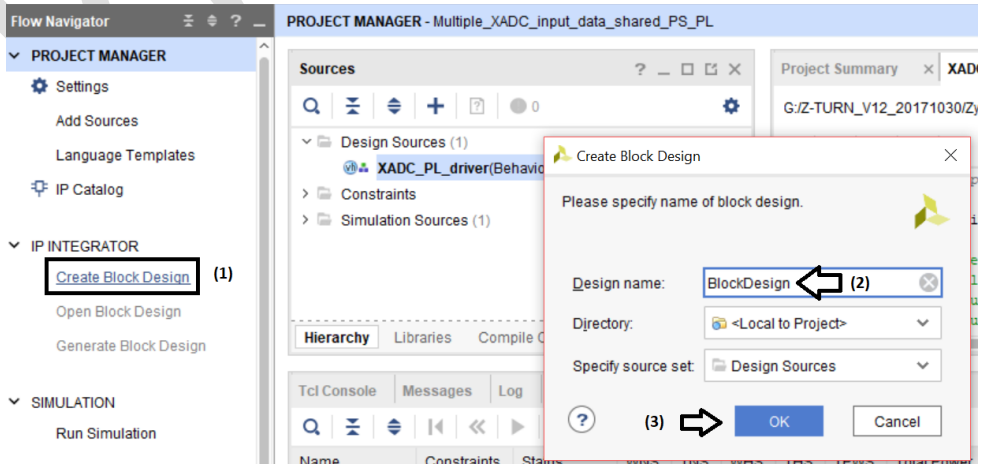


Figure 10. 3: Creating a Block Design

Click on “create a Block Design”. In the following pop up window give a name to the block design and then click on OK.

Adding the Zynq PS system

The Zynq PS system must be added to your block design because it will download the .bit file of the VHDL part of the project to the FPGA part of the Zynq SoC. Apart from that, in this project, the Zynq 7 Processing System will be used to monitor *in parallel* the ADC data from XADC.

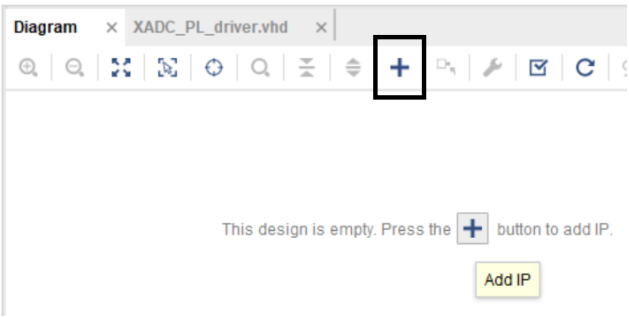


Figure 10. 4: Adding an IP to the Block Design

To add an IP on the canvas, one can either hover the mouse on the + sign in the middle or on the + sign that is part of the menu. Click on either one of them and a new pop up window pops up.

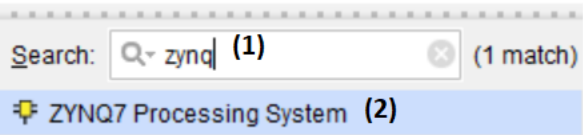


Figure 10. 5: Calling the IPs

Write the name of the IP in the field provided and double click on it to add it to the block design.

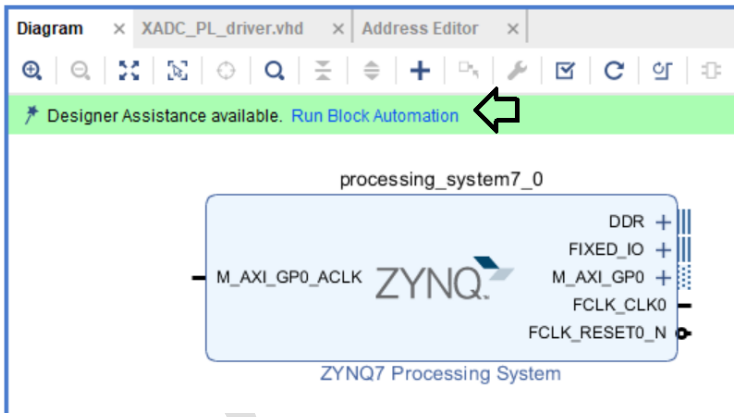


Figure 10. 6: Zynq PS is part of the Block Design

Click on Run Block Automation

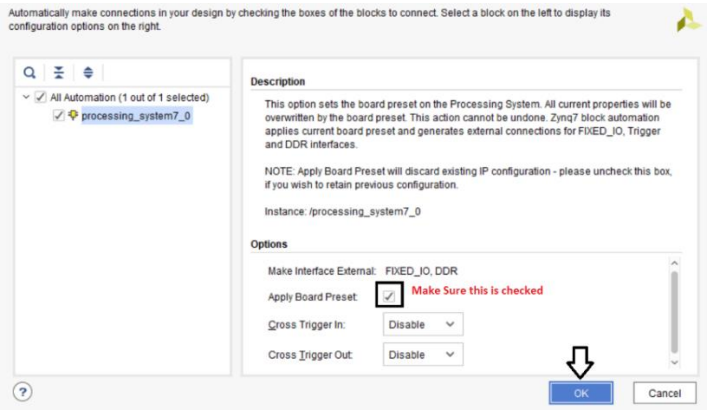


Figure 10. 7: Leave all the Presets

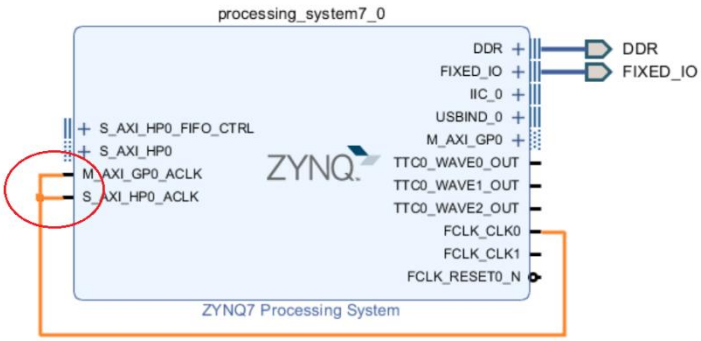


Figure 10. 8: Connect the AXI clocks to the Zynq PS

For this project, two AXI blocks will be used, one to interface the XADC block with the Zynq Processing System and one to send the processed XADC result to the VHDL module. The XADC will output the XADC result in 16 bit format together with the 6 bit channel address. The other AXI block is used to interface the Processing System with the Programmable Logic fabric. Another way to add an IP block is shown in Figure 10.9

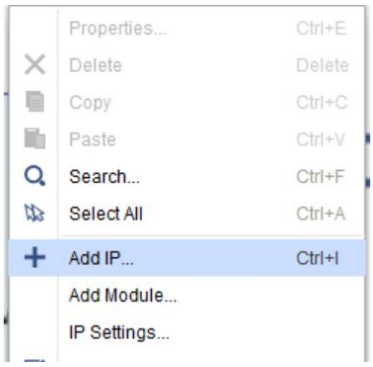


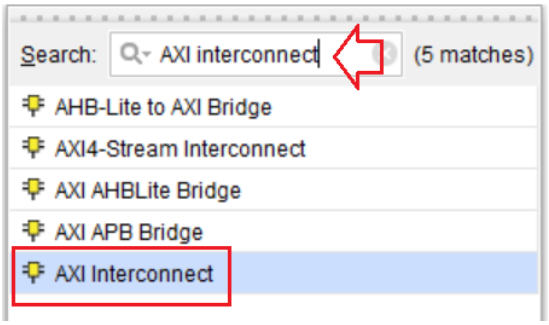
Figure 10. 9: Adding another IP block

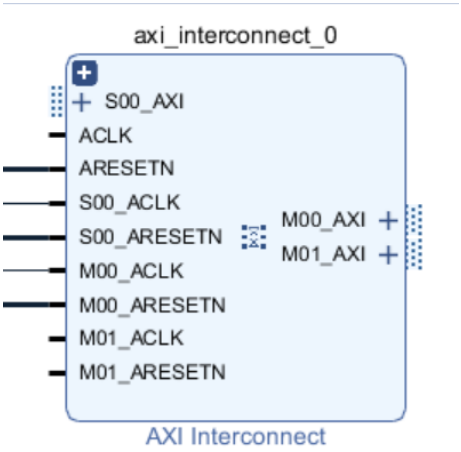
To add another *IP*, right-click on the canvas and select *Add IP* from the menu.

Where there is an AXI block, there should also be an *AXI interconnect block*. This ensures maximum data rate transfer between the Zynq Processing System and the Programmable Logic in the block-design.

Adding the AXI interconnect Block

Figure 10. 10: Adding the AXI interconnect Block





In the field, write AXI interconnect and then double-click on it from the list.

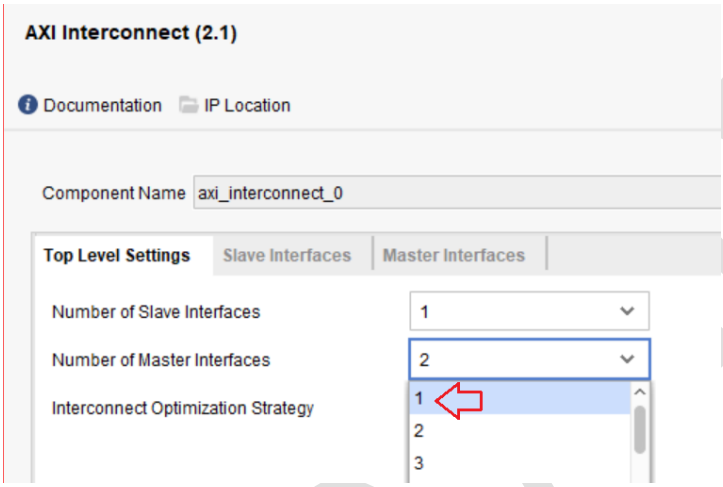
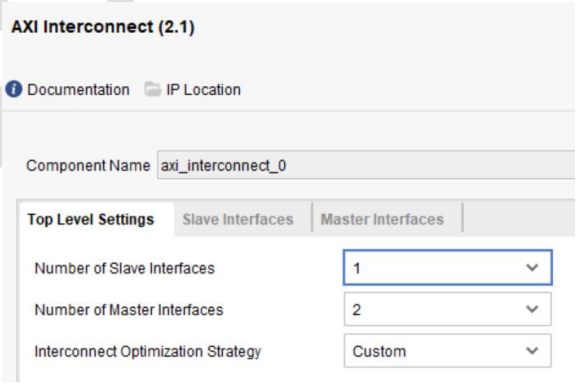


Figure 10. 11: Change the AXI interconnect block settings

AXI interconnect must have two master and one slave interface for **this application**.

With the adjacent settings, two AXI GPIO blocks could be connected to the AXI interconnect. Each AXI GPIO block has two channels. In the first AXI GPIO block, both channels will be used as input channels while the second AXI GPIO will have its channels configured as outputs.



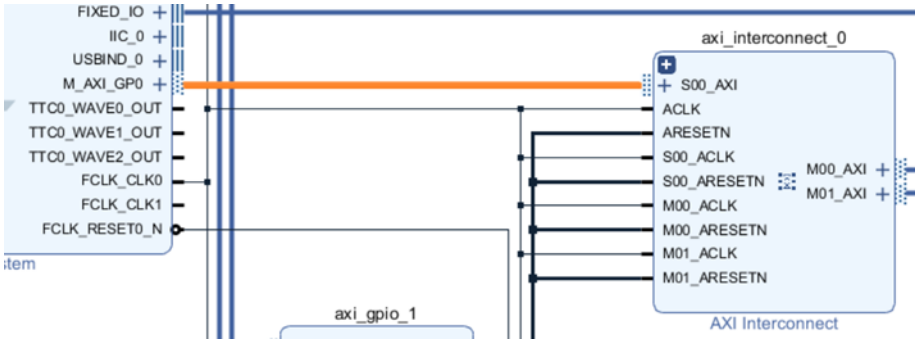


Figure 10. 12: Connecting the AXI interconnect block to the Zynq PS

Figure 10.12 shows how to connect the data bus from the *Processing System* to the *AXI interconnect*. It is advisable to include the *Processor System Reset block* to reduce the amount of warnings while synthesizing the design.

Include the Processing System Reset

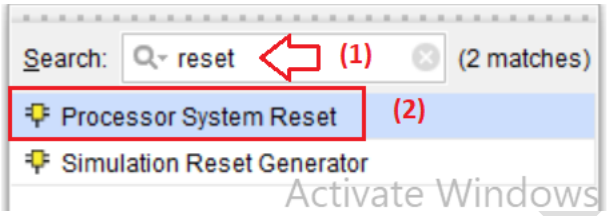


Figure 10. 13: Adding the Reset block

Write ***reset*** in the field and select the Processing System Reset by double clicking on it.

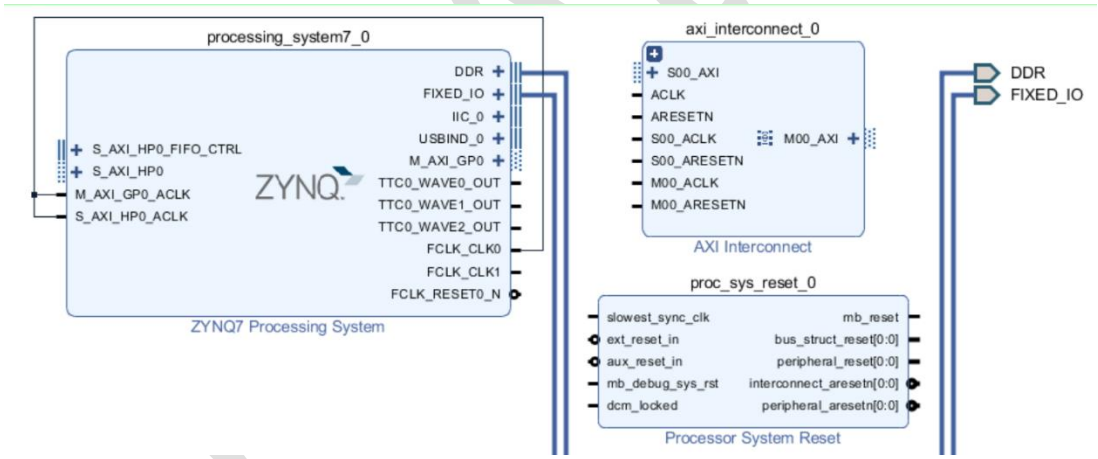


Figure 10. 14: The PS Reset block is part of the Block Design

Now it is time to wire the three blocks together. Start from the reset pins. Connect it to the reset output of the Processing System. This is shown in Figure 10.15.

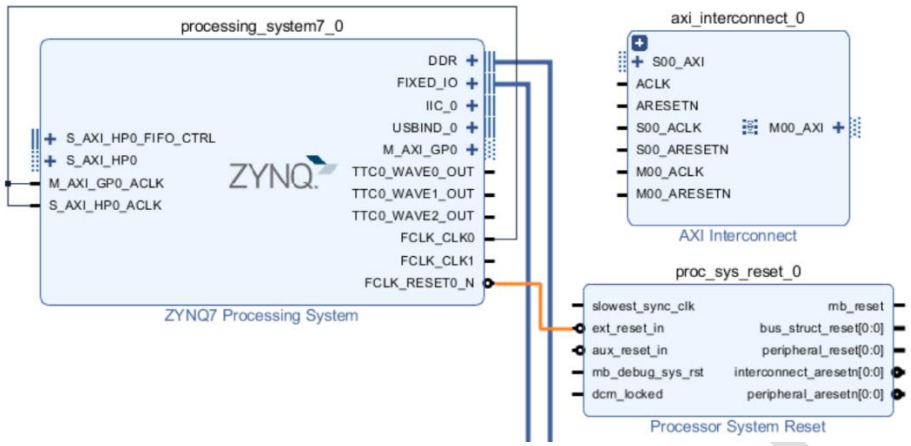


Figure 10. 15: Connecting the Zynq PS to the PS Reset Block

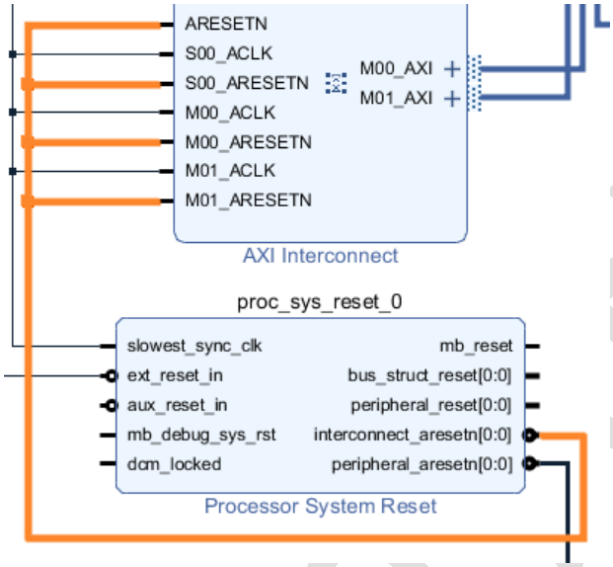


Figure 10. 16: Connecting the PS Reset with the AXI interconnect Block

Figure 10.16 shows all the reset inputs of the AXI interconnect block are connected to the *interconnect_aresetn [ ]* of the Processing System block. This makes sure that there will be minimal delay when resetting the system.

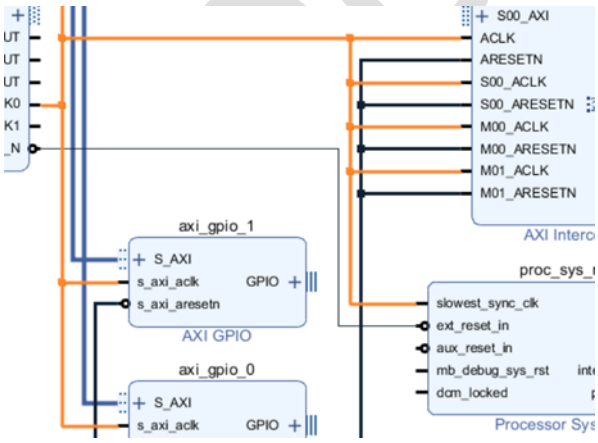


Figure 10. 17: Connecting a few of the clock signals

All clock signals should be connected to the same 100 MHz clock signal, emerging from the Processing System part. This ensures full synchronisation.

Include the AXI GPIO

The AXI GPIO will be used to interface the PS system with XADC. Another AXI GPIO will be used to interface the output pins located on the FPGA part of the SoC to the PS so that the PS part will drive the LEDs connected to the PL part of the Zynq Processing System. The LEDs should give a clear visual indication whether the ADC result is actually reflecting a change in the analogue voltage input.

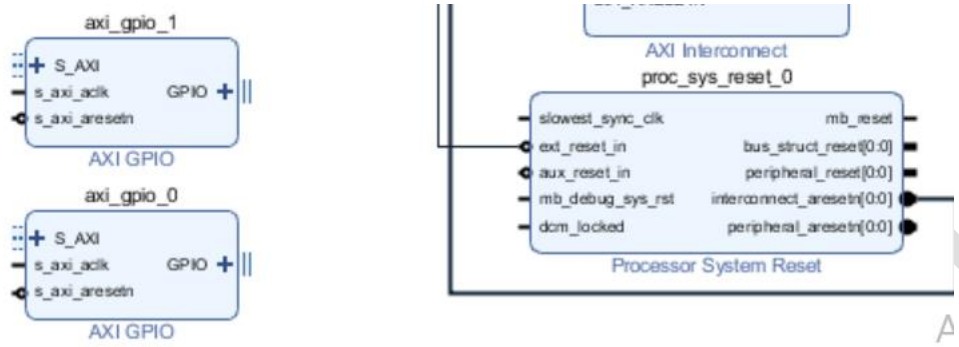


Figure 10. 18: Two AXI GPIO in the same diagram

Wiring the AXI GPIOs

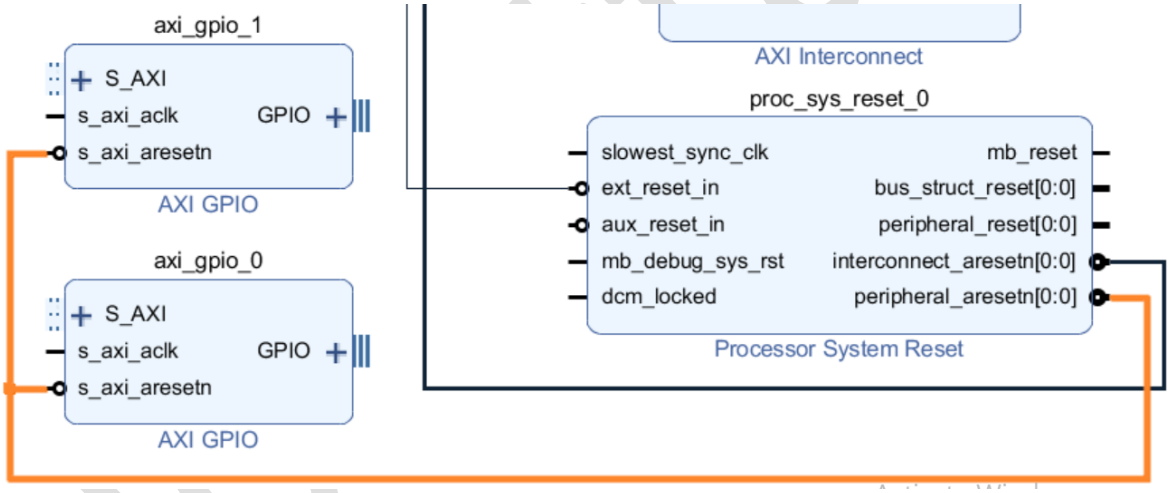
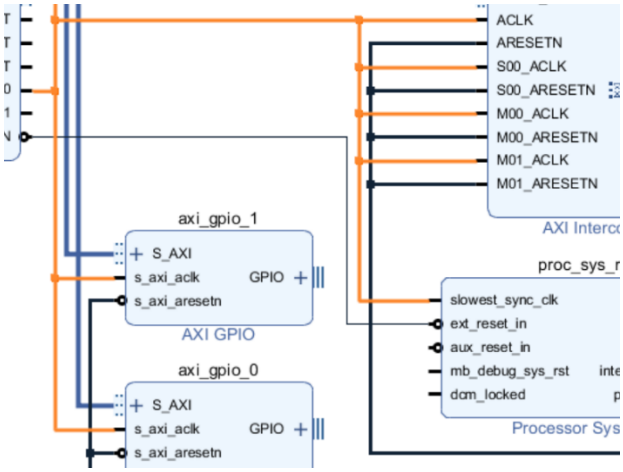


Figure 10. 19: Connecting the AXI GPIO Reset

First, make sure that the AXI GPIO block reset is connected to the *peripheral_aresetn[]* input.



Then connect the S_AXI_ACLK clock signal to all the common 100 MHz clock of the system.

Figure 10. 20: Connecting the clocks of the AXI GPIOs

Connecting the AXI GPIO data bus to the AXI interconnect

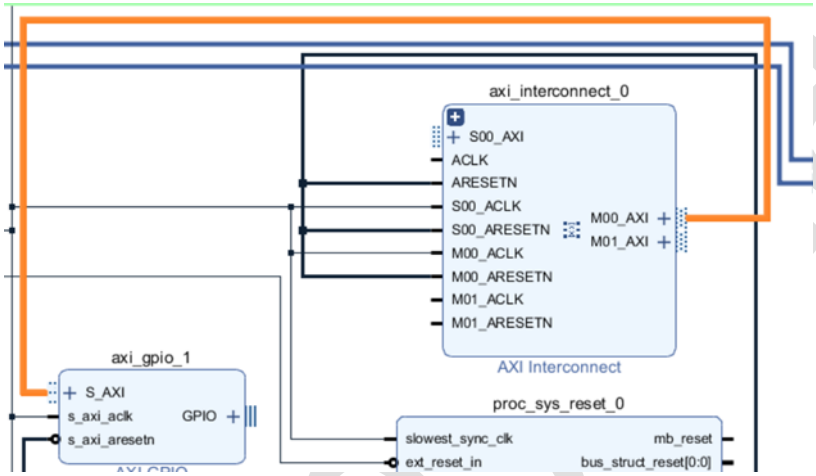


Figure 10. 21: Connecting the first AXI GPIO to the interconnect

This is the communication medium between the Processing System and the AXI GPIO block. It is all hidden from the designer!

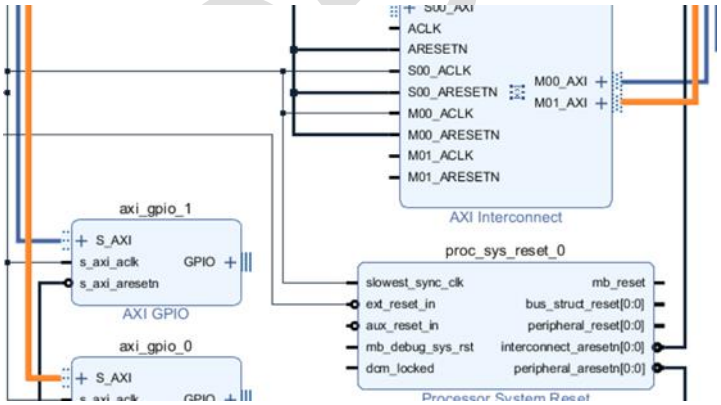


Figure 10. 22: Connecting the second AXI block

Both AXI GPIO blocks should be connected to the AXI interconnect block as shown in Figures 10.21 and 10.22.

Changing the width of the data busses of the AXI GPIOs

As discussed before, one of the AXI GPIO blocks will be used as an interface between the Processing System and the XADC, while the other AXI GPIO block be used to extend the external pinouts of the Processing System by using some of the external

pins allocated to the Programmable Logic side of the System-on-Chip. The following Figures show how to configure the AXI GPIO blocks for custom applications.

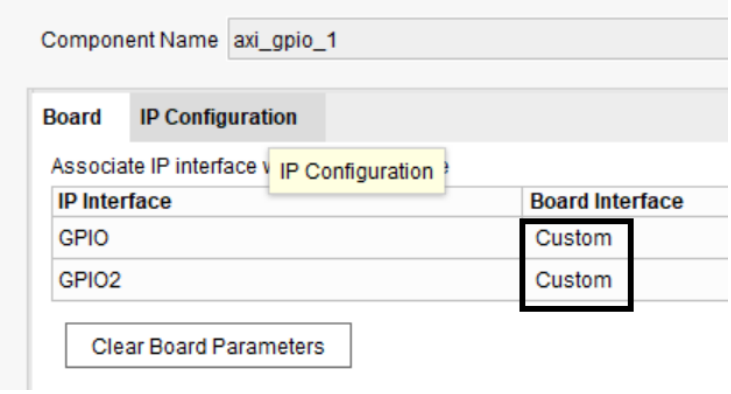


Figure 10. 23: Configuring the AXI GPIO 1

Leave the board interface as **custom.**

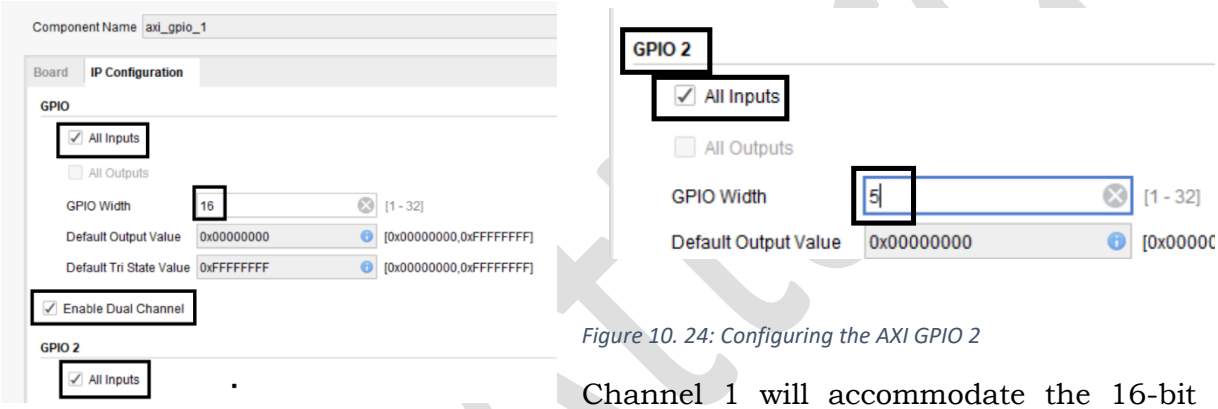


Figure 10. 24: Configuring the AXI GPIO 2

Channel 1 will accommodate the 16-bit ADC result from the XADC, while channel 2 accommodates the channel address from XADC.

So, Figure 26 above shows a single AXI GPIO block consisting of two channels. One of the channels is made up of 16 bits while the second channel is made up of 5 bits. Note that both channels are configured as inputs.

The second AXI GPIO will also be left as **custom**.

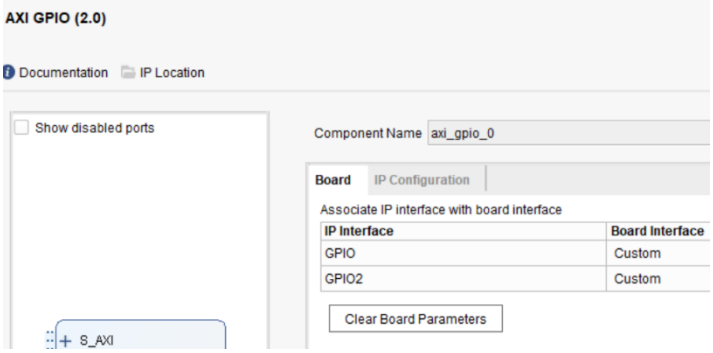


Figure 10. 25: Configuring AXI GPIO 3

Configuring the second AXI GPIO.

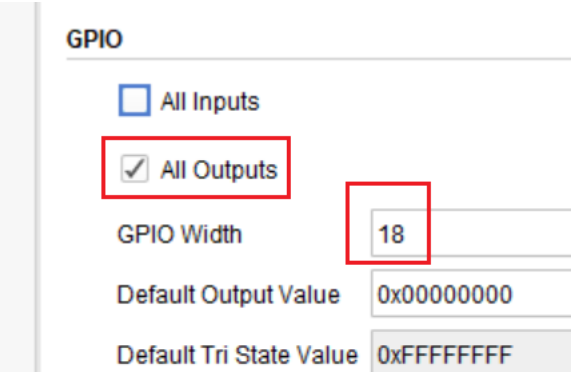


Figure 10. 26: Configuring AXI GPIO 2_2

The second AXI GPIO will drive 18 LEDs connected on the devBoard that designed specifically for the z-turn board. The second channel of this AXI GPIO is not going to be used.

<u>Including the XADC in the block design</u>

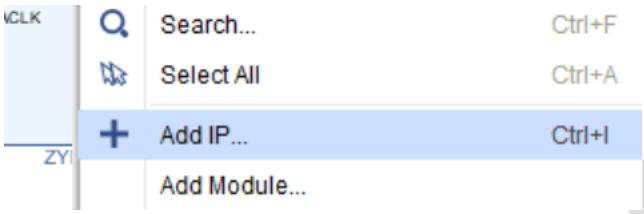


Figure 10. 27: Adding the XADC

Right-click anywhere on the canvas then choose *Add IP*.

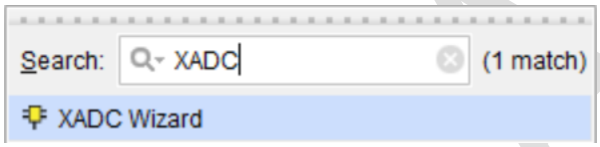


Figure 10. 28: Call XADC from List

Write XADC in the field provided and double click on *XADC* wizard.

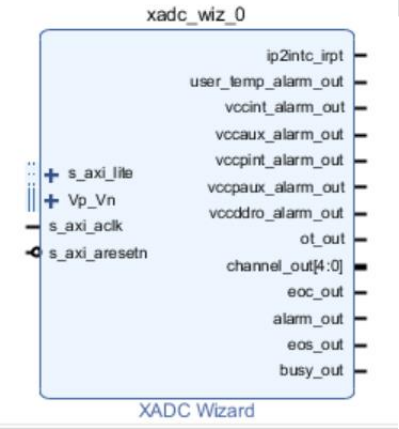


Figure 10. 29: The XADC block

This is the original XADC block. Double-clicking on it to configure it according to the needs of this application.

Also, the XADC mode of communication will be changed from *AXI4 lite* to *Dynamic Reconfiguration Port (DRP)* between the XADC block, the PS and PL.

Configuring the XADC block to be compatible with the software and hardware of this project

The BASIC page:

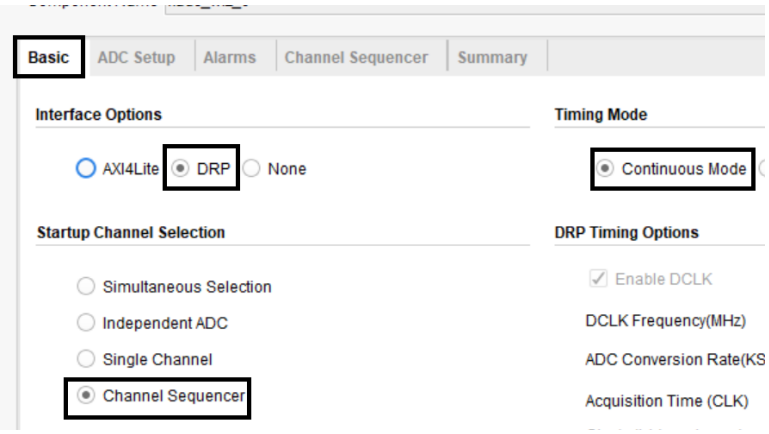


Figure 10. 30: XADC Basic Page

In the basic page:

- change the interface options to *DRP*
- leave the timing mode in *continuous mode*.
- Change the startup channel selection to *channel sequencer*

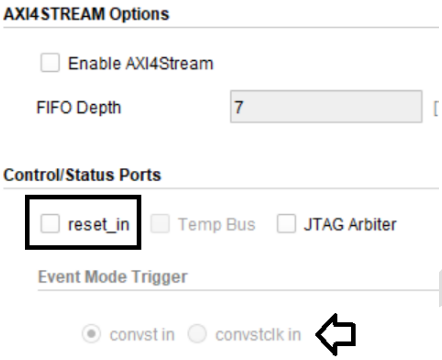


Figure 10. 31: XADC Configuration 2

- Leave the AXI4STREAM as is
- Remove the tick from ***reset_in*** box so that the XADC will be *free running*
- Note the Event Mode Trigger is not an option. This is because the continuous sampling mode was selected.

Analogue Sim File Option

Leave everything as is in the *Analogue Sim File* option section.

The ADC Setup Page

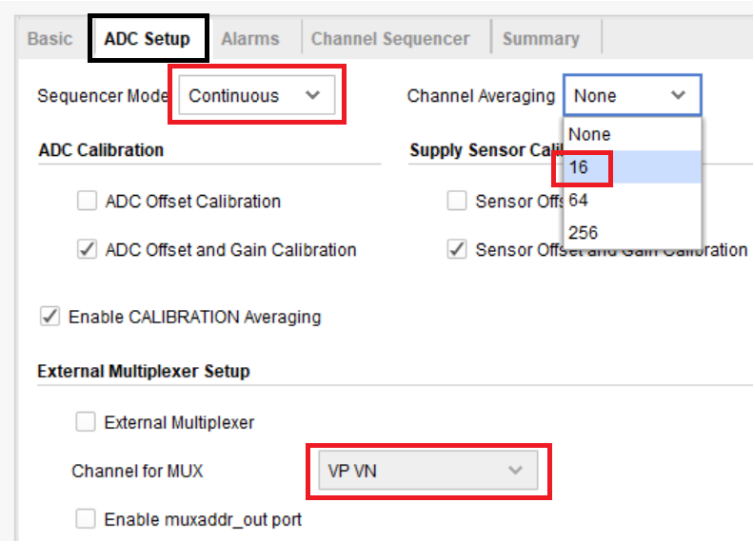


Figure 10. 32: The ADC Setup Page

In the ADC setup page, leave *sequencer mode* in *continuous* mode so that the XADC will operate in *free running* mode.

Opt for *averaging 16* and therefore XADC will sample and add 16 ADC results and output their average. This is very convenient because a low pass filter is created in hardware and the designer does not have to worry about it.

Leave the ADC calibration as is.

Leave channel for MUX as is.

The Alarms Page

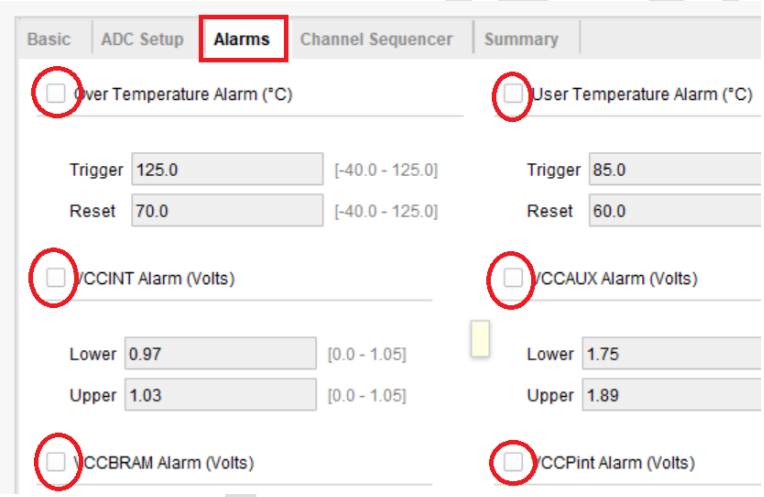


Figure 10. 33: XADC Alarms Page

Remove all the ticks in the alarms page. For this project the alarms are not needed.

The Channel Sequencer Page

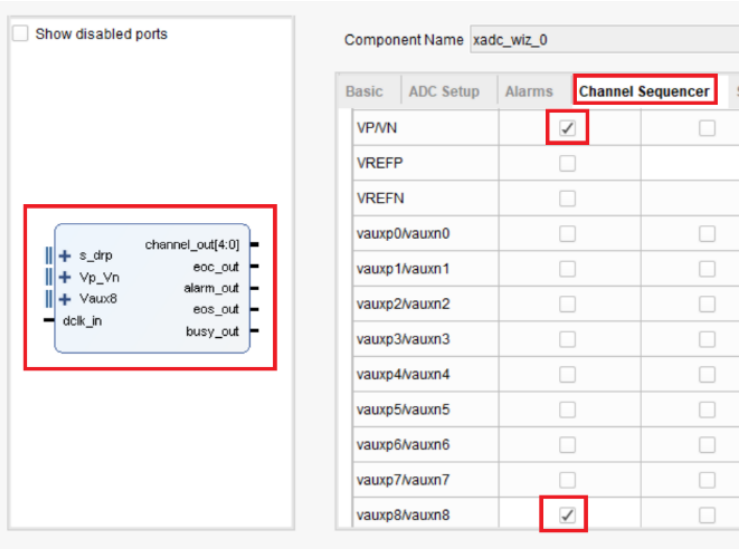


Figure 10. 34: XADC Channel Sequencer Page

In the channel sequencer page, the channels that are to be sampled must be selected.

Note in the left pane that the XADC block has reduced in size due to the changes we have done

Wiring the XADC

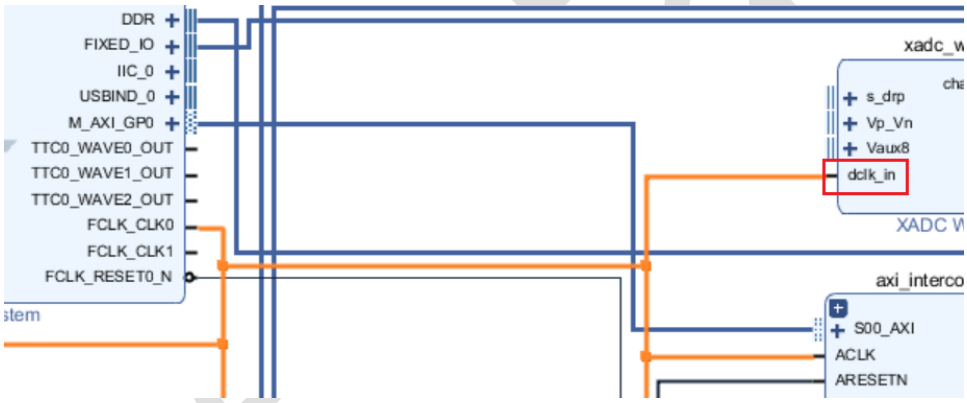


Figure 10. 35: Connecting the XADC Clock

Dclk_in should be connected to the 100 MHz clock because this is the default clock input for the 1MSPS can be achieved on *Vp/Vn analogue inputs*.

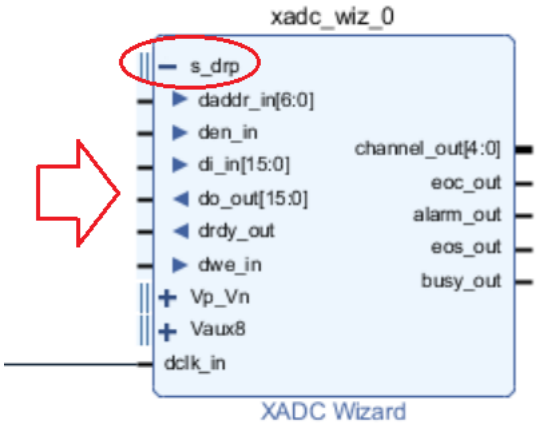


Figure 10. 36: Extending the DRP Bus

Hover the mouse over ***s_drp***. Notice two arrows pointing downwards. At that point left-click the mouse to reveal the DRP busses as shown in Figure 10.36

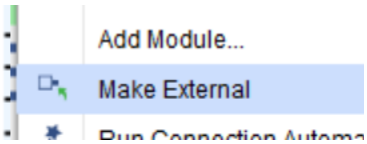


Figure 10. 37: Connecting the Vp/Vn to external pins

Vp/Vn will be connected to their dedicated external pin. To do this, hover the mouse on **Vp/Vn**, right-click and choose ***make external*** as shown in Figure 10.37. Do the same for **Vaux8**.

Now according to ***page 73 of UG480***, for XADC to operate in continuous mode, one must do the following connections:

- Connect **channel[4:0]** to **daddr_in[4:0]** – **daddr_in[6:5]** must be connected to logic **0**.
- Connect **d_en_in** with **eoc_out**
- Connect **drdy_out** with **dwe_in**

For point 1 above, the successful way to do it is to connect them to a VHDL module and concatenate "00" to bits 6:5 of daddr_in.

channeladdr_out <= "00" & ADCchannel;

where ***ADCchannel*** is connected to ***channel_out*** of the XADC block.

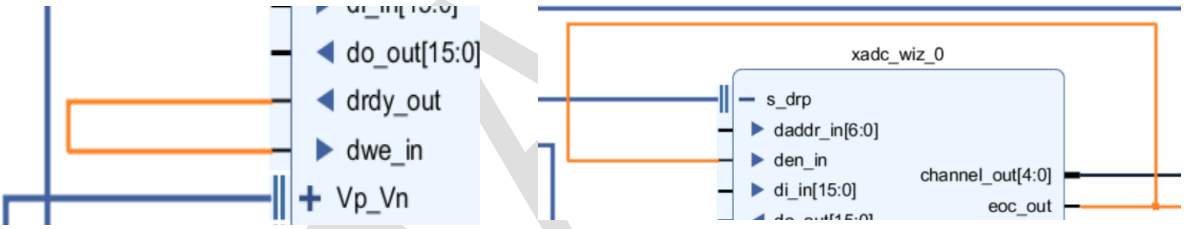


Figure 10. 38: Wiring the XADC

Figure 10.38 show the rest of the connections of the XADC block.

Connecting the ADC result bus to the AXI GPIO block

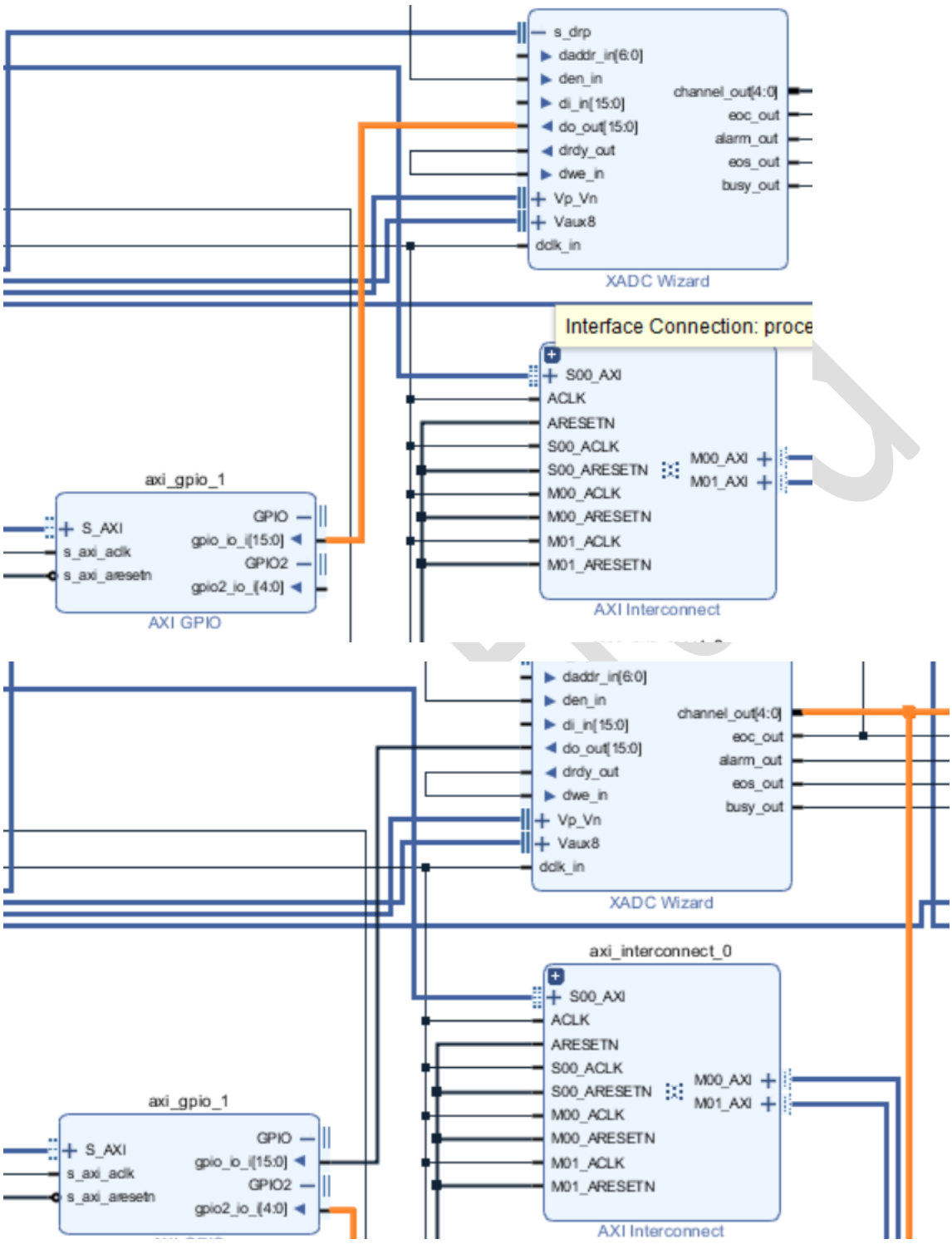
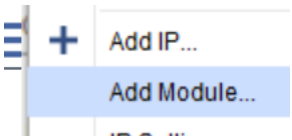


Figure 10. 39: Connecting the ADC result bus to the AXI GPIO Block

Including the VHDL module in the block design



Now, to include a custom VHDL module, right-click anywhere on canvas and select add module, the following pop up window appears:

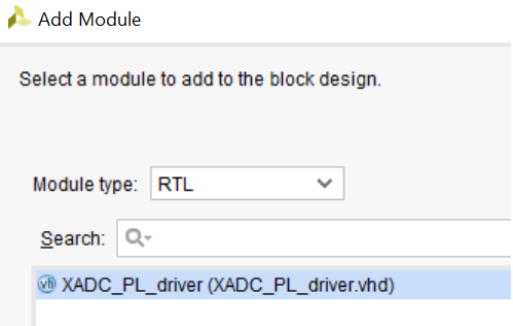


Figure 10. 40: Adding a VHDL Module to the Block Design

The VHDL modules that have passed the synthesis test after saving the code will appear in the list. If there are no modules in the list, then it means that the VHDL module has an error and one needs to rectify that error before the module will be available to be added in the block design.

Figure 10. 41: Connecting the HDL module to outside peripherals

Hover mouse on the respective pins, right click and then choose “make external”.

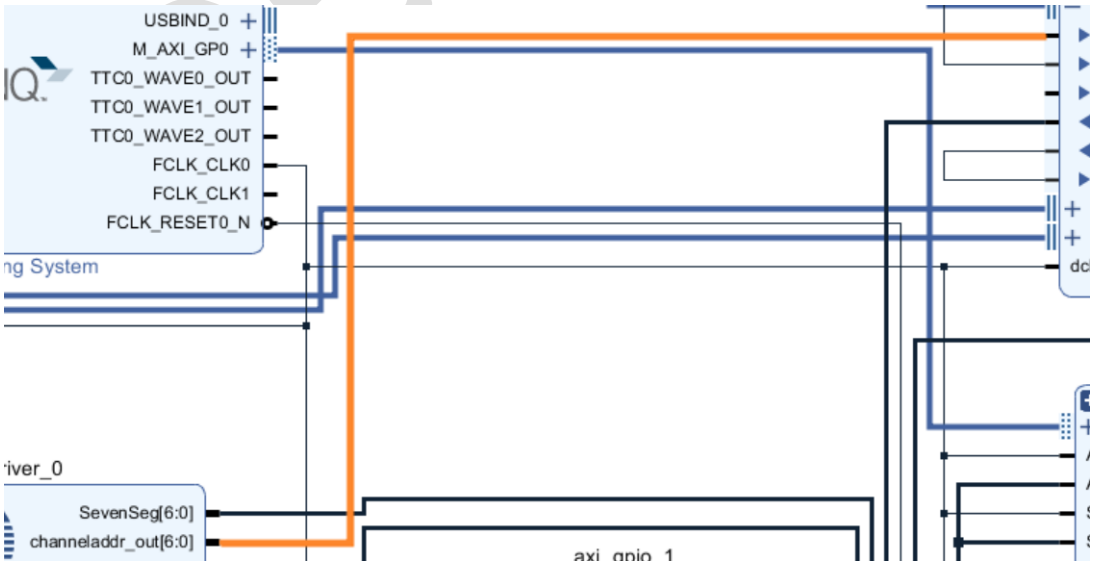


Figure 10. 42: Connecting the Channel Address Bus via the VHDL module

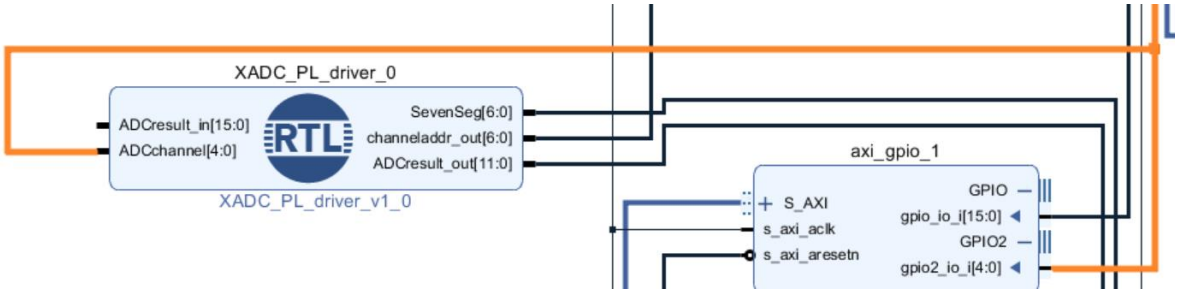


Figure 46: The channel address is input to the VHDL module

The ***ADCchannel*** bus is connected to ***channel_out*** of the XADC block. This will make sure that ***daddr_in[6:5]*** will be connected to *logic 0* and ***daddr_in[4:0]*** will be connected to ***channel_out*** and therefore the channel address is still 7 bits wide but only the first 5 bits are really selecting which channel is being sampled! This is specified on page 73 of UG480.

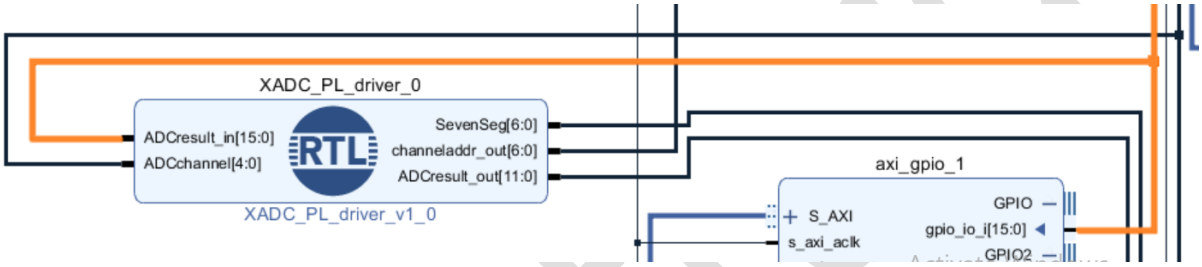


Figure 10. 43: ADC result bits shared between AXI GPIO and VHDL module

ADCresult is connected to ***do_out*** of *XADC*. This data will be shared with the PS via the AXI GPIO.

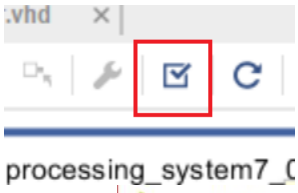
The architecture of the VHDL module is shown in the code snippet 10.1 below:

```
architecture Behavioral of XADC_PL_driver is

signal internalADCresult : integer;

begin
channeladdr_out <= "00" & ADCchannel;
internalADCresult <= to_integer(ADCresult_in(15 downto 4));

ADCresult_out <= std_logic_vector(ADCresult_in (15 downto 4)) when ADCchannel = "11000" else
                 "000000000000";

process(ADCchannel,internalADCresult,clk)
```

```
begin
if rising_edge(clk) then
   if ADCchannel = "00011" then
        if internalADCresult >= 0 and internalADCresult < 400 then sevenseg <= "0111111";      --0
        elsif internalADCresult >= 400 and internalADCresult < 800 then sevenseg <= "0000110"; --1
        elsif internalADCresult >= 800 and internalADCresult < 1200 then sevenseg <= "1011011";--2
        elsif internalADCresult >= 1200 and internalADCresult < 1600 then sevenseg <= "1001111";--3
        elsif internalADCresult >= 1600 and internalADCresult < 2000 then sevenseg <= "1100110";--4
        elsif internal integer lt >= 2000 and internalADCresult < 2400 then sevenseg <= "1101101";--5
        elsif internalADCresult >= 2400 and internalADCresult < 2800 then sevenseg <= "1111101";--6
        elsif internalADCresult >= 2800 and internalADCresult < 3200 then sevenseg <= "0000111";--7
        elsif internalADCresult >= 3200 and internalADCresult < 3600 then sevenseg <= "1111111";--8
        elsif internalADCresult >= 3600 and internalADCresult < 4096 then sevenseg <= "1100111";--9
        else sevenseg <= "0000000";
        end if;
  end if;
end if;
end process;

end Behavioral;
```

Code Snippet 10. 2: VHDL code

Validating the schematic



Click on the validate schema icon on the tool bar of the canvas

Figure 10. 44: Validate the Block Design

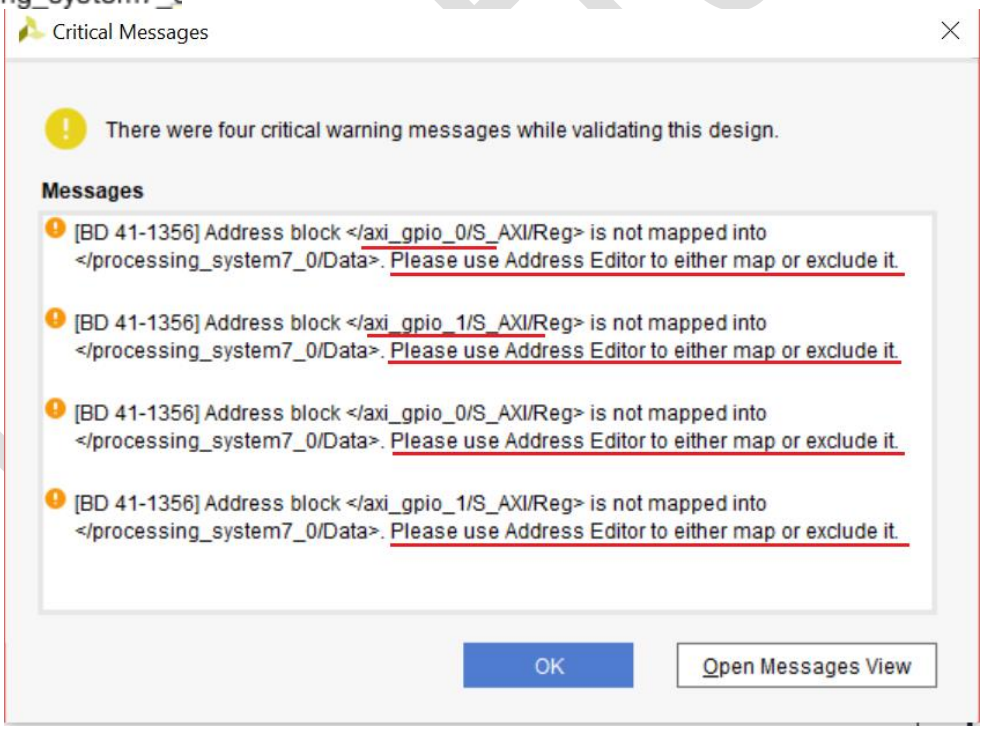


Figure 10. 45: Critical Warning that can be solved

To solve the above warnings, one has to follow the steps in Figure 10.46.

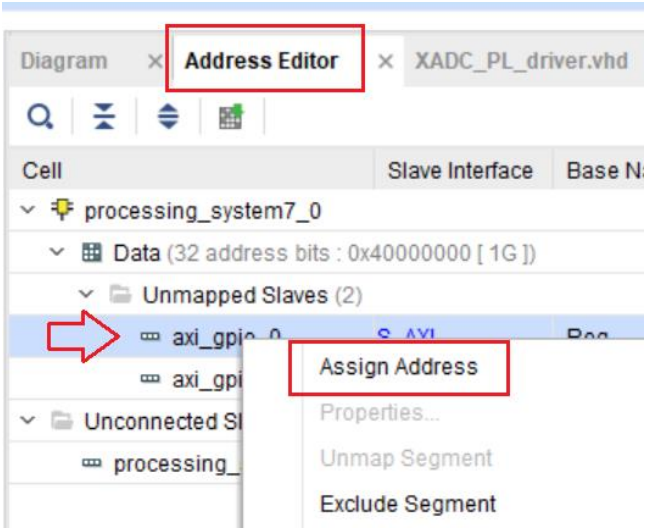


Figure 10. 46: The Address Editor

- Click on the ***address editor*** to reveal the problematic blocks.
- ***Highlight*** one of the blocks, then right click on it and choose ***Assign address***
- Do the same for the second block
- Figure 10.47 shows the assigned address of the AXI blocks

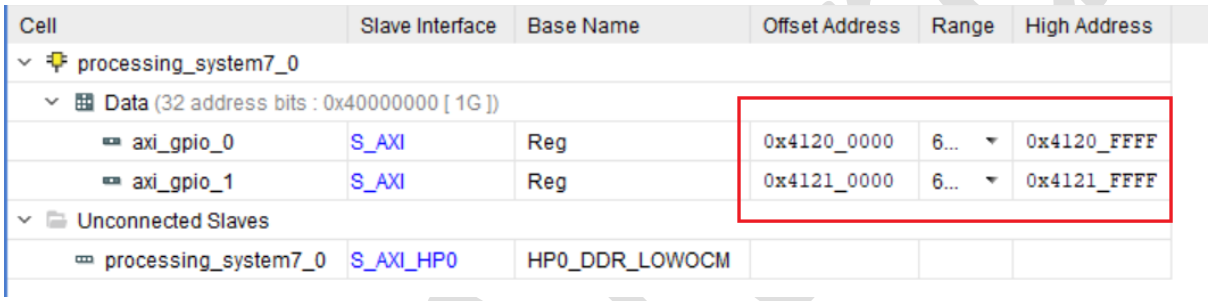


Figure 10. 47: The AXI GPIO are assigned an address

Create a Hardware Wrapper

Create a hardware wrapper for the block design. This will act like a top-level module.

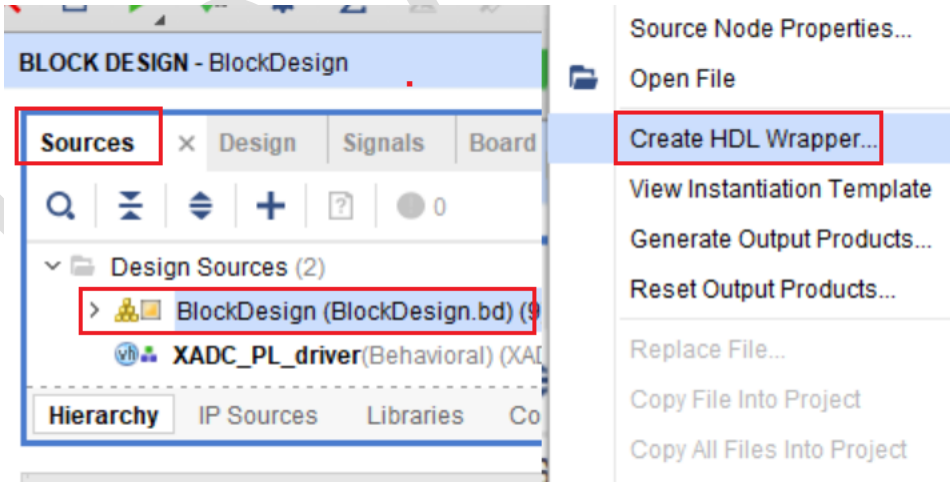


Figure 10. 48: Creating a Hardware Wrapper

Start synthesis

After the hardware wrapper is created, it is time to run ***synthesis***. Click on ***Run Synthesis*** on the left-hand-side of the IDE and click on ***OK*** for the following window.

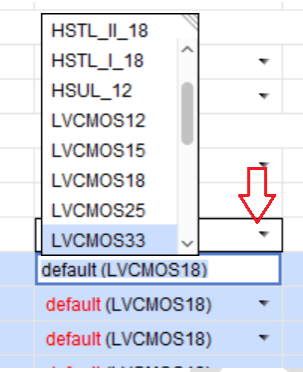
Before running implementation, it would be wise to assign the appropriate pin numbers to all external pins as shown in Figure 10.49.

gpio_io_o_0[17]	OUT							LVCMOS33*
gpio_io_o_0[16]	OUT							LVCMOS33*
gpio_io_o_0[15]	OUT							LVCMOS33*
gpio_io_o_0[14]	OUT							LVCMOS33*
gpio_io_o_0[13]	OUT							LVCMOS33*
gpio_io_o_0[12]	OUT							LVCMOS33*
gpio_io_o_0[11]	OUT							LVCMOS33*
gpio_io_o_0[10]	OUT							LVCMOS33*
gpio_io_o_0[9]	OUT							LVCMOS33*
gpio_io_o_0[8]	OUT				L16	✓	35	LVCMOS33*
gpio_io_o_0[7]	OUT				K19	✓	35	LVCMOS33*
gpio_io_o_0[6]	OUT				L19	✓	35	LVCMOS33*
gpio_io_o_0[5]	OUT				M17	✓	35	LVCMOS33*
gpio_io_o_0[4]	OUT				M19	✓	35	LVCMOS33*
gpio_io_o_0[3]	OUT				F16	✓	35	LVCMOS33*
gpio_io_o_0[2]	OUT				E18	✓	35	LVCMOS33*
gpio_io_o_0[1]	OUT				D19	✓	35	LVCMOS33*
gpio_io_o_0[0]	OUT				E17	✓	35	LVCMOS33*

Figure 10. 49Part of the pin assignments

Since there is not enough IOs to cover both PS and PL, half of them will be allocated to the PS part through the AXI GPIO and half of them to the PL part.

The single seven segment display is connected to the Programmable Logic part while the RS232 port will be used to send data to PC to confirm that PS is reading XADC data.



Note that all the voltages where changed to 3V3 instead of their default 1V8.

Notice the tick on the pins that were assigned a pinout!

Figure 10. 50: Changing the Operating Voltage to 3V3

SevenSeg_0 (7)	OUT					✓	35	LVCMOS33*
SevenSeg_0[6]	OUT				H20	✓	35	LVCMOS33*
SevenSeg_0[5]	OUT				G20	✓	35	LVCMOS33*
SevenSeg_0[4]	OUT				H18	✓	35	LVCMOS33*
SevenSeg_0[3]	OUT				K18	✓	35	LVCMOS33*
SevenSeg_0[2]	OUT				L17	✓	35	LVCMOS33*
SevenSeg_0[1]	OUT				F20	✓	35	LVCMOS33*
SevenSeg_0[0]	OUT				G18	✓	35	LVCMOS33*

Figure 10. 51: Assignment of Seven Segment Pins

Due to the changes made to the external configuration of the design, a new constraints file has to be created because now it differs from the standard constraints file included with the board support files. So, Figure 10.51 shows that a new constraints file will be created that will include the changes made.

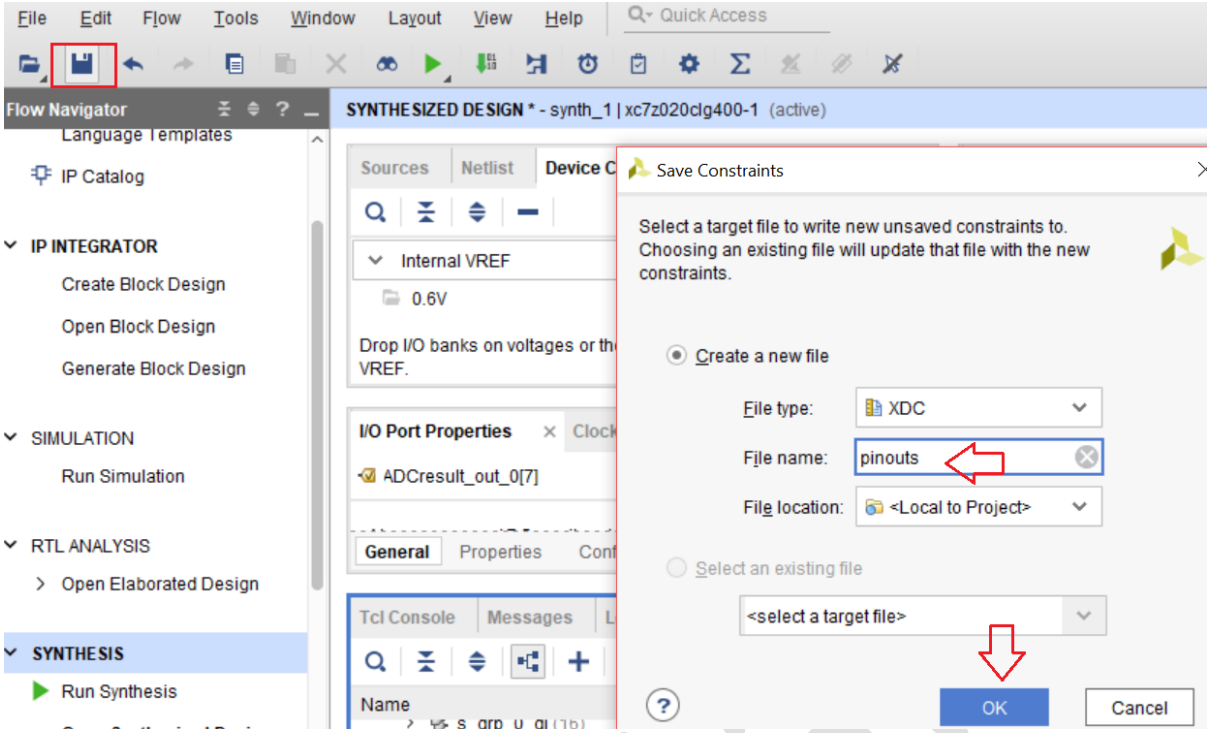


Figure 10. 52: Creating a new constraints file

After saving the new changes in the pinouts, the IDE will ask you to save to the new constraints file . Just give it a name and then click on *OK*.

Bitstream failure

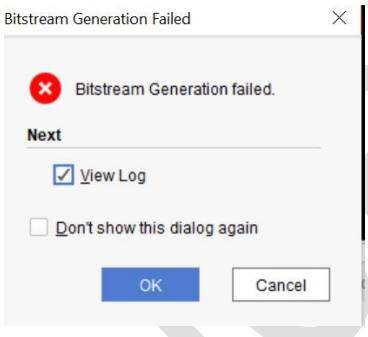


Figure 10. 53: Bitstream File Failure

So, the implementation run has passed successfully but the bitstream failed. Let's see the errors:

[DRC NSTD-1] Unspecified I/O Standard: 4 out of 189 logical ports use I/O standard (IOSTANDARD) value 'DEFAULT', instead of a user assigned specific value. This may cause I/O contention or incompatibility with the board power or connectivity affecting performance, signal integrity or in extreme cases cause damage to the device or the components to which it is connected. To correct this violation, specify all I/O standards. This design will fail to generate a bitstream unless all logical ports have a user specified I/O standard value defined. To allow bitstream creation with unspecified I/O standard values (not recommended), use this command: set_property SEVERITY {Warning} [get_drc_checks NSTD-1]. NOTE: When using the Vivado Runs infrastructure (e.g. launch_runs Tcl command), add this command to a .tcl file and add that file as a pre-hook for write_bitstream step for the implementation run. Problem ports: eos_out_0, eoc_out_0, busy_out_0, and alarm_out_0.

alarm_out_0	OUT				T19		34	LVCMOS33*
busy_out_0	OUT				P16		34	default (LVCMOS18)
eoc_out_0	OUT				P15		34	default (LVCMOS18)
eos_out_0	OUT				P18		34	default (LVCMOS18)

These must be changed to 3V3.

alarm_out_0	OUT				T19		34	LVCMOS33*
busy_out_0	OUT				P16		34	LVCMOS33*
eoc_out_0	OUT				P15		34	LVCMOS33*
eos_out_0	OUT				P18		34	LVCMOS33*

Figure 10. 54: Error Messages

Now for the second error:

[DRC UCIO-1] Unconstrained Logical Port: 38 out of 189 logical ports have no user assigned specific location constraint (LOC). This may cause I/O contention or incompatibility with the board power or connectivity affecting performance, signal integrity or in extreme cases cause damage to the device or the components to which it is connected. To correct this violation, specify all pin locations. This design will fail to generate a bitstream unless all logical ports have a user specified site LOC constraint defined. To allow bitstream creation with unspecified pin locations (not recommended), use this command: set_property SEVERITY {Warning} [get_drc_checks UCIO-1]. NOTE: When using the Vivado Runs infrastructure (e.g. launch_runs Tcl command), add this command to a .tcl file and add that file as a pre-hook for write_bitstream step for the implementation run. Problem ports: ADCresult_out_0[11], ADCresult_out_0[10], ADCresult_out_0[9], ADCresult_out_0[8], gpio_io_o_0[17], gpio_io_o_0[16], gpio_io_o_0[15], gpio_io_o_0[14], gpio_io_o_0[13], gpio_io_o_0[12], gpio_io_o_0[11], gpio_io_o_0[10], gpio_io_o_0[9], eos_out_0, eoc_out_0... and (the first 15 of 19 listed).

Figure 10. 55: The Second Error

This error was generated because not all external pins were assigned a physical IO pin. As the message suggests, create a .tcl file and pre-hook it to reduce this error into a warning.

So, click on File→ new file

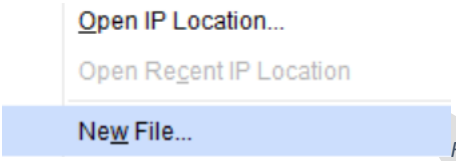


Figure 10. 56: Adding a .tcl File

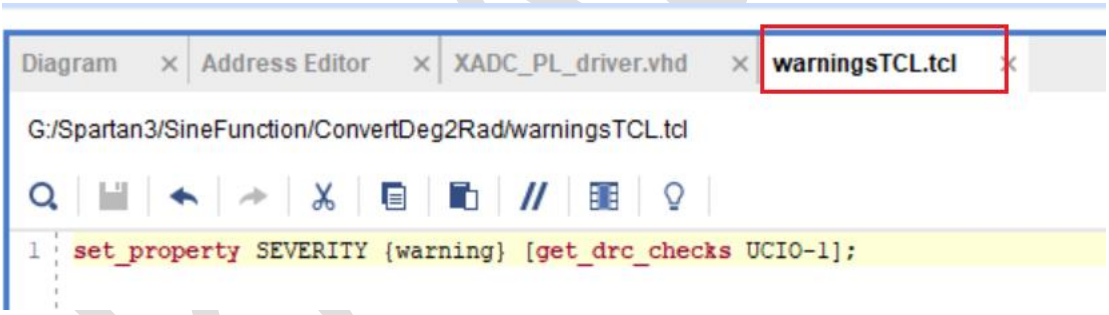


Figure 10. 57: Creating a .tcl File

The TCL file is an option under the ***new file*** selection in the ***File*** menu. The above statement in the TCL file was written by the author to reduce the errors into warnings for those pins who were not assigned any external IO pins.

Make sure that there is space between the braces and the square brackets and a semicolon at the end!

Now right-click on "generate Bitstream"

and click on Bitstream settings.

Figure 10. 58: TCL file affecting the bitstream generation

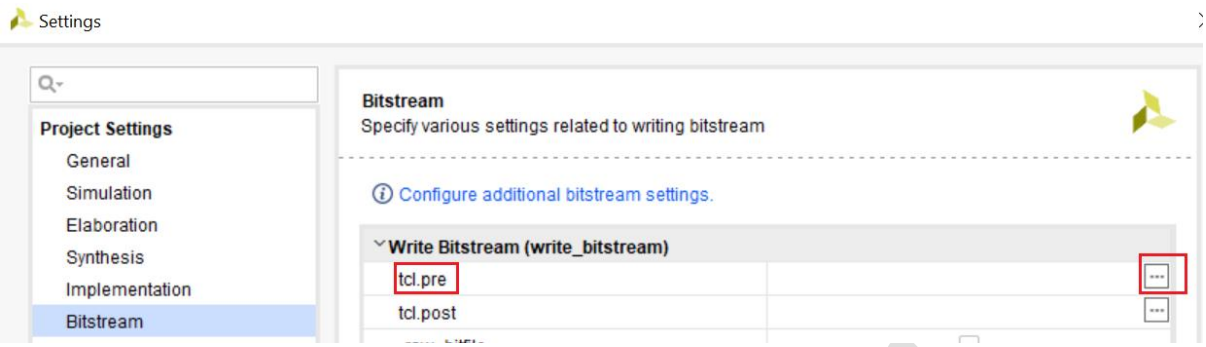


Figure 10. 59: Pre-Hooking the TCL File

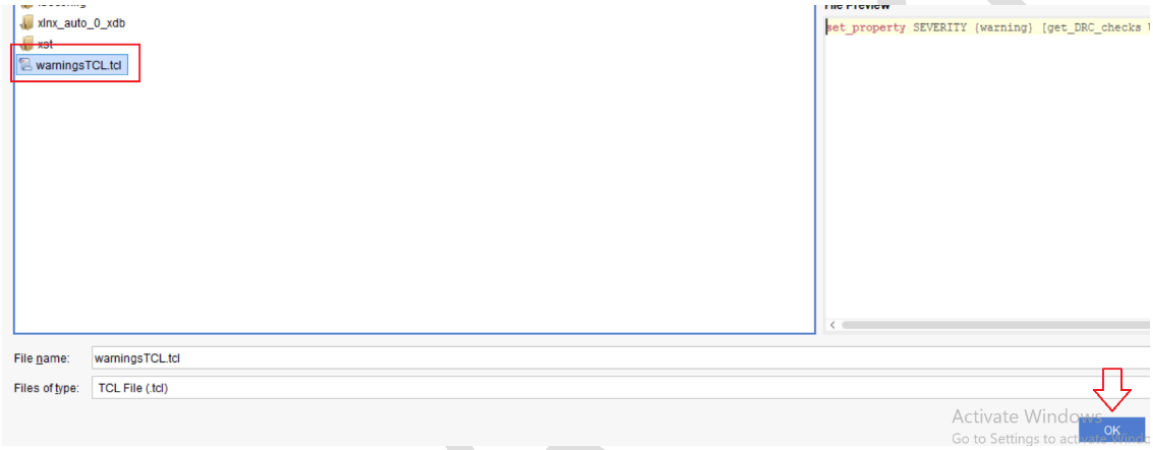


Figure 10. 60: Selecting the new TCL File

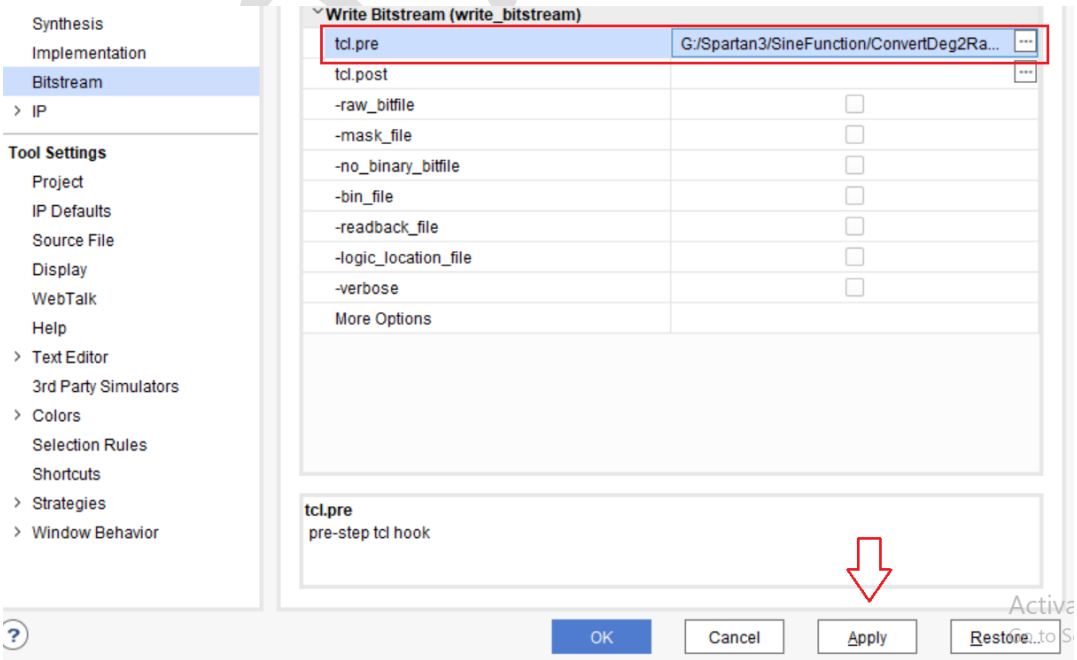


Figure 10. 61: confirming the TCL File

Do not forget to click on Apply! Then OK

Click on generate bitstream again and see what happens.

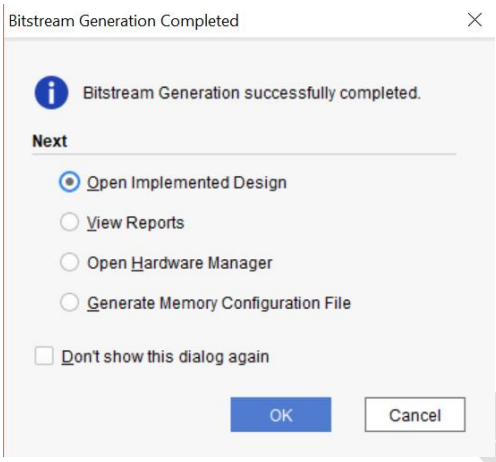


Figure 10. 62: Successful Generation of the Bitstream File

Now export the project to hardware including the bitstream file.

Exporting the hardware design

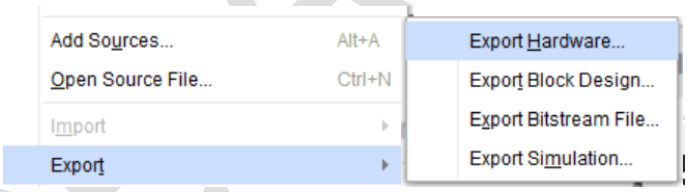


Figure 10. 63: Exporting the Hardware

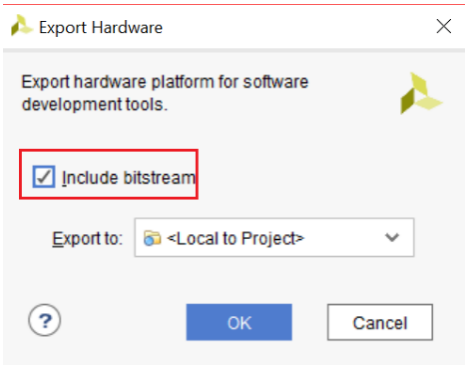
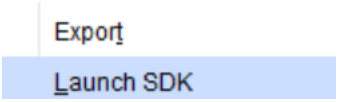


Figure 10. 64: Including the Bitstream File

Now launch SDK from within the Vivado IDE environment. ***File -> Launch SDK***



Click on OK for the following window.

Now create an FSBL project.

File → new → application project →

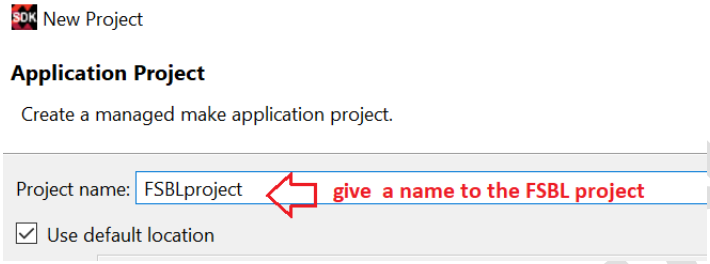
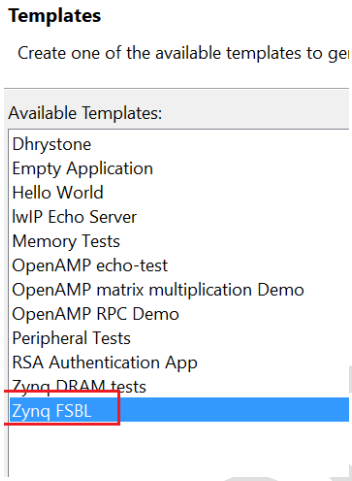


Figure 10. 65: Naming the FSBL project

Click on ***NEXT*** underneath.



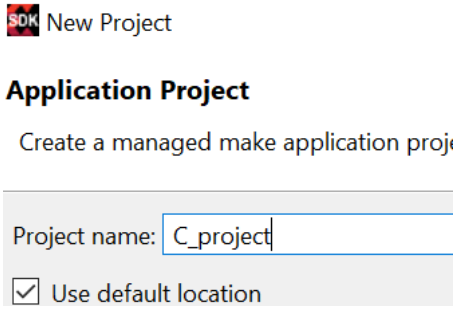
Choose Zynq FSBL from list and then click on FINISH underneath.

Make sure that you allow SDK to create the work environment.

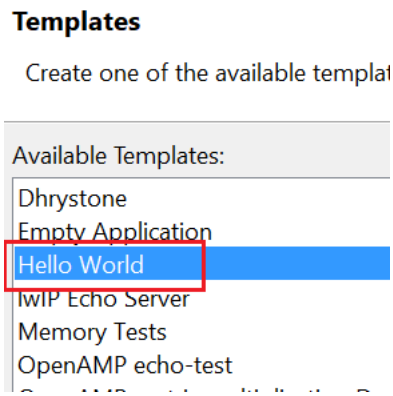
Figure 10. 66: Choosing the FSBL Project

Now let's create a C project

File → new → application project → give a name to the project



Click on ***NEXT***



This time select ***hello world*** from the list and click on ***FINISH***.

Figure 10. 67: Selecting the C project

The SDK will add the C project to the Vivado project.

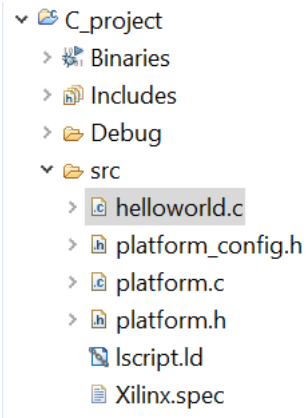


Figure 10. 68: Locating the Hello World C program

- Include the AXI GPIO library #include "***xgpio.h***"

The xgpio library is in libsrc folder under ps7_cortexa9_0

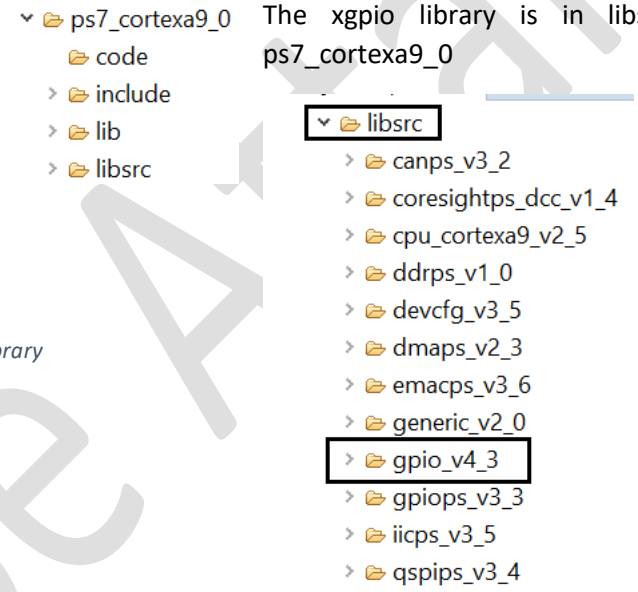


Figure 10. 69: Locating the xgpio library

AXI GPIO block library for the PS part is shown in Figure 10.69. This is different from the ***gpiops*** Library so watch out!

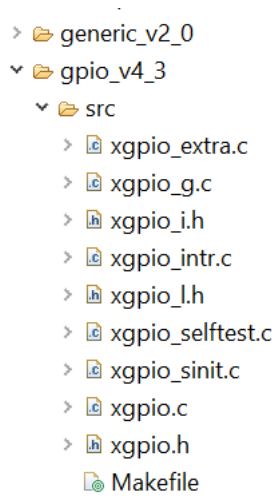


Figure 10. 70: The AXI GPIO Library

Initializing the AXI GPIO

In ***xgpio_sinit.c*** file copy the lookup().

XGpio_Config* *XGpio_LookupConfig(u16 DeviceId)

U16 DeviceId can be found in ***xgpio_g.c*** file. There can be ***only two*** instances of AXI GPIO. If the application needs three then one has to see whether there is a workaround. The above function call is changed to:

AXIgpio1Ptr = XGpio_LookupConfig(XPAR_AXI_GPIO_0_DEVICE_ID);

Then write the function call:

int XGpio_CfgInitialize(XGpio * InstancePtr, XGpio_Config * Config,UINTPTR EffectiveAddr)

converted to:

AXIgpio1success=XGpio_CfgInitialize(&AXIgpio1,AXIgpio1ConfigPtr,AXIgpio1ConfigPtr->BaseAddress);

AXIgpio1success is of type ***int***.

Now use the returned variable value to check whether the initialization has been successful or not.

```
if(AXIgpio1success != XST_SUCCESS)
    {
        return XST_FAILURE;
    }
```

Usually if there is a failure here, the program will stop running here.

Repeat the same instructions to AXI GPIO 2.

```
int main()
{
    XGpio_Config *AXIgpio0ConfigPtr;
    XGpio_Config *AXIgpio1ConfigPtr;
    int AXIgpio0success,AXIgpio1success;
    XGpio AXIgpio0,AXIgpio1;

    init_platform();

    /* Initialise AXI GPIO 0*/
     AXIgpio0ConfigPtr = XGpio_LookupConfig(XPAR_AXI_GPIO_0_DEVICE_ID);
     AXIgpio0success = XGpio_CfgInitialize( &AXIgpio0,AXIgpio0ConfigPtr,AXIgpio0ConfigPtr
             ->BaseAddress);
     if(AXIgpio0success != XST_SUCCESS)
     {
         return XST_FAILURE;
     }

     /* Initialise AXI GPIO 1*/
     AXIgpio1ConfigPtr = XGpio_LookupConfig(XPAR_AXI_GPIO_1_DEVICE_ID);
          AXIgpio1success = XGpio_CfgInitialize( &AXIgpio1,AXIgpio1ConfigPtr,AXIgpio1Config
                 ->BaseAddress);
          if(AXIgpio1success != XST_SUCCESS)
          {
             return XST_FAILURE;
          }
```

Code Snippet 10. 3: Initializing the AXI GPIOs

Now set the direction of ***each channel*** in each AXI GPIO block.

void XGpio_SetDataDirection(XGpio *InstancePtr, unsigned Channel, u32 DirectionMask)

The above function is found in ***xgpio.c*** file. ***0*** means that particular bit is an output while ***1*** means that particular bit as input.

```
/*All of AXI GPIO 0 is set as outputs*/
XGpio_SetDataDirection(&AXIgpio0,1,0x00000000);

/* All of AXI GPIO 1 are set as inputs*/
XGpio_SetDataDirection(&AXIgpio1,1,0xFFFFFFFF);
XGpio_SetDataDirection(&AXIgpio1,2,0xFFFFFFFF);
```

Code Snippet 10. 4: Port Direction of each AXI GPIO

The ***channel*** number can be either channel 1 or channel 2 within each AXI block.

Now read from **channel 2 of AXI GPIO 1** to know which ADC channel is giving the ADC result from ***channel 1 of AXI GPIO 1*** and then output the value in ***channel 1 of AXI GPIO 2***.

u32 XGpio_DiscreteRead(XGpio * InstancePtr, unsigned Channel)

the above function returns a value of type ***u32.***

ADCchannel = (XGpio_DiscreteRead(&AXIgpio1,2) & 0x0000001F);

now to write to a channel in one of the AXI GPIOs use:

void XGpio_DiscreteWrite(XGpio * InstancePtr, unsigned Channel, u32 Data)

```c
while(1)
{
   ADCchannel = (XGpio_DiscreteRead(&AXIgpio1,2) & 0x0000001F);
   /* the ADC channel is 5 bits wide*/
   if (ADCchannel == 0x00000003) //reading Vp/Vn "00011"
   {
      ADCresultVp = ((XGpio_DiscreteRead(&AXIgpio1,1) & 0x0000FFF0) >> 4);
      /* result is stored between 15:4 */
      XGpio_DiscreteWrite(&AXIgpio0, 1, ADCresultVp);
      printf("Vp ADC result is:%d",(int)ADCresultVp);//type cast u32 to int
   }
   else if (ADCchannel == 0x00000018) //reading Aux8 "11000"
   {
       ADCresultVAux8 = ((XGpio_DiscreteRead(&AXIgpio1,1) & 0x0000FFF0) >> 4);
      /* result is stored between 15:4 */
      XGpio_DiscreteWrite(&AXIgpio0, 1, ADCresultVAux8);
      printf("Aux8 ADC result is:%d",(int)ADCresultVAux8); //type cast from u32
   }                                                        //to int type

}
```

Code Snippet 10. 5: Reading and Writing from AXI GPIO

In the code above, the ***read function*** together with ***bit-masking*** was used so that if there are any other 1s which are not of interest will be removed. So, read the channel number first, then use it as a reference to know which ADC channel is transmitting data at the output. Send them on UART and at the same time display the result on LEDs through the AXI GPIOs to access the pins located on the PL side.

Reconfiguring the Board Support Package

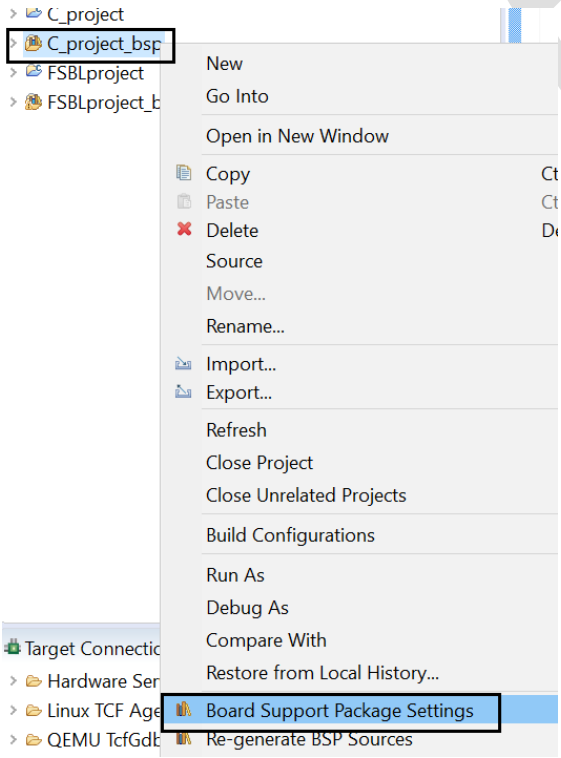
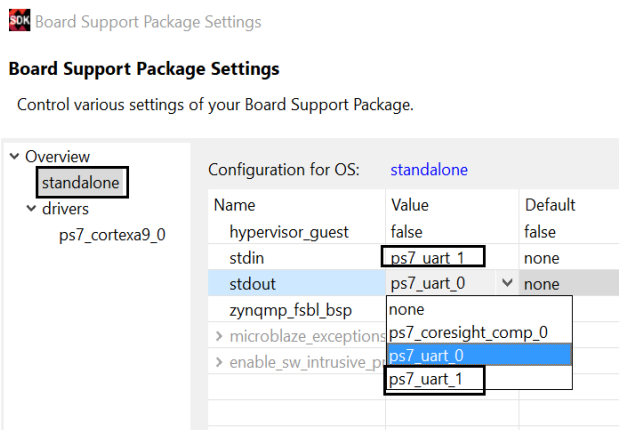
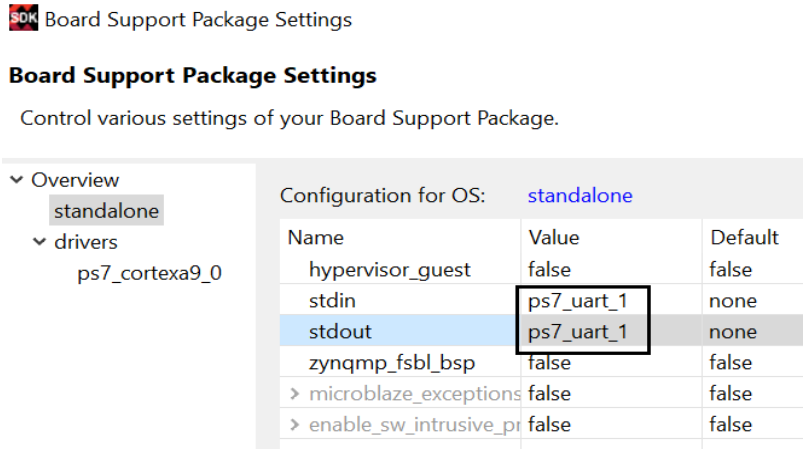


Figure 10. 71: Board Support Package

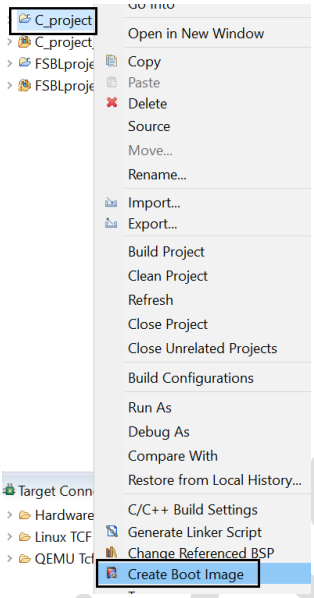
To be able to communicate with PC, ***UART 1*** must be made as the default UART not UART 0.





Make sure to wait for the update to take place because it takes a few minutes.

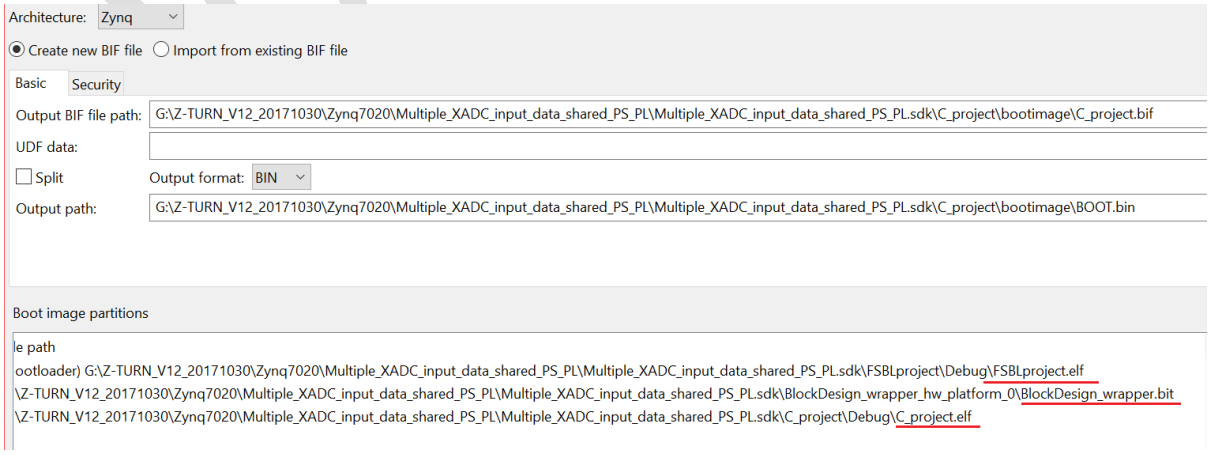
Creating the Boot Image File



The Zynq 7 can only be programmed in two ways, either using a JTAG cable or via the ARM Cortex A9 by loading a bootloader file on SD card. The ARM A9 reads the boot image file from the SD card and loads the PL part of the Zynq SoC.

Right click on the C project and select ***create boot image***.

Figure 10. 72: Creating the Boot image file



Make sure that there are ***three files*** in the ISO file. Click on ***Create boot image*** underneath.

Now look for the generated boot image file in the appropriate folder. Copy and paste on the SD card.

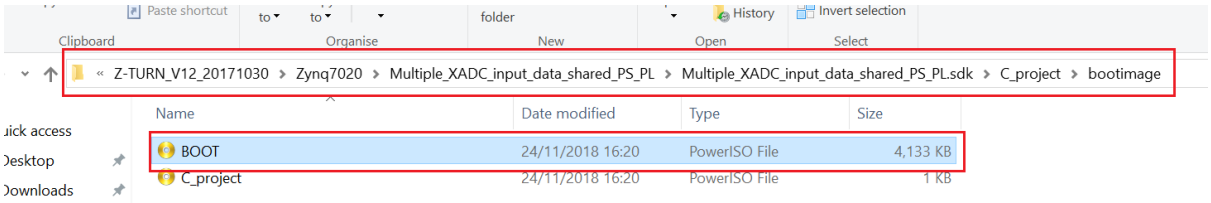


Figure 10. 73: Locating the BOOT image file in File Explorer within Windows

Event driven sampling of multiple XADC channels from the Programmable Logic

The aim of this chapter is to sample XADC channels using event driven technique from a VHDL module. Using this technique, the programmer has full control of the XADC block and therefore could determine which ADC channel to sample and when the ADC result is available.

In previous chapters, the XADC was configured to do continuous sampling and all that was needed from the designer's side is to sample the channel address available at the output of the XADC block and route the ADC data on the ***do_out*** bus to the output port. This is convenient, it is challenge to write an XADC driver that is able to operate the XADC block in event-driven mode.

During experimentation, it was discovered that at power-on-reset, **the XADC needed some finite time to settle, before the first sample.** This should be in the form of a small delay of 100ms.

End-of-Conversion Signal

In previous code the ***DRDY*** signal was being monitored while the ***EoC*** signal was not. This was recommended by ***UG480 on page 74 section Dynamic Reconfiguration Port Configuration (DRP)***. However, taking a closer look at the timing diagram of the **Event-Driven Sampling**, one will soon realize that it was ***imperative*** to check ***EoC*** signal. By doing so, the XADC was given enough time to finish the previous conversion and start a new sample. UG480, shows that the XADC needs ***four clock cycles between conversions*** especially if the next conversion is going to be done from a different channel.

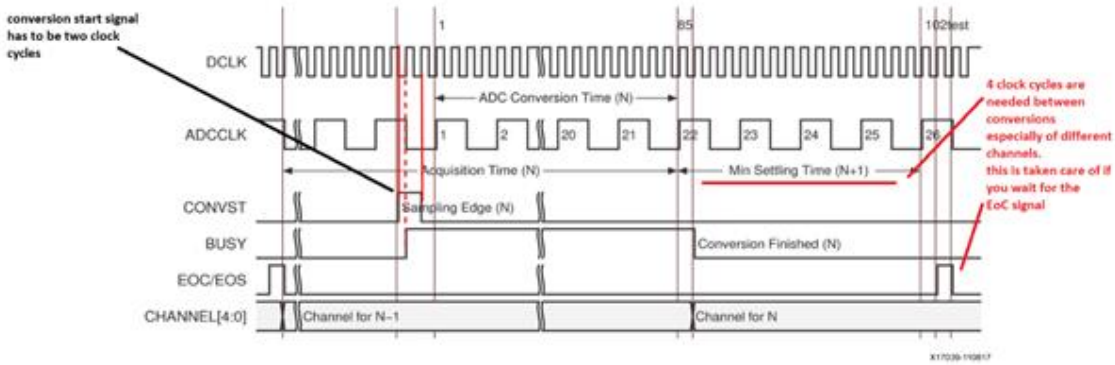


Figure 11. 1: Event Driven Mode Timing Diagram

CONVST signal

From the timing diagram it is clearly seen that ***two clock cycles*** are needed for ***CONVST*** signal to be effective. This gives time for the ***BUSY*** signal to become ***logic 1***.

- So, check the ***BUSY*** signal to be at logic 1 after driving the ***CONVST*** signal to logic low.
- Make sure that the ***BUSY*** signal goes low again.

- Then wait for the EoC signal to go high.

Now read the channel ADC data from the data bus. This is described in the next section.

Reading ADC data from the data bus

Now to read the ADC result from XADC, one must go through the following steps according to timing diagram of Figure11.2. This was taken from page 75 of UG480:

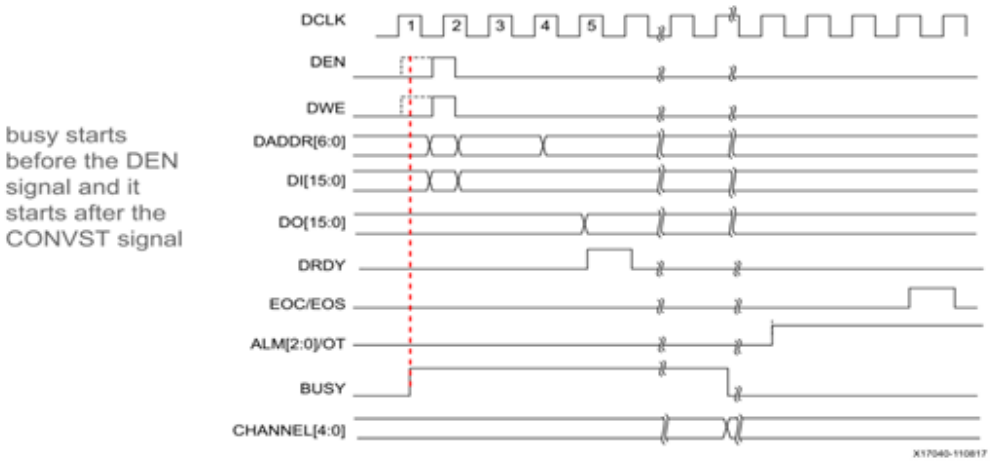


Figure 11. 2: DRP Timing Diagram

So, after the ***EoC*** is asserted, it is time to read the ADC result by following these steps:

- Assert the ***D_en*** signal to logic 1 for one clock cycle
- The ***D_we*** input of the XADC should be hardwired to ground
- Wait for ***DRDY*** signal of XADC to go high
- Get the ADC data

Obviously the above must be implemented in a state machine, and therefore one must make sure to store the ADC result in a register.

Since more than one XADC channel must be read, with my VHDL code, another state-machine was created to controls which channel the controller will sample. The following sections show the block diagram and explain the VHDL code.

The Block Diagram

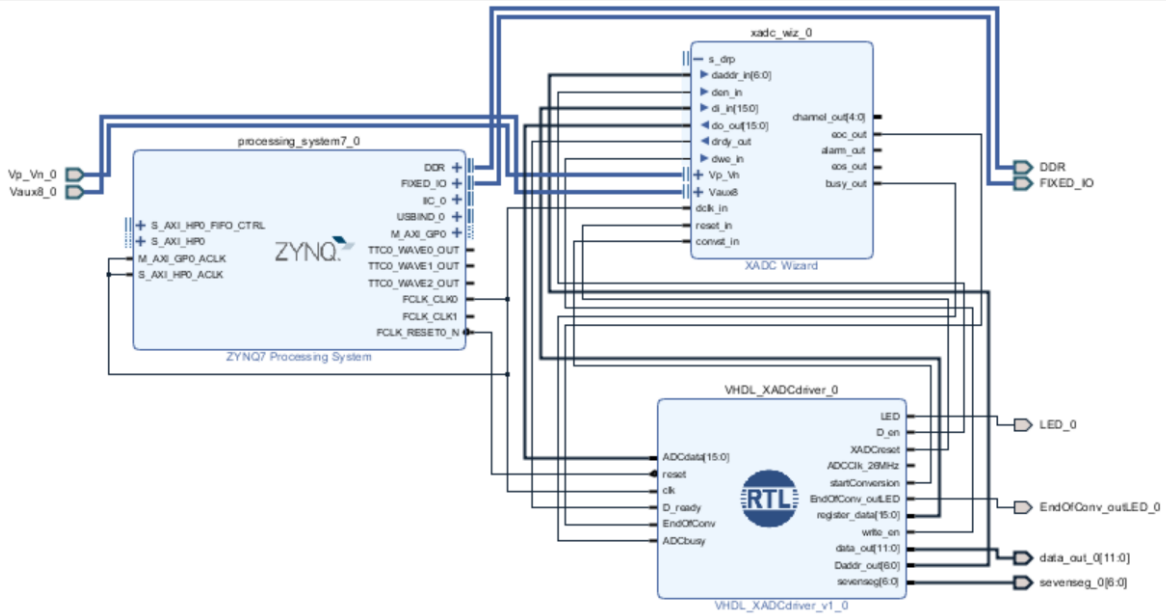


Figure 11. 3: System Block Diagram

One should go through all the steps described in previous chapters to draw the circuit as shown in Figure 11.3. The XADC configuration will be explained next.

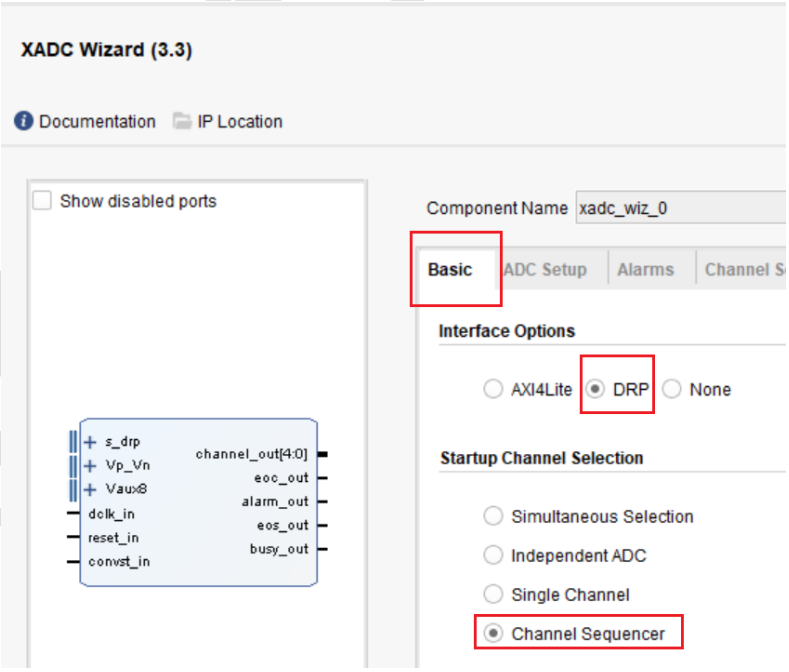


Figure 11. 4: XADC Basic Configuration Page

After double-clicking the XADC wizard block, the *basic* page pops up. Enable the *DRP* radio button and the Channel Sequencer button as shown in Figure 11.4. Scroll the horizontal bar to the right to reveal the *Timing Mode* section as shown in Figure 11.5. Tick the *Event Mode* radio button. Leave the frequencies as they are.

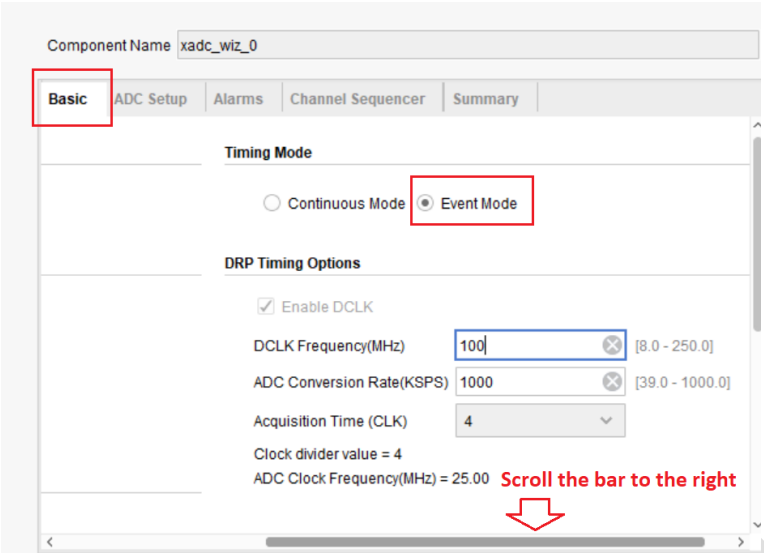


Figure 11. 5: Enable the Event Mode

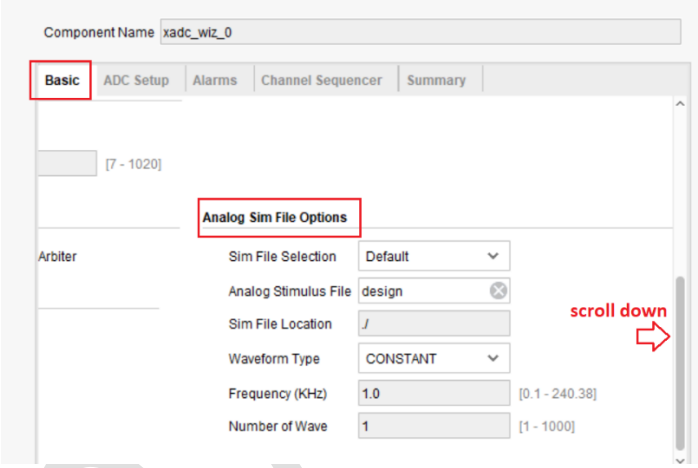


Figure 11. 6: Analog Sim File Options

Leave the Analog Sim File Options section as it is.

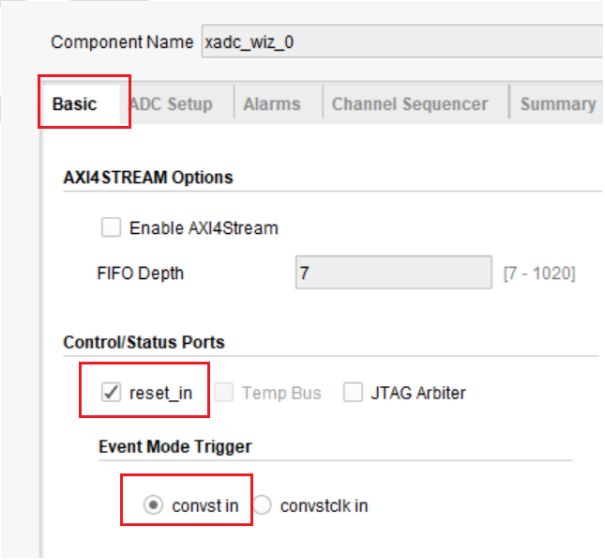


Figure 11. 7: Event Mode Settings

Leave the *AXI4STREAM* section as it is but make sure to tick the *reset in* square and the *convst in* radio button.

Leave the ADC setup page as it is.

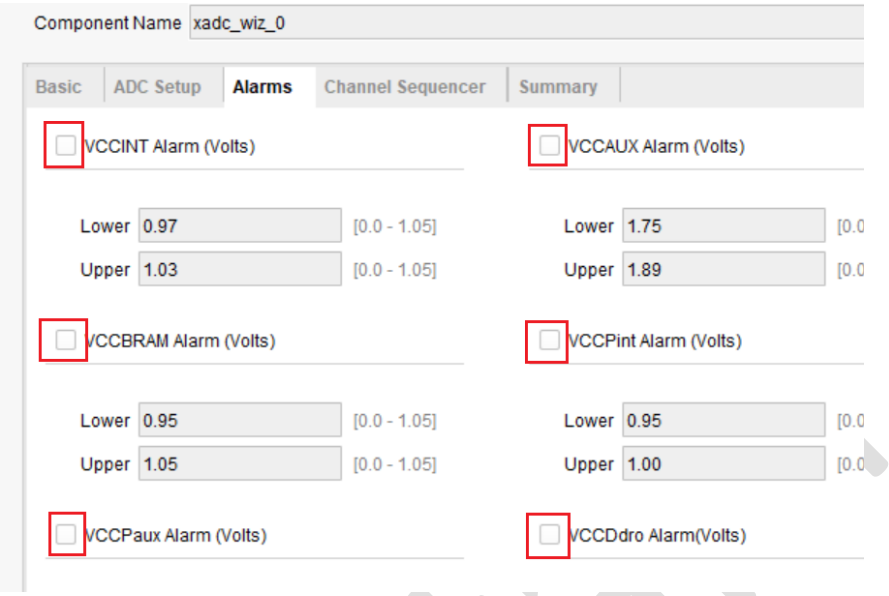


Figure 11. 8: Alarms Setup page

Remove all the ticks in the alarms setup page as shown in Figure 11.8. make sure to scroll down and disable the remaining alarms.

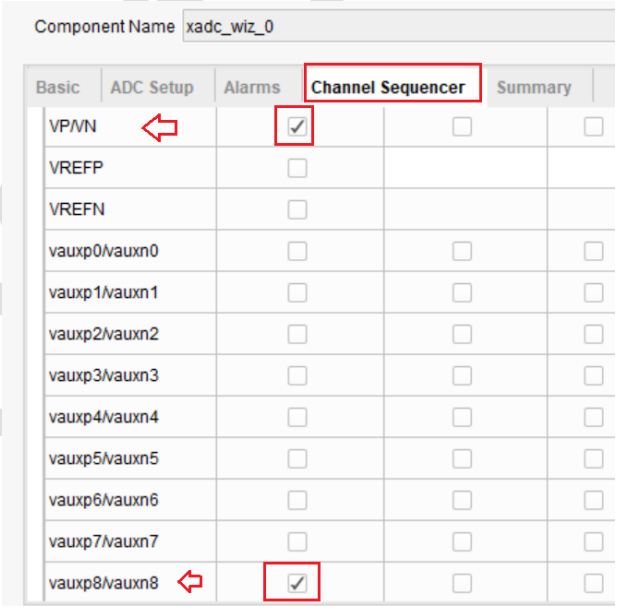


Figure 11. 9: Selecting the Analogue Channels

In the *Channel Sequencer* page, select the channels for the application. In this project, only the external *Vp/Vn* and *Auxiliary channel 8* are selected.

That's it! Now click on OK to finish the XADC setup. Now its time to reveal the VHDL code.

The VHDL driver

```
entity VHDL_XADCdriver is
    Port (
            ADCdata : in unsigned (15 downto 0);
            reset,clk : in std_logic;
            LED : out std_logic;
            D_en,XADCreset,ADCClk_26MHz: out std_logic;
            startConversion,EndOfConv_outLED: out std_logic;
            D_ready,EndOfConv,ADCbusy : in std_logic;
            register_data : out std_logic_vector(15 downto 0);
            write_en : out std_logic;
            data_out : out unsigned (11 downto 0);
            Daddr_out : out std_logic_vector(6 downto 0);
            sevenseg : out std_logic_vector(6 downto 0));
end VHDL_XADCdriver;
```

Code Snippet 11. 1: Entity Declaration

Code snippet 11.1 shows the entity declaration. This shows all the inputs and the outputs of the VHDL driver.

```
type states is (resetXADC,exitReset,strtConv,resetSTconv,check_DRDY,D_EnableHIGH,D_EnableLOW,busyADC,
                check_DRDY2,getdata,checkEoC,delay); --check_busyOUT,dummy_state,master,,delay2
signal current_state,next_state : states;

type master_states is (Ms0,Ms1,Ms2,Ms3);
signal master_current_state,master_next_state : master_states;

signal start_delay, end_delay : std_logic;
signal getADCdata,ADCdata_done : std_logic;
signal ADC_reg_addr : std_logic_vector(6 downto 0);
signal internalADCresult : integer;
```

Code Snippet 11. 2: Declaring the internal signals

Code snippet 11.2 show all the internal signals used. Note the state machine declaration.

```
process (clk,reset)
begin
if reset = '0' then master_current_state <= Ms0;
elsif rising_edge(clk) then master_current_state <= Master_next_state;
end if;
end process;
```

Code Snippet 11. 3:

The process shown in Code Snippet 11.3 is used to control the master state machine. As can be seen, it's clock is the 100 MHz clock.

```
process(master_current_state,ADCdata_done)

begin

case master_current_state is

when Ms0 => ADC_reg_addr <= "0000011"; getADCdata <= '0';
             master_next_state <= Ms1;

when Ms1 => ADC_reg_addr <= "0000011"; getADCdata <= '1';
            if ADCdata_done = '1' then master_next_state <= Ms2;
            else master_next_state <= Ms1; end if;

when Ms2 => ADC_reg_addr <= "0011000"; getADCdata <= '0';
            master_next_state <= Ms3;

when Ms3 => ADC_reg_addr <= "0011000"; getADCdata <= '1';
            if ADCdata_done = '1' then master_next_state <= Ms0;
            else master_next_state <= Ms3; end if;

end case;
end process;
```

Code Snippet 11. 4: Master SM

Code Snippet 11.4 shows the syntax for the master state machine (SM). There are two handshake lines between the master state machine and the slave state machine. The slave state machine is the one that is directly interfaced to the XADC block. The master SM issues a signal (*getADCdata signal*) to trigger the slave SM, then it waits for the slave SM to finish (*ADCdata_done signal*). There is also the channel address denoted as *ADC_reg_addr*, that selects which ADC channel should be sampled.

```
process(clk,reset)
begin
if reset = '0' then current_state <= resetXADC;
elsif clk'event and clk = '1' then current_state <= next_state;
end if;
end process;
```

```
process(current_state,ADCbusy,EndOfConv,D_ready,end_delay,getADCdata)
begin
case current_state is

when resetXADC => D_en <= '0';XADCreset <= '0';startConversion <= '0';write_en <= '0';EndOfConv_outLED <= '0';
                  Daddr_out <= ADC_reg_addr; --03H is the address of status reister for Vp/Vn
                  start_delay <= '0';ADCdata_done <= '0';
                  next_state <= exitReset;


when exitReset => D_en <= '0';XADCreset <= '1';startConversion <= '0';write_en <= '0';EndOfConv_outLED <= '0';
                  Daddr_out <= ADC_reg_addr; --03H is the address of status reister for Vp/Vnp
                  start_delay <= '0';ADCdata_done <= '0';
                  next_state <= check_DRDY;

when delay =>     D_en <= '0';XADCreset <= '0';startConversion <= '0';write_en <= '0';EndOfConv_outLED <= '0';
                  Daddr_out <= ADC_reg_addr; --03H is the address of status reister for Vp/Vnp
                  start_delay <= '1';ADCdata_done <= '0';
                  if end_delay = '1' then next_State <= check_DRDY;
                  else next_state <= delay; end if;


when check_DRDY =>  D_en <= '0';XADCreset <= '0';startConversion <= '0';write_en <= '0';EndOfConv_outLED <= '0';
                    Daddr_out <= ADC_reg_addr; --03H is the address of status reister for Vp/Vn
                    start_delay <= '0';ADCdata_done <= '0';
                    if D_ready = '0' then next_state <= strtConv;
                    else next_state <= check_DRDY; end if;
```

```
when strtConv => D_en <= '0';XADCreset <= '0';startConversion <= '1';write_en <= '0';EndOfConv_outLED <= '0';
                 Daddr_out <= ADC_reg_addr; --03H is the address of status reister for Vp/Vn
                 start_delay <= '0';ADCdata_done <= '0';
                 next_state <= resetSTconv;

when resetSTconv => D_en <= '0';XADCreset <= '0';startConversion <= '1';write_en <= '0';EndOfConv_outLED <= '0';
                    Daddr_out <= ADC_reg_addr; --03H is the address of status reister for Vp/Vn
                    start_delay <= '0';ADCdata_done <= '0';
                    if ADCbusy = '1' then next_state <=  busyADC;
                    else next_State <= resetSTconv ;end if;

when busyADC =>   D_en <= '0';XADCreset <= '0';startConversion <= '0';write_en <= '0';EndOfConv_outLED <= '0';
                 Daddr_out <= ADC_reg_addr; --03H is the address of status reister for Vp/Vn
                 start_delay <= '0';ADCdata_done <= '0';
                 if ADCbusy = '1' then next_state <=  busyADC;
                 else next_State <= checkEoC ;end if; --D_EnableHIGH

-- we do not check the End of Conversion signal but we go straight to retrieving the result----
when checkEoC => D_en <= '0';XADCreset <= '0';startConversion <= '0';write_en <= '0';EndOfConv_outLED <= '1';
                 Daddr_out <= ADC_reg_addr; --03H is the address of status reister for Vp/Vn
                 start_delay <= '0';ADCdata_done <= '0';
                 if EndOfConv = '0'  then next_state <= checkEoC;
                 else next_state <= D_EnableHIGH; end if;

when D_EnableHIGH =>  D_en <= '1';XADCreset <= '0';startConversion <= '0';write_en <= '0';EndOfConv_outLED <= '0';
                      Daddr_out <= ADC_reg_addr; --03H is the address of status reister for Vp/Vn
                      start_delay <= '0';ADCdata_done <= '0';
                      next_state <= D_EnableLOW;


when D_EnableLOW =>  D_en <= '0';XADCreset <= '0';startConversion <= '0';write_en <= '0';EndOfConv_outLED <= '0';
                     Daddr_out <= ADC_reg_addr; --03H is the address of status reister for Vp/Vn
                     start_delay <= '0';ADCdata_done <= '0';
                     next_state <= check_DRDY2;

when check_DRDY2 => D_en <= '0';XADCreset <= '0';startConversion <= '0';write_en <= '0';EndOfConv_outLED <= '0';
                    Daddr_out <= ADC_reg_addr; --03H is the address of status reister for Vp/Vn
                    start_delay <= '0'; ADCdata_done <= '0';
                    if D_ready = '0' then next_state <= check_DRDY2;
                    else next_state <= getdata; end if;

when getdata => D_en <= '0';XADCreset <= '0';startConversion <= '0';write_en <= '0';EndOfConv_outLED <= '0';
                Daddr_out <= ADC_reg_addr; --03H is the address of status reister for Vp/Vn
                start_delay <= '0'; ADCdata_done <= '1';
                if getADCdata = '0' then next_state <= check_DRDY; --check_DRDY
                else next_state <= getdata; end if;

end case;
end process;
```

Code Snippet 11. 5: Event Driven XADC driver

So in the first three states, the SM will first reset the XADC block, then it will go into a delay of 100 mS. At this point the SM will monitor the DRDY signal, and once this goes low, the SM will trigger the conversion process. The SM will then wait for the busy line to go high at which point the start-conversion signal will be reset. The SM waits for the end-of-conversion signal to go high, at which point, the data enable signal will be set by the SM for one clock cycle. Then the SM will monitor the DRDY signal again to go high. After that, the SM will enable the 16-bit ADC data and copies it from the data bus.

```
process(start_delay,clk)
variable count : integer;
begin
if start_delay = '0' then count := 0; end_delay <= '0';
elsif rising_edge(clk) then count := count + 1;
    if count < 10000000 then end_delay <= '0';
    elsif count >= 10000000 then end_delay <= '1';
    end if;
if count >= 10000000 then count := 0; end if;
end if;
end process;

data_out <= ADCdata(15 downto 4) when ADC_reg_addr = "0011000" else "000000000000";
```

Code Snippet 11. 6: Delay of 100 ms

As already stated, in this project two ADC channels were employed and therefore these had to be displayed on different display-media. Code Snippet 11.6 shows the 100 mS delay process and also a statement that assigns the raw 12-bit data to output LEDs. It must be stressed here the versatility of the FPGA as opposed to a microcontroller. In a microcontroller, the raw data had to be shifted to the left by 4 places, however with FPGAs, all one has to do is to extract the bits of interest and assign them to a new register already placed according to their binary weight!

The ADC data is only shown on the LEDs when the address of the Auxiliary channel 8 is available at the output of the XADC block. This is very convenient.

```
internalADCresult <= to_integer(ADCdata(15 downto 4));

sevenseg <= "0111111" when internalADCresult >= 0 and internalADCresult < 400 and ADC_reg_addr = "0000011" else   --0
            "0000110" when internalADCresult >= 400 and internalADCresult < 800 and ADC_reg_addr = "0000011" else  --1
            "1011011" when internalADCresult >= 800 and internalADCresult < 1200 and ADC_reg_addr = "0000011" else  -- 2
            "1001111" when internalADCresult >= 1200 and internalADCresult < 1600 and ADC_reg_addr = "0000011" else  -- 3
            "1100110" when internalADCresult >= 1600 and internalADCresult < 2000 and ADC_reg_addr = "0000011" else  -- 4
            "1101101" when internalADCresult >= 2000 and internalADCresult < 2400 and ADC_reg_addr = "0000011" else  -- 5
            "1011011" when internalADCresult >= 2400 and internalADCresult < 2800 and ADC_reg_addr = "0000011" else  -- 6
            "1001111" when internalADCresult >= 2800 and internalADCresult < 3200 and ADC_reg_addr = "0000011" else  -- 7
            "1100110" when internalADCresult >= 3200 and internalADCresult < 3600 and ADC_reg_addr = "0000011" else  -- 8
            "1100111" when internalADCresult >= 3600 and internalADCresult < 4096 and ADC_reg_addr = "0000011" else  -- 9
            "0000000";
```

Code Snippet 11. 7: Seven Segment Driver

Code Snippet 11.7 shows a convenient way to convert the voltage of a signal from the ADC into decimal levels from 0 to 9. However, to use the relational operators in VHDL, one has to use integer data types. So, a concurrent statement was included to convert the input ADC data from unsigned to integer. Then the new variable or register could be used in the when-else statement.

That should wrap everything up, in the next chapter, the same project will be extended to include the Processing System of the Zynq 7 System-on-Chip.