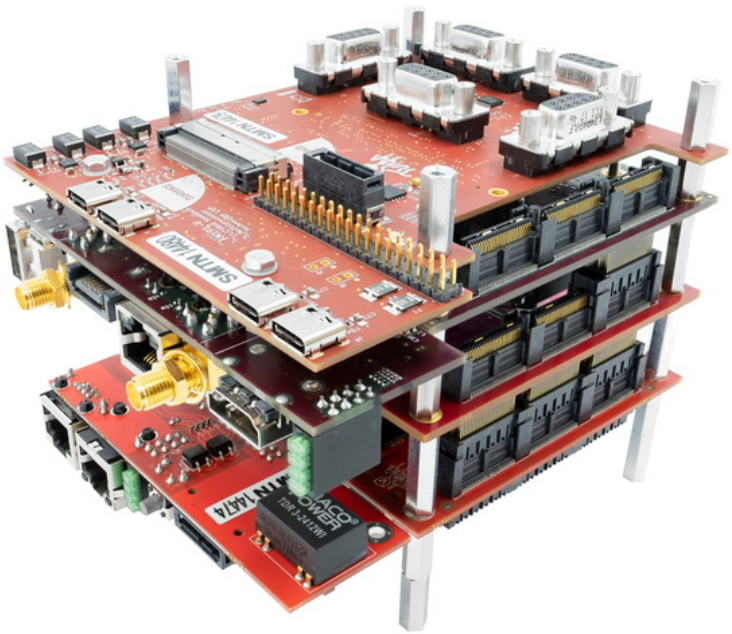
Precision Robotics using the VCS Platform

Step-by-Step tutorial of how to use a Xilinx Zynq FPGA with Sundance's VCS-1 board developed for the EU VineScout Project Grant #737669

Timoteo García Bertoa

August 13, 2019



To María del Carmen Meizoso López, who taught me the fundamentals of VHDL, FPGA and CPLD architectures.

Special thanks to:
Emilie Wheatley, Pedro Machado and Flemming Christensen for your support.

Thank you Mateo Vázquez for your reviewing, and I hope you found this book useful in your journey with FPGAs.

Contents

List of Figures

CHAPTER 1

FPGAs. What is that?

I've been asked many times out there: *What is your job in electronics?*, *FPGA, what is that?*. To be honest, I wondered the same thing in university when I heard about Field Programmable Gate Arrays for the first time.

Most people have notions of what a processor is, although not everybody would be able to define differences between a processor and a microcontroller, but many get lost when the question is: *What is an FPGA?*

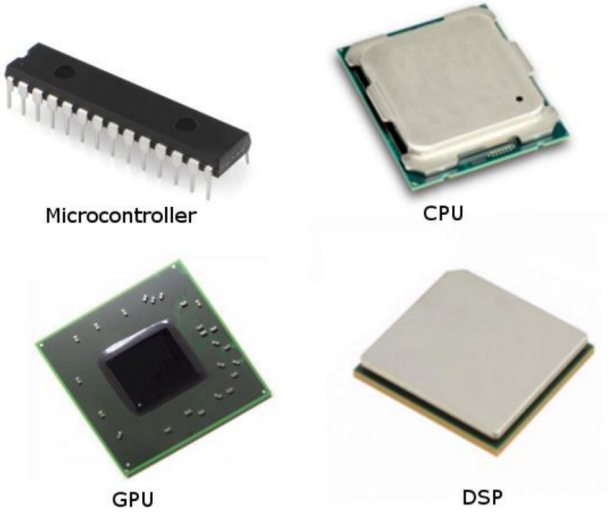


Figure 1.1: Processing devices

We are all familiar with the concept of CPU or Central Processing Unit, an integrated circuit which is capable of interpret instructions, and execute tasks using internal hard-fixed elements as registers, arithmetic logic unit, etc. The concept of processor was extrapolated to other similar architectures that target different applications, for different needs or end-users. For example, a DSP, or Digital Signal Processor, is a processor which has been optimised to execute tasks related to digital signal processing. Likewise, a GPU or

Graphics Processing Unit, will be optimised for image processing. Even, a processor's architecture can be used for a certain task, in a reduced form factor, for a very low price, integrating in the same chip the necessary I/Os, making things much easier, in a way that any child would be able to interact with it, or what it's commonly known as a microcontroller.

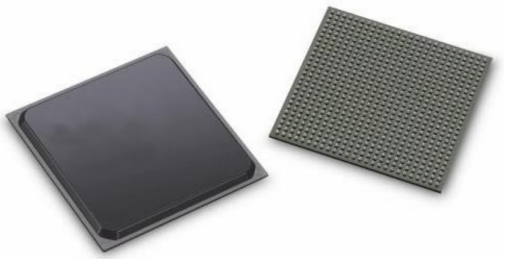


Figure 1.2: FPGA

So, where does an FPGA fit among these other integrated circuits? Just observing one, it doesn't look so much different.

In a very informal way, an FPGA is a bunch of logic gates, in which you define what the inputs and outputs are for each one of them.

You don't remember what logic gates are? And flip-flops? If you haven't gone across these concepts, it would be good if you pick up a book that explains the basics of digital design.

Nevertheless, to understand the architecture of an FPGA, is important to know that everything can be reduced to the minimum expression of digital design, which is a logic gate.

There are different brands of FPGAs out there, and everybody uses their own

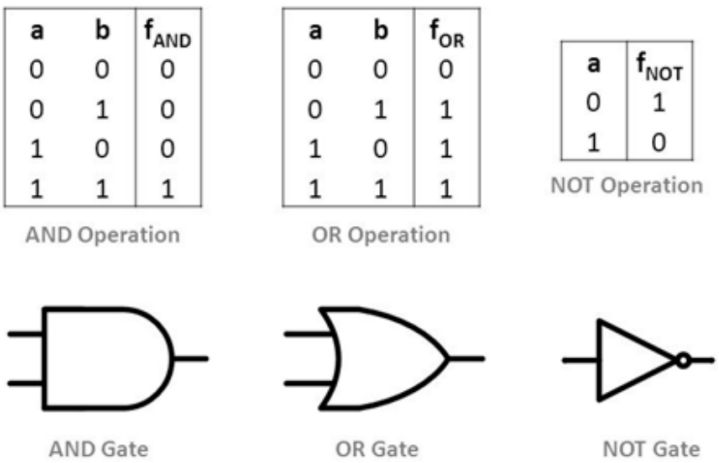


Figure 1.3: Basic logic gates

terminology to refer to the different elements of their architecture. In the case of Xilinx FPGAs, a CLB or Configurable Logic Block, is the most basic structure, which is repeated along the die thousands of times.

A CLB is formed by different elements: Look-Up Tables (LUTs), which are in essence what it's considered in digital design as the truth table, and are able to implement the most basic logic gates; Flip-Flops, which are the smallest element capable of holding information and work synchronously with a clock input; Carry-Chains, which help with arithmetic operations; Multiplexers, which allow to output data by selection of the input; Distributed RAM, which stores data, and Shift Registers, which allow to shift data to ease the use of Flip-Flops for other tasks.

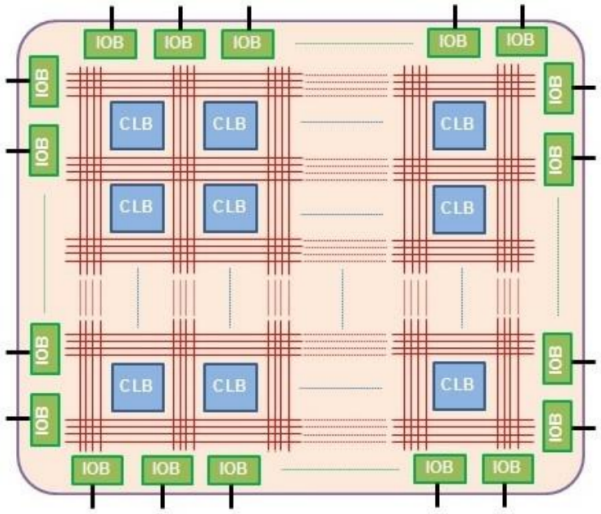


Figure 1.4: CLB structure

Maybe these are too many concepts to swallow in at once, but it happened to me that after learning all this, I still had a question in my mind. Right, FPGAs, based on digital logic. Now, what's the point? Are they better? And if so, why?

When I explain very briefly that I work with FPGAs, and as I mentioned before, that they are a bunch of logic gates, in which you define what the inputs and outputs are for each one of them, people generally nod, looking at me as in: *Yeah, I know what you mean, but... what's the difference then? Why FPGAs?*, so I always give them the same example:

Imagine that your system consists of a room, where there is a window, a switch and a lamp. Your program has to be: close the window so that there is no light from outside, go towards the switch, and turn it on so that the lamp illuminates the room.

A processor/microcontroller, programmed in high level programming languages

like C/C++, would be able to do that in a series of sequential instructions:

- Go towards the window, which might take 4 seconds
- Close the window, which might take 1 second
- Go towards the switch, which might take 4 seconds
- Turn on the switch, which might take 1 second

Figure 1.5: Our example

As a result, the room is illuminated. $(4+1+4+1 = 10$ seconds).

An FPGA, programmed using RTL design (Register Transfer Level), would be able to do that with concurrent programming, with languages like VHDL/Verilog. The program is not sequential, instead, is an algorithm treated simultaneously for every input, to provide the required output. In fact, this program has instructions that are not exclusive. You can turn on the switch, and then close the window, and the effect will be the same. But from an FPGA point of view, the system would have 2 inputs and 1 output.

As long as both the window and the switch are in their required conditions (closed and turned on), the room will be illuminated with the lamp.

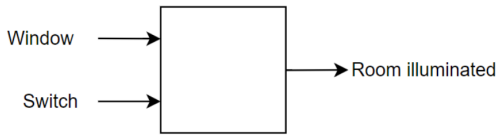


Figure 1.6: I/Os of our system

Before, it was assumed that going towards the window and close it takes 5 seconds, same with the switch, and the whole sequential process takes 10 seconds. An FPGA would operate these 2 tasks at the same time, synchronously, and provide the output (room illuminated) in 5 seconds.

But what if your code in C/C++ is optimised to its maximum extent, and you still want to reduce those 10 seconds with a CPU? Well, overclocking is a practise that allows the processor to run at a higher clock rate. In FPGAs, it's quite simple to increase the frequency of a system, which can be entirely synchronous.

Essentially, this is one of the reasons why FPGAs are very suitable for environments where pre-processing is required. Another very important reason, which won't be covered in these chapters, is power consumption. FPGAs are characterised by its low consumption, and are a clear advantage for certain applications because of that.

CHAPTER

2

Introduction to Zynq

FPGAs evolved with the years, providing more and more logic cells, with more flip-flops, more block rams, more dsp slices, more I/Os, more of everything, for the same size of the integrated circuit. But there always was one question in the air: is this technology only available for some elite of engineers that learn how to program VHDL/Verilog? Well, the answer is yes, for those devices that are faithful to that architecture, but also no: the more the software tools evolve, and companies work hard to develop Intellectual Properties (IPs), the more your workload can be reduced, and nowadays, many complex tasks can be done in the FPGA, that years ago would have been a terrible nightmare (or dream for some!) But still using IPs, basics of VHDL/Verilog are required.

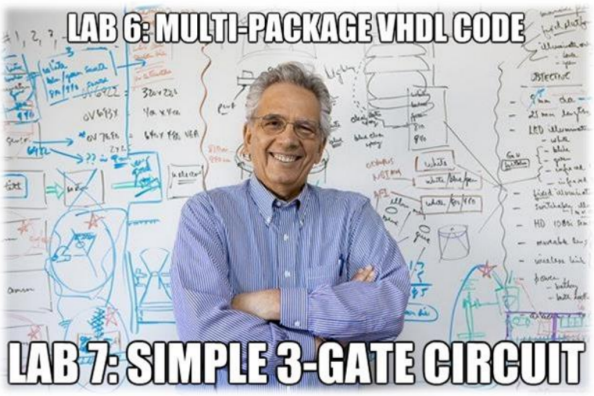


Figure 2.1: VHDL can be adorably painful

Or are they not?

What if I told you that you can program an FPGA, using some bits of VHDL (Vivado), but your most complex algorithms developed in C/C++, and converted for you into VHDL/Verilog? Well, do you still cringe when listening VHDL/Verilog? So, what about your full design in C/C++, implemented

in the FPGA? (Vivado HLS).

Figure 2.2: Xilinx tools

It still feels like you have to know how the architecture of the FPGA is for you to optimise the C/C++ code effectively implemented with Vivado HLS. So..

What if I told you, that there is an architecture (Zynq), which provides hard processor cores and an FPGA, altogether within the integrated circuit? Basically, you can develop your own program in C/C++, and decide which functions you want to accelerate in the FPGA, blindly trusting that the software tool understands the FPGA architecture for you (SDx).

Zynq 7000 by Xilinx, was released in 2011, and the architecture is formed by a PS (Processing System), and PL (Programmable Logic).

The PS provides a Dual ARM Cortex A9, with some built-in controllers with classic interfaces (SPI, I2C, UART, CAN, etc.). The PL follows a 7 series FPGA architecture, including PCIe Gen2 control. The PS and the PL can communicate with each other, with a protocol called AXI.

This architecture has a lot of experience and years on its back, which provides a lot of

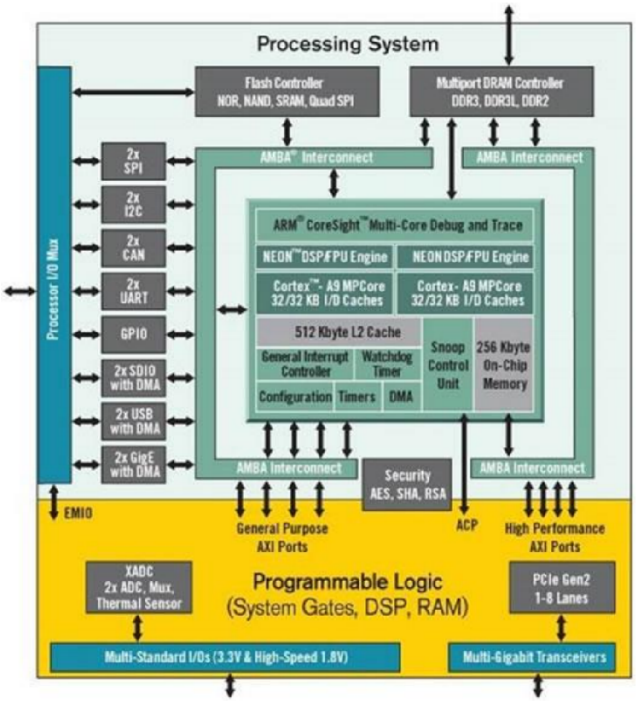


Figure 2.3: Zynq 7 Series

resources and support.

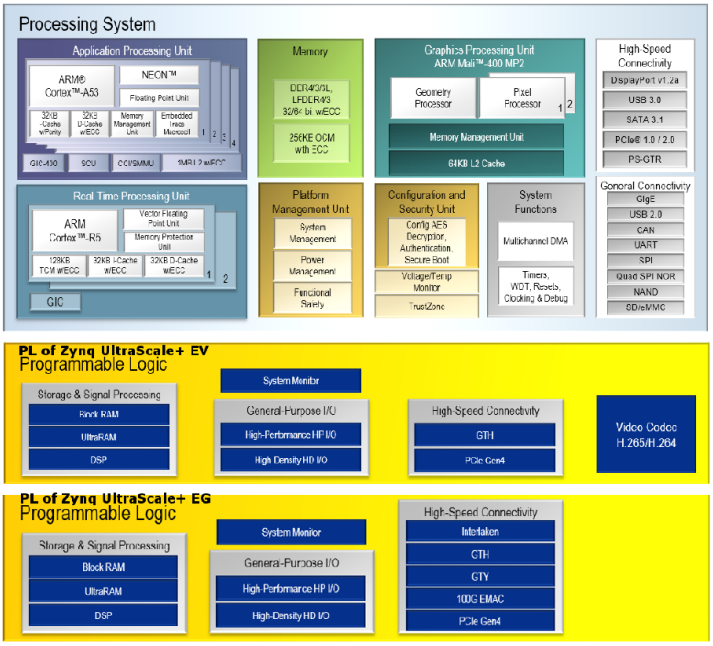


Figure 2.4: Zynq Ultrascale+

Zynq 7000 evolved into Zynq Ultrascale+ in 2016, with a multiprocessor system-on-chip which provides, again, a PS and a PL.

The PS has a Quad ARM Cortex A53, GPU, Real-Time Dual ARM R5, and built-in controllers, adding USB3, PCIe Gen2, SATA 3.0.

The PL, moved from PCIe Gen2 to Gen4, and added, depending on the package, additional transceivers, or a built-in video codec H.265/H.264.

CHAPTER 3

Playing with a LED

Sundance Multiprocessor Technology LTD, has been manufacturing products based on FPGAs for 30 years in the United Kingdom. The hardware kit that is being used for this chapter, designed and produced by Sundance, is VCS-1.

The VCS-1 (Vision, Control and Sensors) is a hardware kit intended to perform vision applications in real time using ZYNQ (ARM + FPGA) architectures using embedded LINUX.

Information about VCS-1 can be found here:
https://www.sundance.com/vcs-1/
https://www.slideshare.net/SundanceDotCom/sundance-vcs1-for-precision-robotics

- EMC2-DP, a carrier board, compatible with different SoMs.
  https://www.sundance.technology/som-cariers/pc104-boards/emc2-dp/

  Currently, software support is given for the following SoMs:

  - TE0715-30 (Zynq 7000 Z7030)
    https://www.sundance.technology/som-cariers/pc104-boards/emc2-z7030/
  - TE0820-3EG(Zynq Ultrascale+ ZU3EG)
    https://www.sundance.technology/som-cariers/pc104-boards/emc2-zu3eg/
  - TE0820-4EV (Zynq Ultrascale+ ZU4EV)
    https://www.sundance.technology/som-cariers/pc104-boards/emc2-zu4ev/

  - TE0820-4CG (Zynq Ultrascale+ ZU4CG)
    https://www.sundance.technology/som-cariers/pc104-boards/emc2-zu4cg/

- FM191-RU V1, FMC module
  https://www.sundance.technology/system-on-modules-som/fmc-modules/adc-dac-fmc-modules/fm191/

For more information of how to set up the hardware and use the prebuilt files, contact Sundance, a starter's guide is provided to be followed step by step.

This chapter intends to show how to use the Zynq architecture, explaining very basic concepts to be able to make a simple LED blink. As I always say: *If you can turn on a LED, you can do anything.*

Now we roughly know what an FPGA is, we know what the advantages are, and we are ready to get to know the Zynq architecture. In order to make a LED blink, there are many different paths. And I will try to consider all of them, from the simplest one, to the most complex. All of them are important, and all of them have their own advantages, strengths and weaknesses. But knowing every single way of doing things around the tools, grant you the power of being able to design, not only what you want to design, but efficiently.

So, I will list off all the methods I will cover, and go one by one (the tools I will be using are Vivado 18.3, SDK 18.3, Petalinux 18.3). Also, within each method, I will list the concepts briefly covered:

- Method 1: Using VHDL, programming the PL through JTAG
  - Vivado project creation
  - Adding sources
  - Brief introduction to VHDL's structure
  - Adding constraints (locations, I/O Standards)
  - Building process
  - Project's files hierarchy
  - JTAG programming
- Method 2: Using VHDL, using IP Integrator, programming the PL through JTAG

  - Hierarchical design
  - IP Integrator
  - VHDL modules in IPI
- Method 3: Creating and packaging an IP
  - Custom IP creation
- Method 4: Using the PS to provide clocking, standalone, using JTAG
  - Using the PS in IPI
  - SDK
  - FSBL
- Method 5: Using C, programming the PS/PL, standalone, using JTAG
  - AXI Protocol
  - Standalone drivers
  - Debugging from SDK
- Method 6: Booting from SD card
  - Booting methods
  - SD card files generation
- Method 7: Using custom board files from Sundance, using a board interface
  - Adding board files
  - How board files work, and their structure
  - Using board interfaces in IPI
- Method 8: Automating all the previous methods using scripts
  - Generating .bit and .hdf using tcl
  - Generating SD boot files using SDK in batch mode
- Method 9: Creating a kernel out of the design, and using Embedded Linux
  - The need of an OS

  - Introduction to Linux
  - Standalone vs Linux structure of a project
  - Kernel generation using Petalinux
  - Setting up OS-based SD card
  - Accessing signal from Embedded Linux
- Method 10: Using C and Python running Ubuntu
  - Setting up Ubuntu file system using cross-compiling
  - Accessing signal using C from Ubuntu for ARM
  - Accessing signal using Python

3.1 Method 1: Using VHDL, programming the PL through JTAG

Before I start with any method, I recommend avoiding spaces, in either path names or file names.

If we open Vivado, and click on *Create Project*, a window appears. Click on *Next*, and define a project name and location. Leave *Create project subdirectory* ticked, and click *Next*.

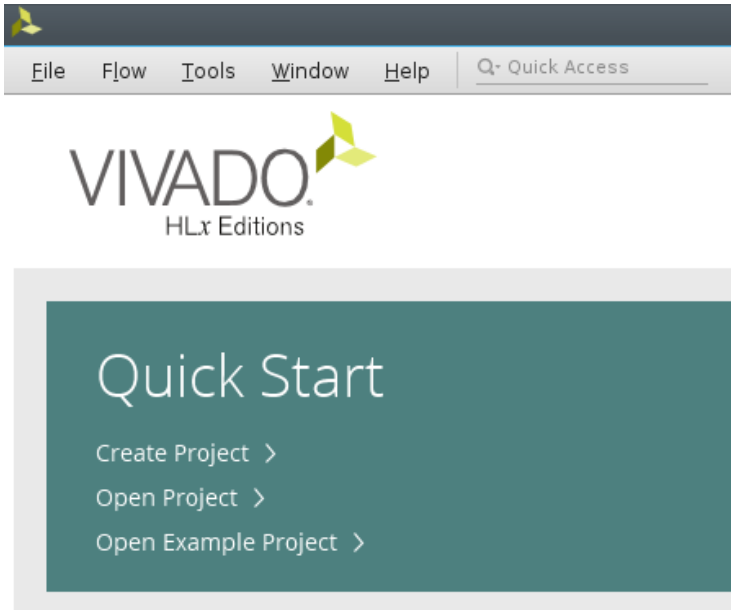


Figure 3.1: Create Project

Select *RTL Project*, and leave *Do not specify sources at this time* ticked.

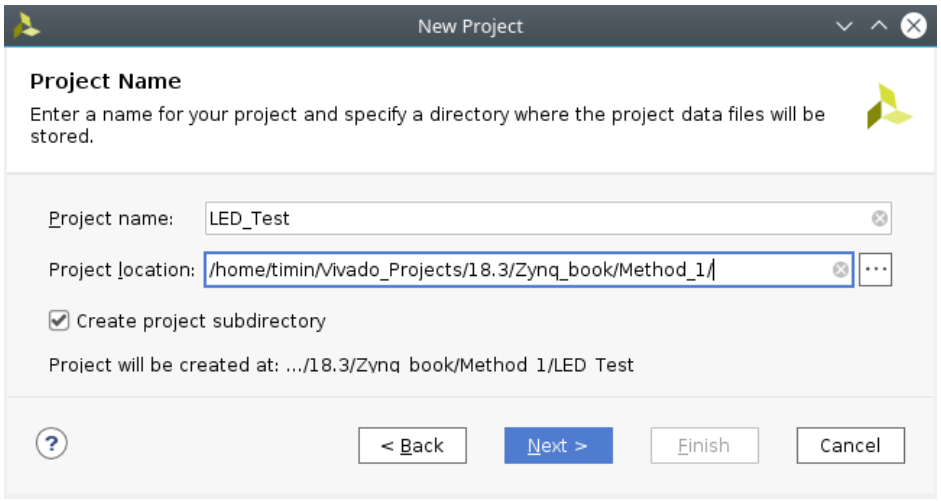


Figure 3.2: Project name

Click on *Next*, and search in *Parts* the corresponding device:

- *xc7z030sbg485-1* (TE0715-30 -Zynq 7000 Z7030-)
- *xczu3eg-sfvc784-1-e* (TE0820-3EG -Zynq Ultrascale+ ZU3EG-)
- *xczu4ev-sfvc784-1-e* (TE0820-4EV -Zynq Ultrascale+ ZU4EV-)
- *xczu4cg-sfvc784-1-e* (TE0820-4CG -Zynq Ultrascale+ ZU4CG-)

Click *Next* and *Finish*.

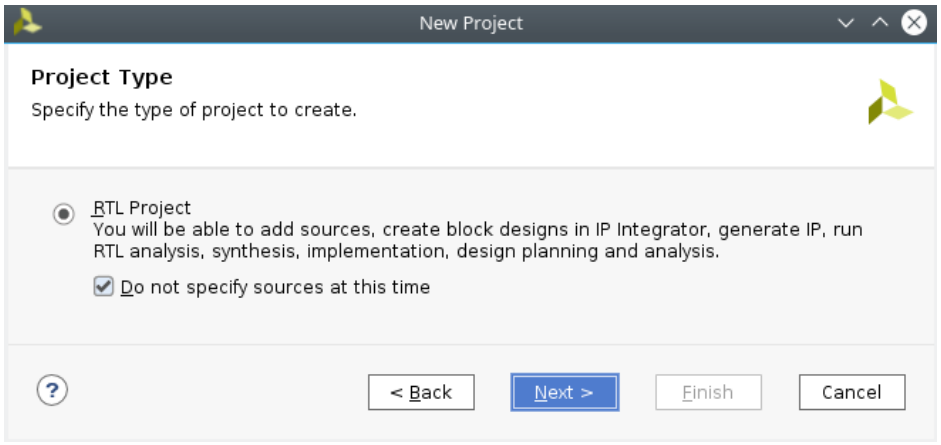


Figure 3.3: Project type

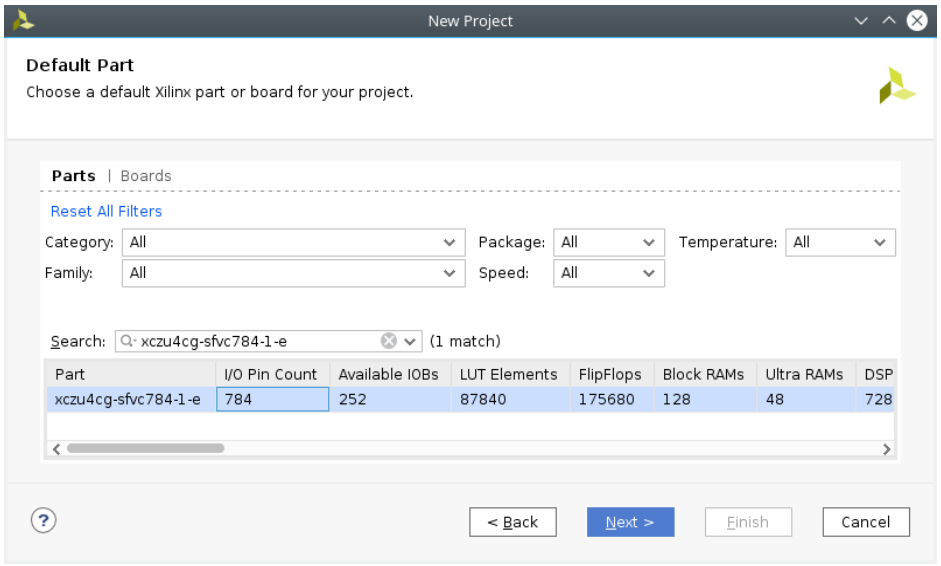


Figure 3.4: Part selection

A project is now created. Right click on *Design Sources* in the window *Sources*, and click on *Add Sources*. Select *Add or create design sources*, and click *Next*. Click on *Create file*. Select *VHDL* File type, and give it a name. Click on *OK*, and then *Finish*.

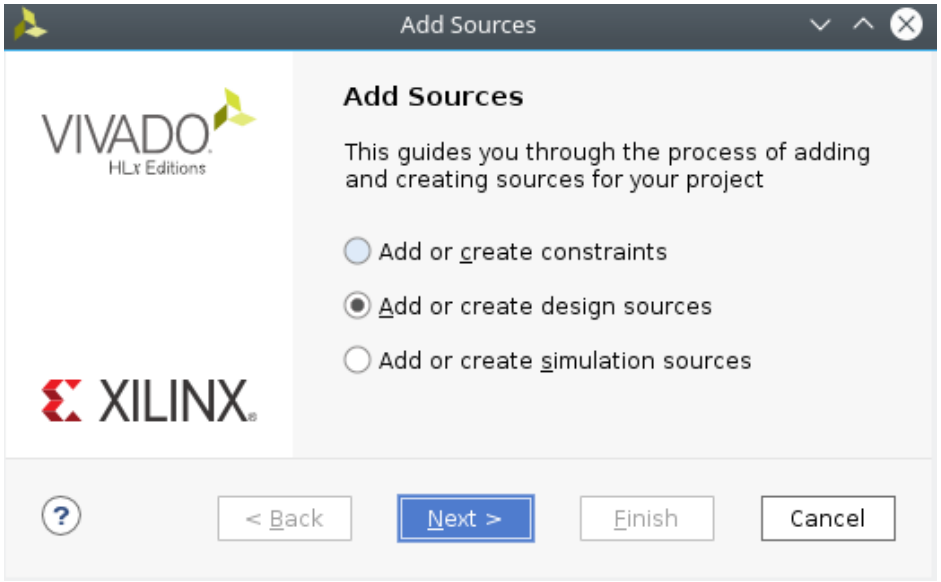


Figure 3.5: Add sources

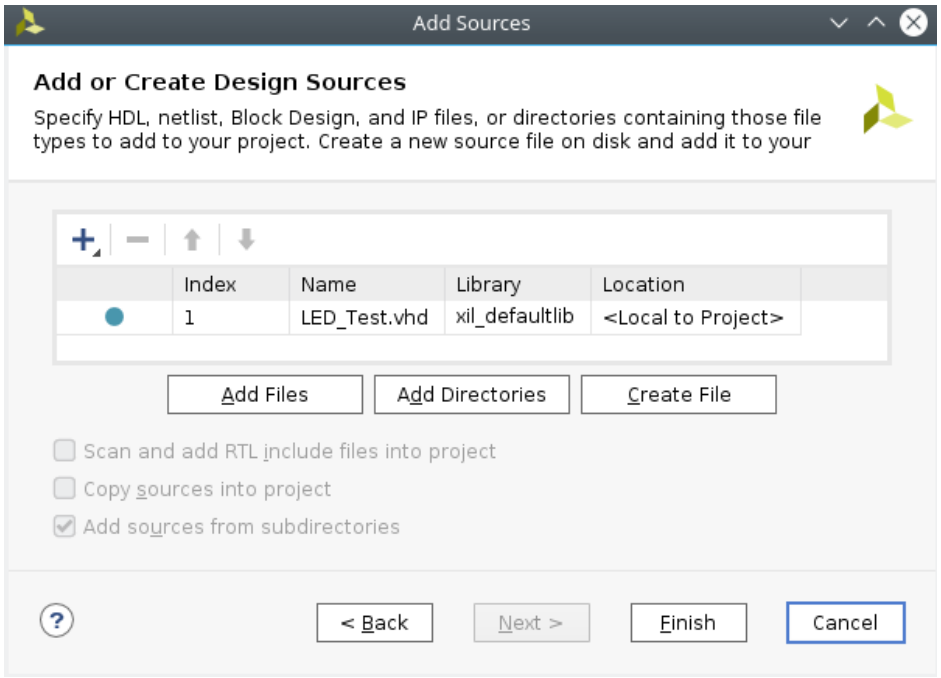


Figure 3.6: Creating .vhd file

A window will pop up, for you to define I/O Ports by default. Just leave it as it is, and click *OK*, and *Yes* when it asks if you are sure.

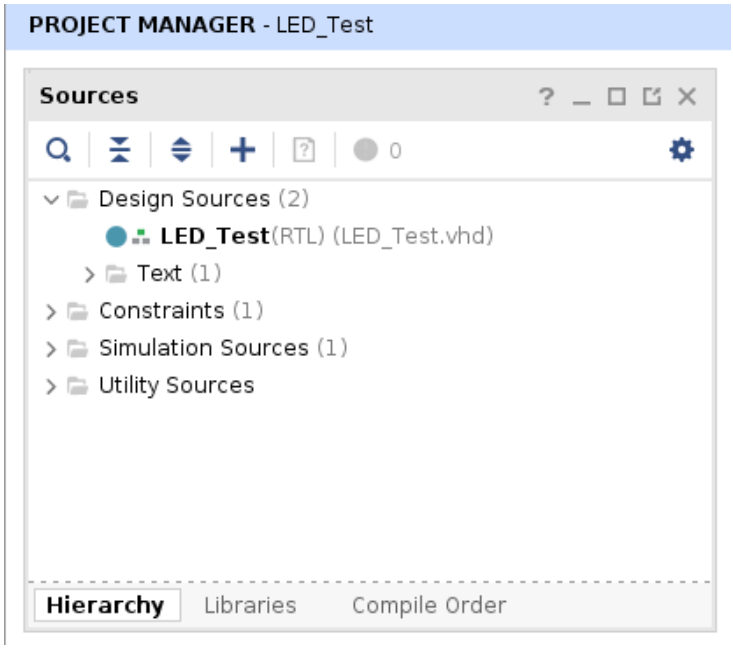


Figure 3.7: Source file created

Double click on your new design file, and it's time to code!

After some years of coding in VHDL, I always define things in the same way, and everybody has their own general rules. I believe I have to improve a lot, but I will try to guide you through in a clear way.

First of all, it's not relevant for the outcome of the implementation, but it's good to describe what this file is about.

```
----------------------------------------------------------------------------------
-- Company: Sundance Multiprocessor Technology LTD
-- Engineer: Timoteo Garcia Bertoa
--
-- Design Name: LED_Test
-- Module Name: LED_Test
-- Project Name: Blinking LED: Method 1
-- Target Devices: xczu4ev-sfvc784-1-e
-- Tool Versions: Vivado 2017.4
--
-- Additional Comments:
--
----------------------------------------------------------------------------------
```

This clarifies a lot what this file is, and what's expected from it. Now, every VHDL file requires libraries that contain the basic standard variable

definitions, operators, etc. That's all defined here:

```
----------------------------------------------------------------------------------
--LIBRARIES
----------------------------------------------------------------------------------
library IEEE;
    USE IEEE.STD_LOGIC_1164.ALL;
    USE IEEE.NUMERIC_STD.ALL;
library UNISIM;
    USE UNISIM.VCOMPONENTS.ALL;
```

Every VHDL module has an entity, which defines what the inputs and outputs are in our system. In this case, we just want to blink a LED, so, we need obviously an output for it, and also, a clock that runs our system.

The problem here, is that generally, evaluation boards come with crystal oscillators on-board, with a straight connection to the FPGA, for anybody to use that clock as an input, or to derive other clocks with internal PLLs in the FPGA.

For this combination of boards, EMC2-DP and TE0820-4EV, there is no direct clock accessible in the PL. When I say there is not direct clock, I mean that the EMC2-DP actually provides 4 clocks to the FPGA through a clock synthesiser, but this one requires a previous configuration. As we will see in further methods, we can just use a clock from the PS, which would solve this issue straight away, but as I wanted to make this work with just the PL, what we could do is using an external clock, from a signal generator, and use the SMA connector J5 on the board, to inject this clock in the FPGA.

Beware that we are working under 1.8V, so this external clock shouldn't have more amplitude than that.

The entity reflects an input (clk), and an output (led). A generic variable called FREQ is there to define what frequency will be used in the signal generator. For now, we can manually initialise that variable to 25MHz, and assume that will be our clock.

```
----------------------------------------------------------------------------------
--ENTITY
----------------------------------------------------------------------------------
entity LED_Test is
    generic(
        FREQ : integer:=25000000
    );
    port(
        clk : in std_logic;
```


```
        led : out std_logic
    );
end LED_Test;
```

Having defined an entity of our system, with an input and an output, we just need to say what it's happening in between. That's what is called the architecture. This architecture is divided in two processes: *Divider* and *Blinking*. Each process is synchronous to the clock input.

- Divider, basically counts up rising edges of the clock input, and asserts a flag called *toggle* when it counts up to the frequency value in MHz. This means that *toggle* will be asserted every second.
- Blinking, toggles an internal variable called *iled*, every time *toggle* is 1. Assigning *iled* to the output *led*, I think it's pretty obvious what it will happen: the led will toggle every second.

```
----------------------------------------------------------------------------------
--ARCHITECTURE
----------------------------------------------------------------------------------
architecture RTL of LED_Test is

    signal iled     : std_logic:='0';
    signal toggle   : std_logic :='0';
    signal counter  : integer:=0;

begin

Divider: process(clk)
begin
    if rising_edge(clk) then
        if(counter=FREQ-1) then
            toggle <= '1';
            counter <= 0;
        else
            toggle <= '0';
            counter <= counter + 1;
        end if;
    end if;
end process Divider;

Blinking: process(clk)
begin
    if rising_edge(clk) then
        if(toggle='1') then
            iled <= not(iled);
        end if;
    end if;
```


```
end process Blinking;

led <= iled;

end RTL;
```

There are 2 things to do before we can implement this project into the FPGA:

- Define the constraints

  The constraints are needed to "tell" the tool, in this case Vivado, what are your requirements previous to build the project. If you think of it, the only things Vivado knows until now are:

  - The user created a project, and is using *xczu4ev-sfvc784-1-e* as the device target. This means that Vivado already knows how many I/Os are available in the device, how the architecture of the device is, etc.
  - The user made a .vhd file, defining an input port, and an output port, and some code in between both. That file is the top level file.

  But Vivado doesn't know where your input goes, or where your output goes. Vivado doesn't know either if your I/Os are using 3.3V, or 2.5V, or 1.8V.

  Again, that's why constraints are needed, to "tell" Vivado those things.

- Build the project

  When Vivado knows everything it needs to build your project, and give you a file (in this case .bit file) for you to implement it in the FPGA, there are 3 steps that the tool will follow in order to do so.

  - Synthesis. This process essentially means to convert your VHDL code into an RTL schematic, where all the connections are resources available in the device you decided to target.
  - Implementation. This process intends to apply the synthesised design into the architecture of the device targeted, trying to accomplish what the constraints say.
  - Bitstream generation. All the information merged after implementation, is loaded into a single .bit file.

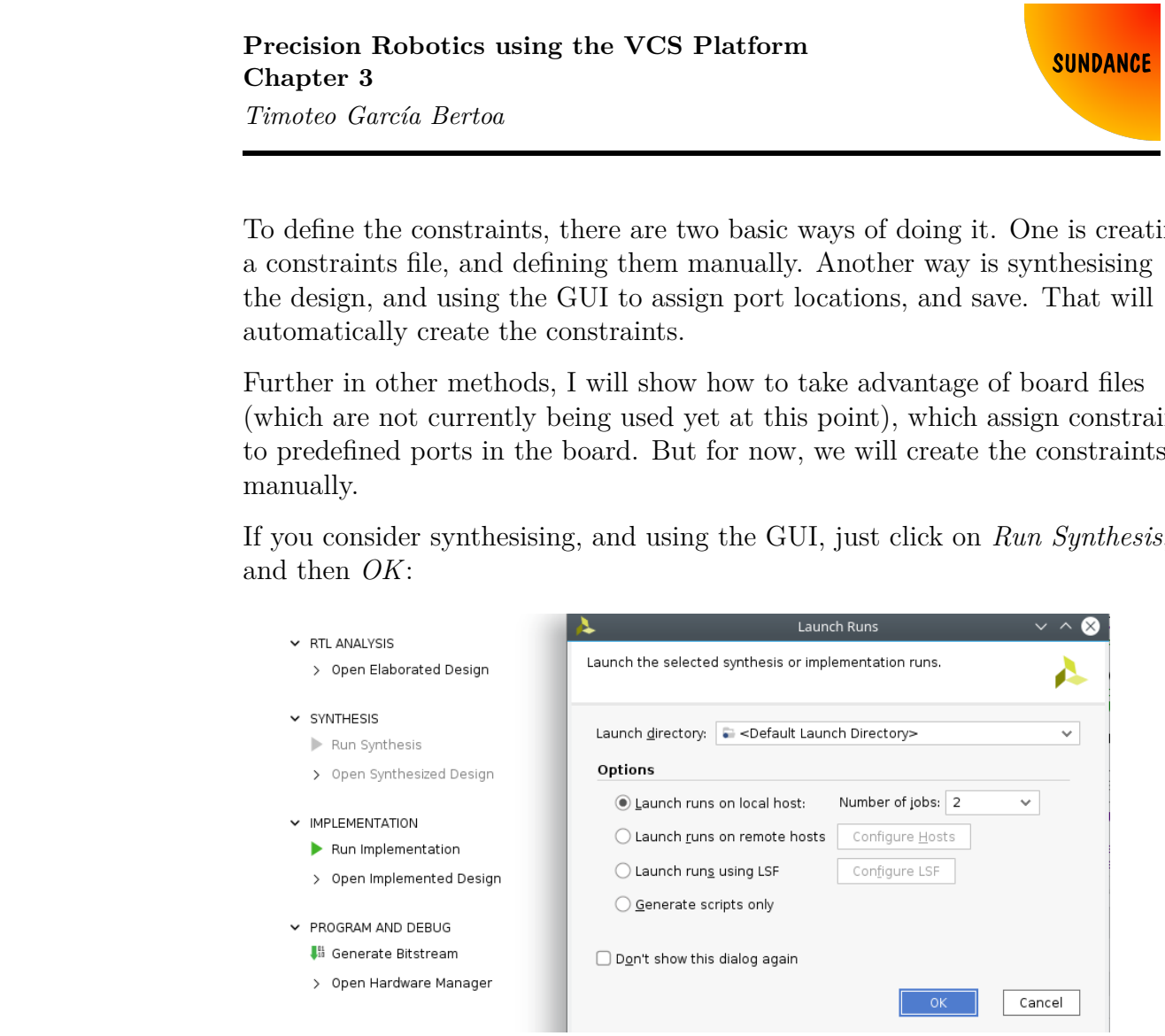


Figure 3.8: Synthesis

To define the constraints, there are two basic ways of doing it. One is creating a constraints file, and defining them manually. Another way is synthesising the design, and using the GUI to assign port locations, and save. That will automatically create the constraints.

Further in other methods, I will show how to take advantage of board files (which are not currently being used yet at this point), which assign constraints to predefined ports in the board. But for now, we will create the constraints manually.

If you consider synthesising, and using the GUI, just click on *Run Synthesis*, and then *OK*:

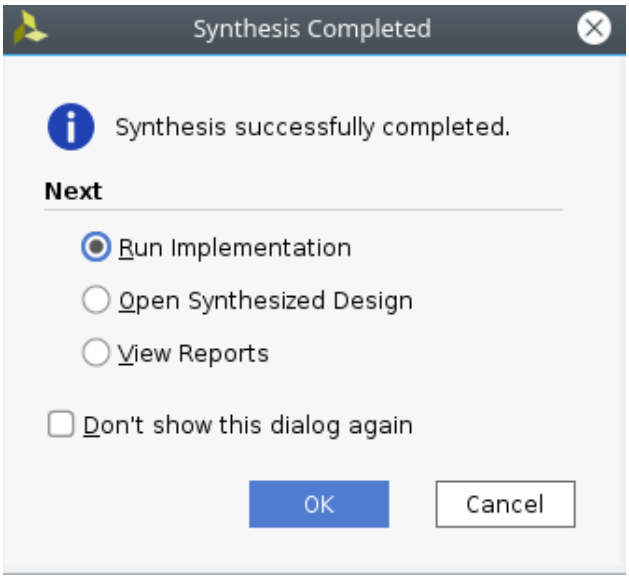


Figure 3.9: Synthesis completed

After synthesis, open the Synthesised design, and then go to the *I/O Ports* tab at the bottom (the layout has to be *I/O Planning*, it can be changed in *Layout*), where you can define *clk* as an input, with location in pin H7 (J5 SMA connector of the EMC2-DP) and *led* as an output located in pin B5 (LED on-bard, both at the edge and at the SEIC). Both signals should be defined as LVCMOS18.

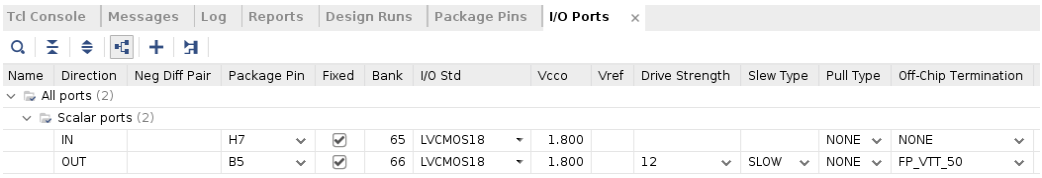


Figure 3.10: Constraints GUI

Clicking on the Save icon, it will ask you to create a constraints file, so just give it a name.

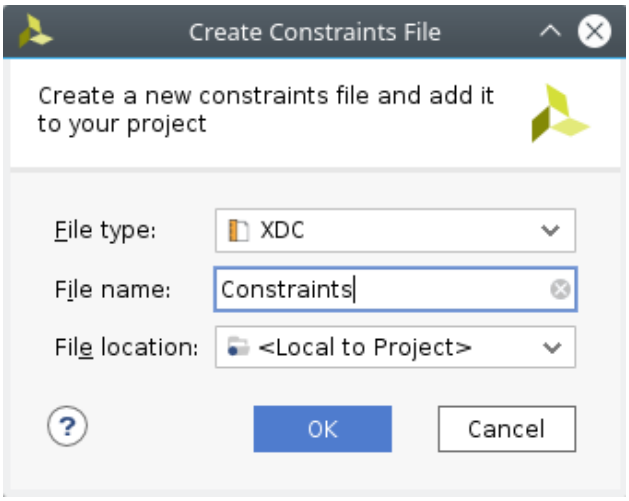


Figure 3.11: Constraints file creation

As I've shown before with the code in VHDL, with the constraints file is the same, comments help to order the information, so I always want to write my own constraints file. This is the code:

```
##############################################################################################

# 25MHz
create_clock -period 40 -name clk -waveform {0.000 20} [get_ports clk]

##############################################################################################

# False route for J5 SMA
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clk_IBUF_inst/O]

##############################################################################################

# LED
# PIN LOCATION
set_property PACKAGE_PIN B5 [get_ports led]
# I/O STANDARD
set_property IOSTANDARD LVCMOS18 [get_ports led]

# CLOCK INPUT
# PIN LOCATION
set_property PACKAGE_PIN H7 [get_ports clk]
# I/O STANDARD
set_property IOSTANDARD LVCMOS18 [get_ports clk]

##############################################################################################
```

On top, I declared a create_clock constraint, which tells Vivado that on the input pin (*clk*) there will be a clock of around 25MHz. I also created a false route for this pin, because is not a clock capable pin, and Vivado will complain otherwise.

Below, I declared the pin locations and I/O Standards of both the input and the output.

From this moment, just build the project, generating the bitstream. The way the projects classify their files is very simple:

- Main folder
  - *<ProjectName>.srcs*, where you will find:
    - Source files
    - Constraints files
  - *<ProjectName>.runs*, where you will find:
    - Generated files, like your bitstream
  - *<ProjectName>.sdk*, where you will find:
    - .hdf file generated when exporting the bitstream (Method 4)
    - SDK local workspace (Method 4)

In order to program the FPGA with the bitstream, it can be done through JTAG, using a Xilinx USB II Platform cable. Click on *Open Hardware Manager*, and then *Open target* → *Open New Target...*

A window pops up, just click on *Next*. At this point, in *Hardware Server Settings*, you can choose a local server or remote server. In this case, *Local Server* is what you want. At this point, your hardware should be recognised by the tools, and you should be able to just click on *Finish*.

Figure 3.12: Xilinx USB II Platform cable

A chain should be visible for you, where you will see the arm cores, and the FPGA, with a built-in XADC core. If you right click on the FPGA, you can choose *Program Device...* and then browse your .bit file. After programming the FPGA, the LED should blink as expected.

3.2 Method 2: Using VHDL, using IP Integrator, programming the PL through JTAG

We covered the simplest way of making a LED blink using VCS-1 and Vivado 17.4. But when I say simplest, I mean simplest conceptually, not easiest. Vivado has grown during the years, easing the design process from tools like ISE Design. As a second method, I just want to explain how to use IP Integrator, without leaving VHDL completely.

If you know about hierarchical design with VHDL, you know that VHDL designs tend to wrap modules on top of other modules, creating a hierarchy where at the end, only one level exists at the top, where the I/Os of the system are defined, and constraint.

In our case, we just have 1 input and 1 output, and creating 2 levels of design would be a bit weird. When we get a defined module (when I say *defined*, I mean that there is an entity and an architecture for it) and *plug* it in another upper level, we call that *instantiation*. So, for example, to instantiate our module (I called it *LED_Test.vhd*) into another module, let's call it *Top_Level.vhd*, it would be something like this:

```
----------------------------------------------------------------------------------
-- Company: Sundance Multiprocessor Technology LTD
-- Engineer: Timoteo Garcia Bertoa
--
-- Design Name: LED_Test
-- Module Name: Top_Level
-- Project Name: Blinking LED: Method 2
-- Target Devices: xczu4ev-sfvc784-1-e
-- Tool Versions: Vivado 2017.4
--
-- Additional Comments:
--
----------------------------------------------------------------------------------


----------------------------------------------------------------------------------
--LIBRARIES
----------------------------------------------------------------------------------
library IEEE;
    USE IEEE.STD_LOGIC_1164.ALL;
    USE IEEE.NUMERIC_STD.ALL;
library UNISIM;
    USE UNISIM.VCOMPONENTS.ALL;
----------------------------------------------------------------------------------
--ENTITY
```


```
----------------------------------------------------------------------------------
entity Top_Level is
    port(
        clk : in std_logic;
        led : out std_logic
    );
end Top_Level;

----------------------------------------------------------------------------------
--ARCHITECTURE
----------------------------------------------------------------------------------
architecture RTL of Top_Level is

component LED_Test
    generic(
        FREQ : integer:=25000000
    );
    port(
        clk : in std_logic;
        led : out std_logic
    );
end component;

begin

inst_LED_Test: LED_Test
    generic map(
        FREQ => 25000000
    )
    port map(
        clk => clk,
        led => led
    );

end RTL;
```

As you can see, there is a declaration of a component in the architecture, and an assignment of the inputs/outputs of our block *LED_Test* to the inputs/outputs of *Top_Level*.

The files look like this in the Sources window:

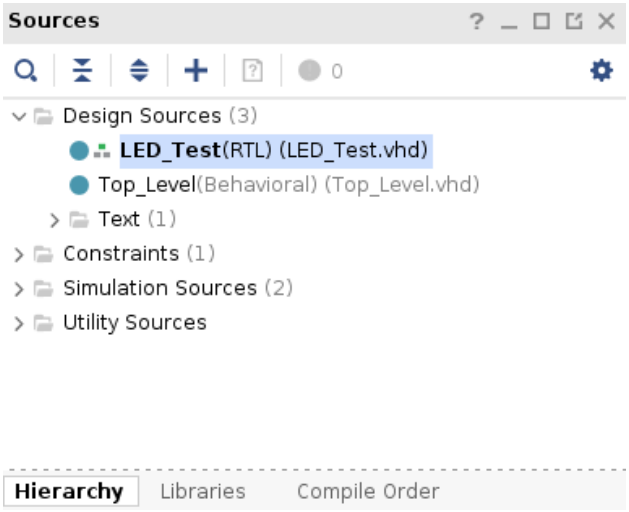


Figure 3.13: Top level source

If we right click on *Top_Level*, and then *Set as Top*:

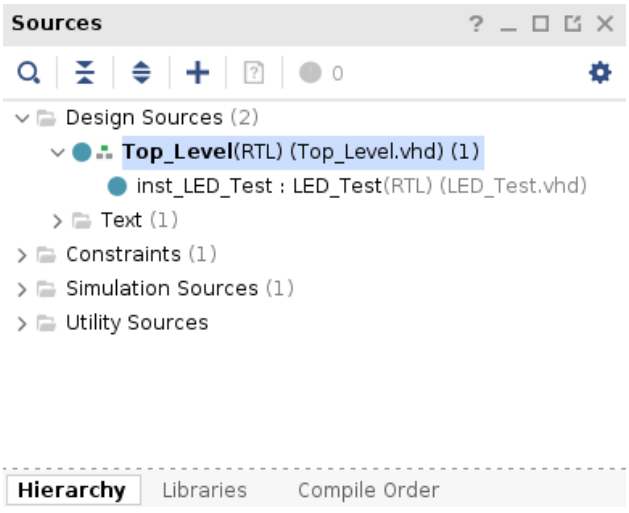


Figure 3.14: Top Level hiearchy

That's what is called hierarchical design in VHDL. I just wanted to mention this, before showing IP Integrator, because this concept is applied.

Top_Level.vhd won't be necessary, it can be deleted.

Go to *Settings* → *General* → *Target language* and select *VHDL*. This will create another implementation folder, where the bitstream and other files

will be generated. The reason why we should do this, is for the files Vivado will generate, as they can be in any language we target.

At the left side, under *IP Integrator*, click on *Create Block Design*, and then *OK*.

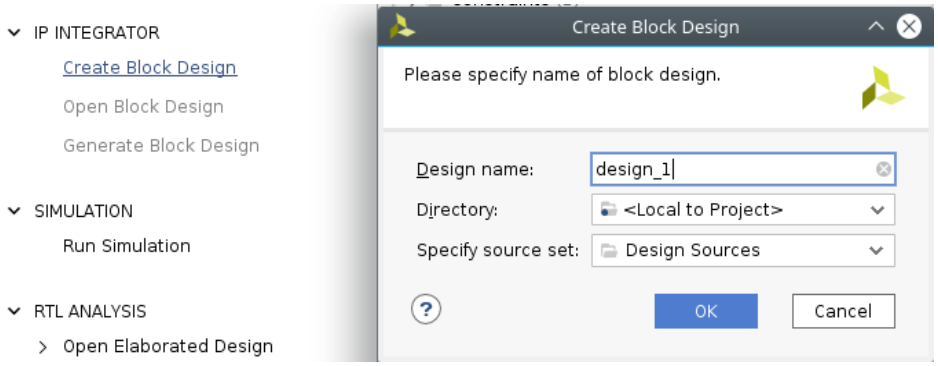


Figure 3.15: Create block design

A blank diagram should appear, and from here, we can add any IP included in IP Catalog. But, for now we will see how to take advantage of IP Integrator using our LED_Test code. Right click on the blank diagram, and click on *Add Module...* Select *LED_Test.vhd*, and *OK*.

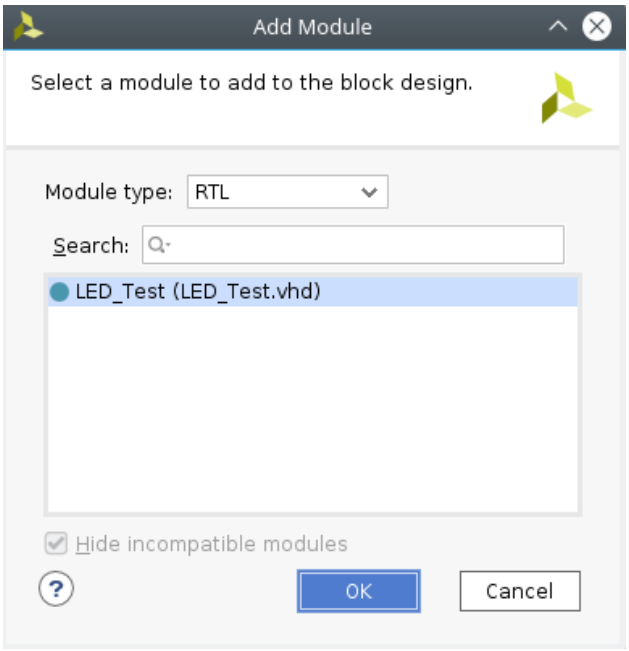


Figure 3.16: Add module in IPI

As you can see, the module has been integrated as a block in the diagram, with one input called *clk*, and an output called *led*, as we defined in VHDL.

Figure 3.17: Module in IPI

But that's not all. If you double click on it, you will see that the generic variable FREQ we created, is accessible for us to change its value. This is

the equivalent of the instantiation in VHDL, when you assign the value in the *generic map*.

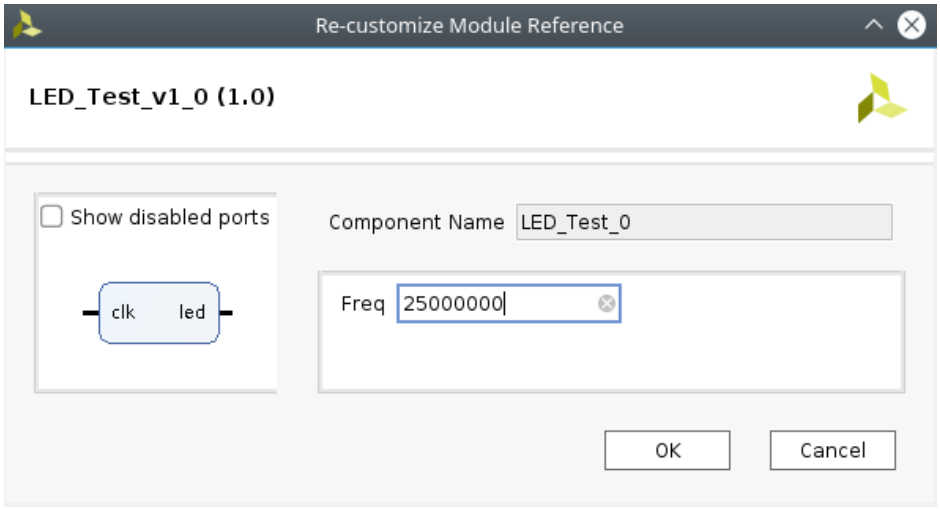


Figure 3.18: Access to generic variables

If we right click on *clk*, and *Make External*, an input port will be created, and connected to *clk*. It automatically gets the name *clk_0*, but we can modify it in the window *External Port Properties* after selecting the input port. Essentially, *clk_0* is the input port *clk* of *Top_Level* when we created it. So, as you see, IP Integrator is basically a top wrapper that instantiates every module you integrate in the diagram. I won't cover much more about IP Integrator at this point, but don't think that there is only one level of hierarchy, there can be as many as you please. Doing the same with *led*, we have now again a system with one input and one output.

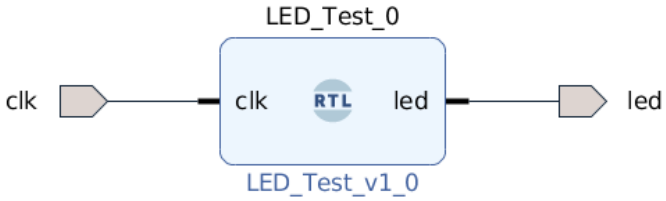


Figure 3.19: Full system in IPI

In order to build the bitstream, a *Top_Level.vhd* must exist. But we don't

have to create it, Vivado can do that for us. In the Sources window, right click on *design_1* (which is the diagram), and click on *Create HDL Wrapper*. Then, *Let Vivado manage wrapper and auto-update*, and *OK*.

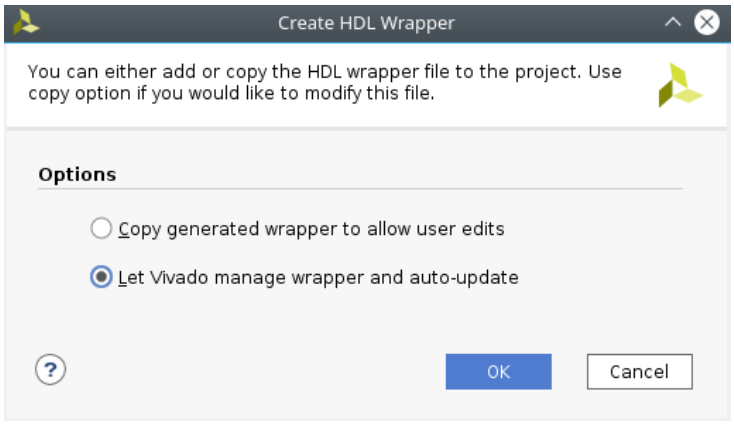


Figure 3.20: Generate wrapper

As you can see, a file called *design_1_wrapper* is created, in VHDL or Verilog, depending on the setting selected in *Settings* → *General* → *Target language* → *VHDL*. Opening this file, we can see that is almost exactly how we had defined *Top_Level* previously.

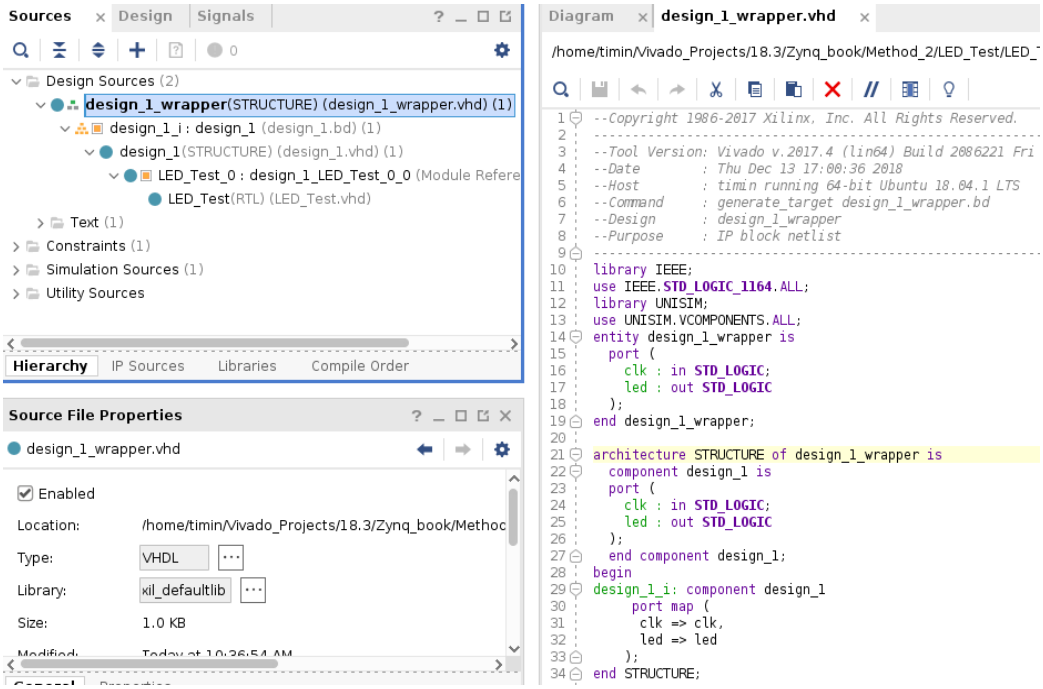


Figure 3.21: Wrapper code, hierarchical design

The constraints file doesn't need any change, as we declared the top ports with the same name. Clicking on *Generate Bitstream*, should generate a bitstream, that should work like in Method 1.

3.3 Method 3: Creating an packaging an IP

I just want you to think for a moment. In Method 1, we made a VHDL code that makes a LED blink. We are just using the PL for now. Later, we learnt how to create the same design using IP Integrator, which is an intuitive way of seeing the design in a graphical way. But what happens if our VHDL code has lots of inputs and outputs? In fact, what if we have several modules with interfaces that share inputs and outputs all over the place? In IP Integrator, if a VHDL module is imported, every single signal, either inputs or outputs, will be represented as an input or output port of the module.

Sometimes, it's nicer if some signals are grouped in one bus. For example, you would rather have an IP with an inout called I2C, which is a bus with two signals, SDA and SCL, before having an IP with 2 inouts, both SDA and SCL. It reduces space, it clarifies that those two signals belong to the same interface, and it standardises interfaces between IPs.

For our case, it's a bit pointless to create an IP, as the module we had can do perfectly the job, and having just 1 input and 1 output won't make any difference. But it's good to know that these tools are available to be used, to create IPs, and create interfaces for the IPs.

Removing everything, just leaving *LED_Test.vhd* from Method 1, and selecting the file, click on *Tools*, and *Create and Package New IP...*. Click on *Next*, and then select *Package your current project*.

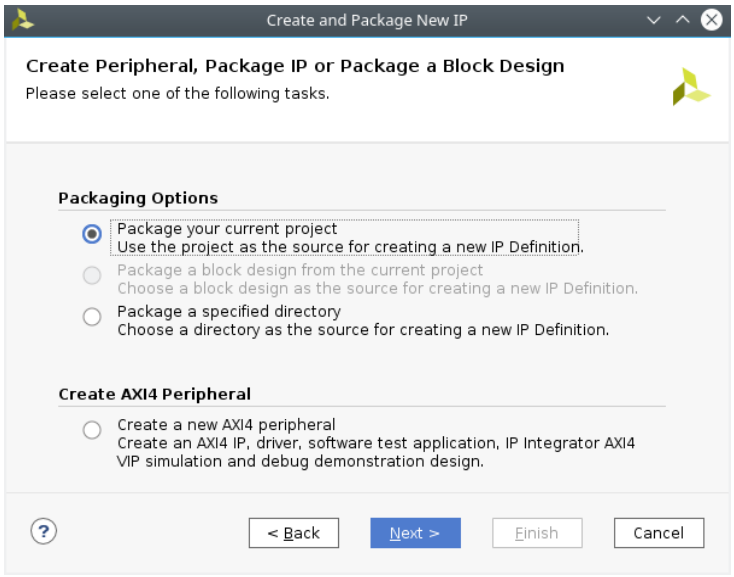


Figure 3.22: Create IP

Next, and then *Include IP generated files*. *Next*, and *Finish*.

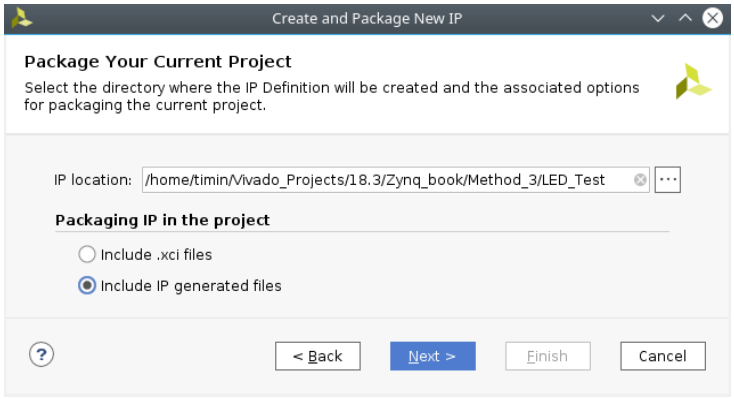


Figure 3.23: Include all files

In this new environment, we can define anything we want for our IP.

Go to *Ports and Interfaces*. In this case, *clk* has been detected as a clock interface, so Vivado automatically applies a clock interface on that port.

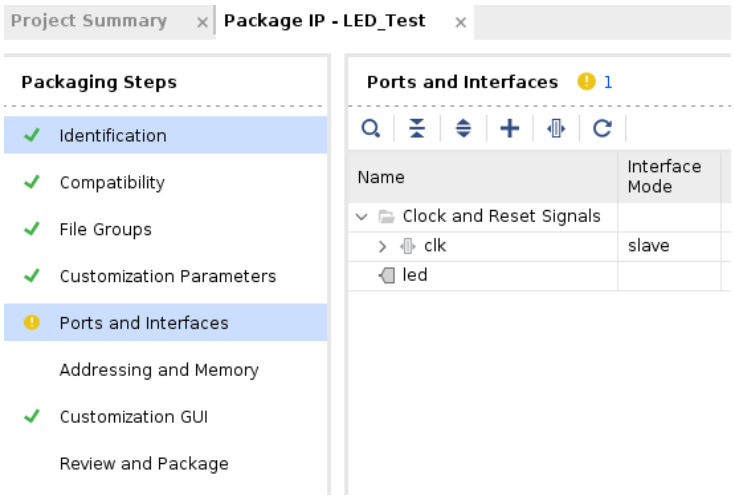


Figure 3.24: Port interface

Go to *Customization GUI*. Here we can see the generic variable *FREQ*. Right click on *Page 0*, and *Add group*. Add a group called *Parameters*, and put Freq inside it.

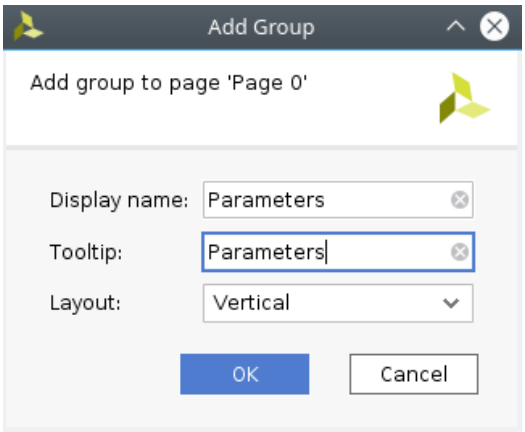


Figure 3.25: Add group

Right click again, and select *Add Text*. In *Display name*, write *Input Frequency*, and in *Text*, write *Determine the external clock's frequency in Hz*. Move it above *Freq*.

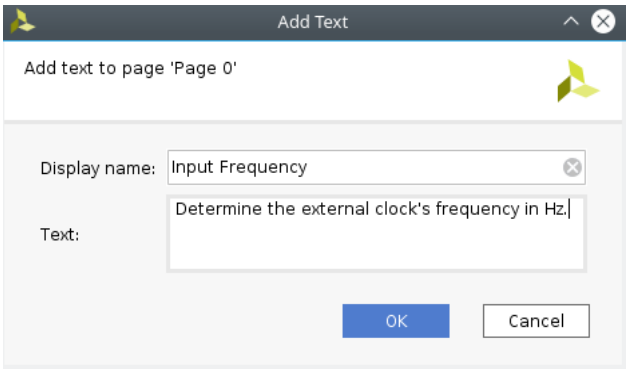


Figure 3.26: Add text

The layout should look like this:

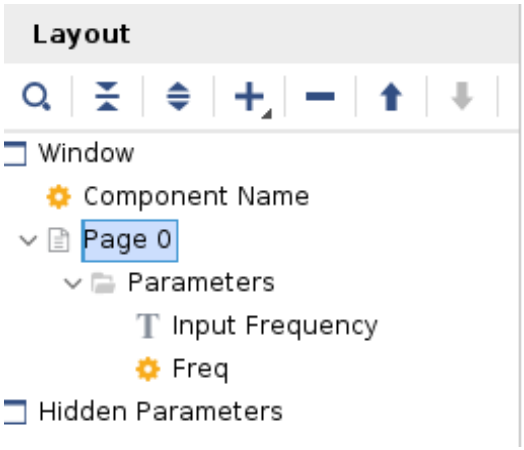


Figure 3.27: Layout of the page

Select *Freq*, and right click on it, selecting *Edit Parameter...*. Tick *Specify range*, change *Type* for *List of values→ Range of integers*. Put 0 in minimum, and 25000000 in maximum.

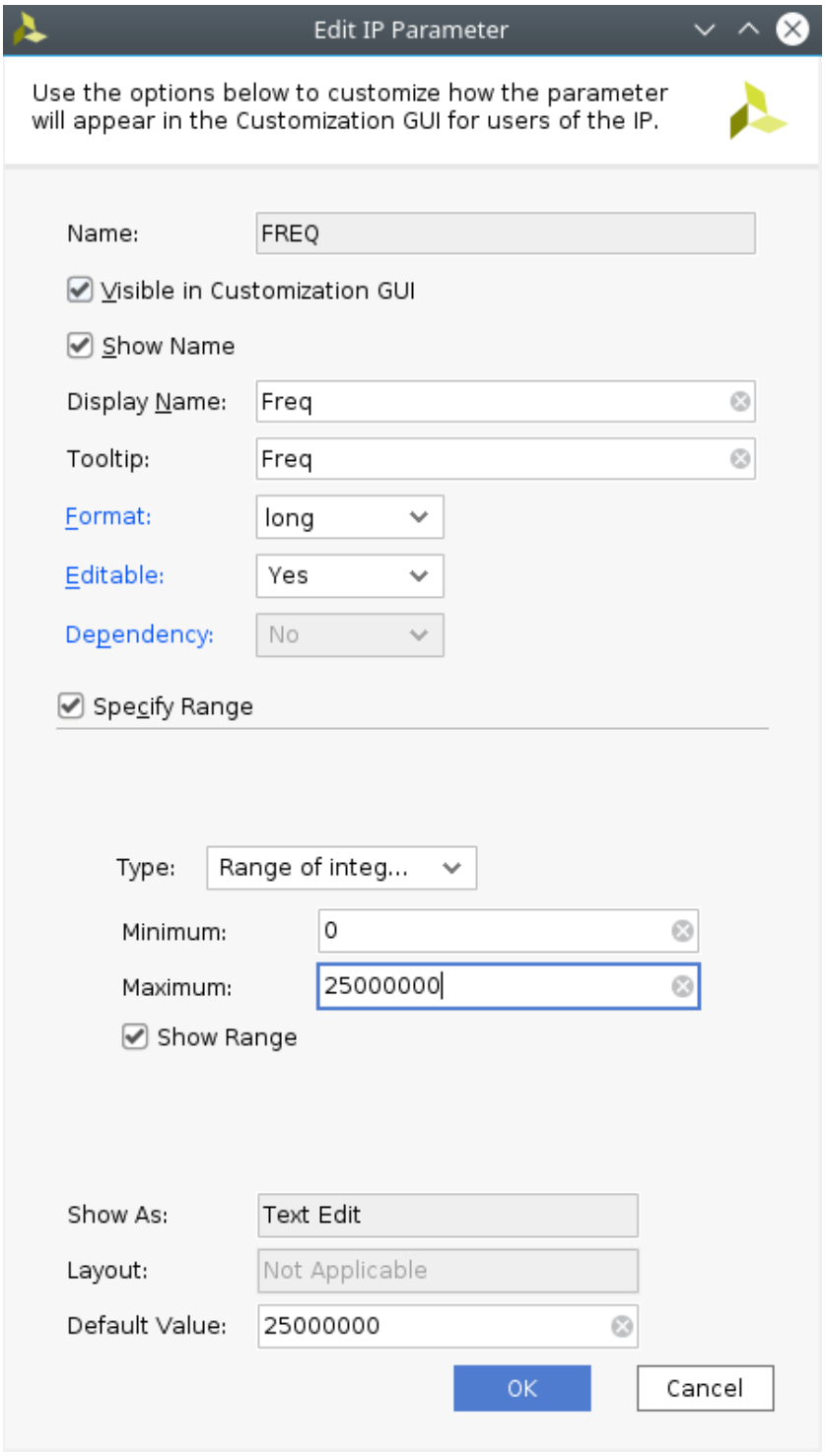


Figure 3.28: Frequency parameters

Go to *Review and Package* and click on *Edit Packaging Settings*. Tick *Create archive of IP*, *OK*.

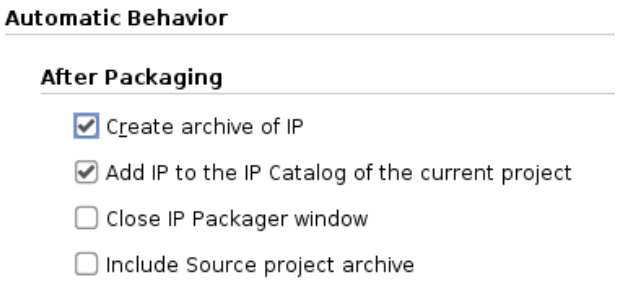


Figure 3.29: Generate archive of IP

Finally, *Package IP*.

The reason why *Create archive of IP* should be ticked, is because the IP will be generated within the project files, to be available in IP Catalog. But, in addition, the user has a compressed file with the IP, which can be extracted in a repository folder where we can store all our IPs.

Creating a folder called *Repo*, and placing the extracted files, the folder should look like Figure 3.30

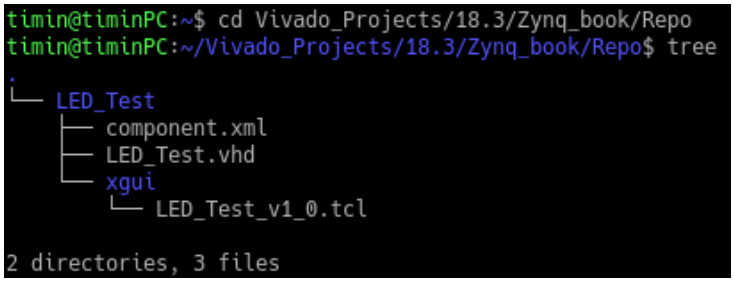


Figure 3.30: Custom IP Repository

Now you have a customised IP, so create a new blank block diagram in IP Integrator, as explained in Method 2. Right click, *Add IP*, and look for your IP.

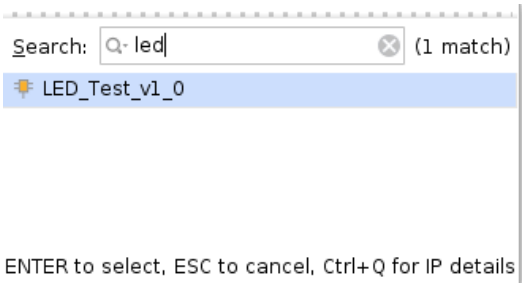


Figure 3.31: Search your IP

If you double click on the IP, you will see that the layout configuration has been applied. It shows the frequency range, and the description.

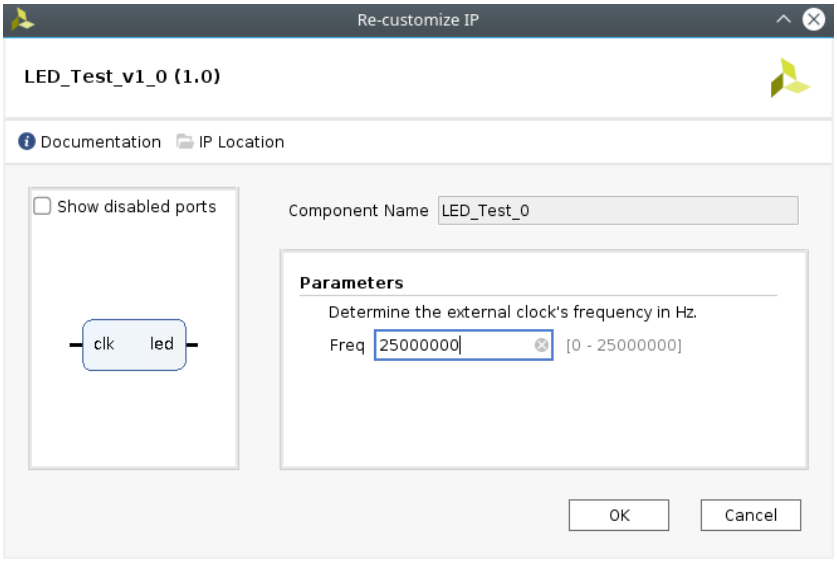


Figure 3.32: IP configuration

Make both ports external, and create an HDL wrapper.

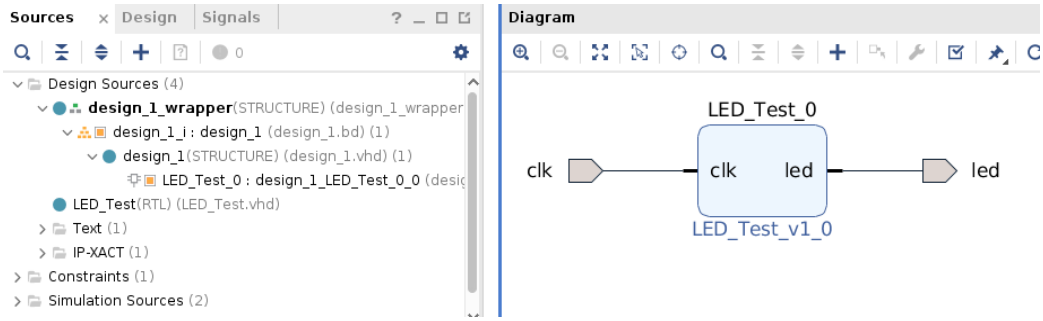


Figure 3.33: Overall project

Generate the bitstream, and again, this should work in the same way it does for Methods 1 and 2.

3.4 Method 4: Using the PS to provide clocking, standalone, using JTAG

So far, we have been using the PL (Programmable Logic) to achieve our goal. But the point of using a Zynq architecture is to take advantage of both the PL and the PS (Processing System). In order to run our IP, we were using an external clock from a signal generator, connected to an I/O pin of the FPGA. This method number 4, is an introduction to method number 5.

In this method, we will be using the same custom IP to toggle de LED, but providing a clock from the PS. This will be useful to introduce the PS, and buy time in next methods, to address different concepts.

As I said, to introduce the PS, we will just use it to provide the clock for our IP. This means that the PS won't be using any internal module or external memory, as the LED will toggle according to what it's implemented in the PL through our VHDL code in our custom IP. The PS will just provide the clock, and the PL resources used will be synchronous to it.

Create a new project, and before anything else, let's add our custom IP. To do so, click on *IP Catalog*, and right click anywhere in the list of folders. Click on *Add Repository...*, and select the *Repo* folder previously created, where *LED_Test* was placed.

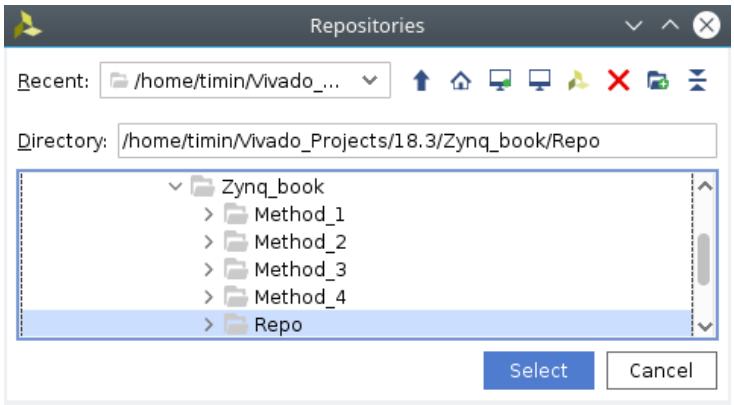


Figure 3.34: Repository in IP Catalog

Create a new block design in IP Integrator, and add the *Zynq Ultrascale+ MPSoC* IP.

Figure 3.35: Zynq Ultrascale+ MPSoC IP

Double click on it, and I will guide you to configure it tab by tab. In the first one (PS Block Design), you can see how the PS is structured (beware that the GPU -among other differences- is only present in TE0820-4EV, whereas in TE0820-4CG or 3EG, it is not):

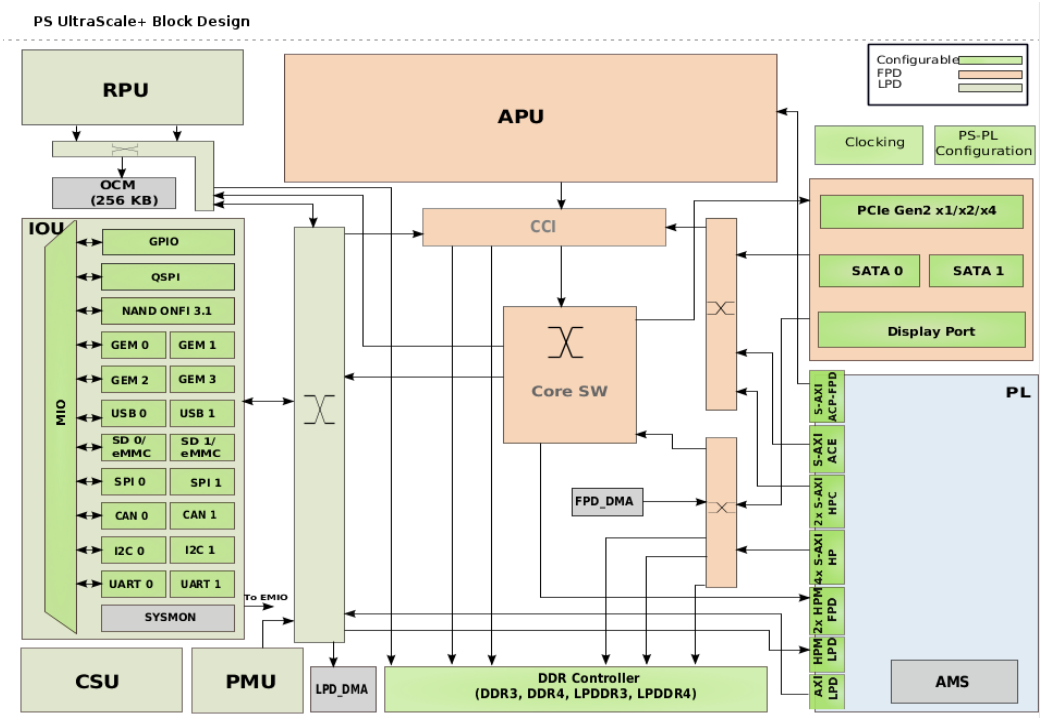


Figure 3.36: PS-PL Structure

In order to have a clock from the PS, it must boot, configuring the different

devices according to what we want to do. Even though we don't have any specific requirement, as we just need a clock, still, we need to have UART communication if we want to debug step by step from SDK. Keeping that in mind, go to the second tab, I/O Configuration.

The MIO Voltage Standards, select *LVCMOS18* for Bank0, Bank2 and Bank3, and leave *LVCMOS33* for Bank1. This sets the voltage I/O standards for the different banks of the FPGA.

At *Low Speed*, find *UART* in *I/O Peripherals*, and tick *UART 0*, selecting *MIO 30 .. 31*. This enables the UART interface, which is routed to the microUSB connector in the EMC2-DP carrier board, and allows us to debug.

In *Processing Unit*, select the watchdog timers (SWDT) and time counters (TTC). See Figure 34.

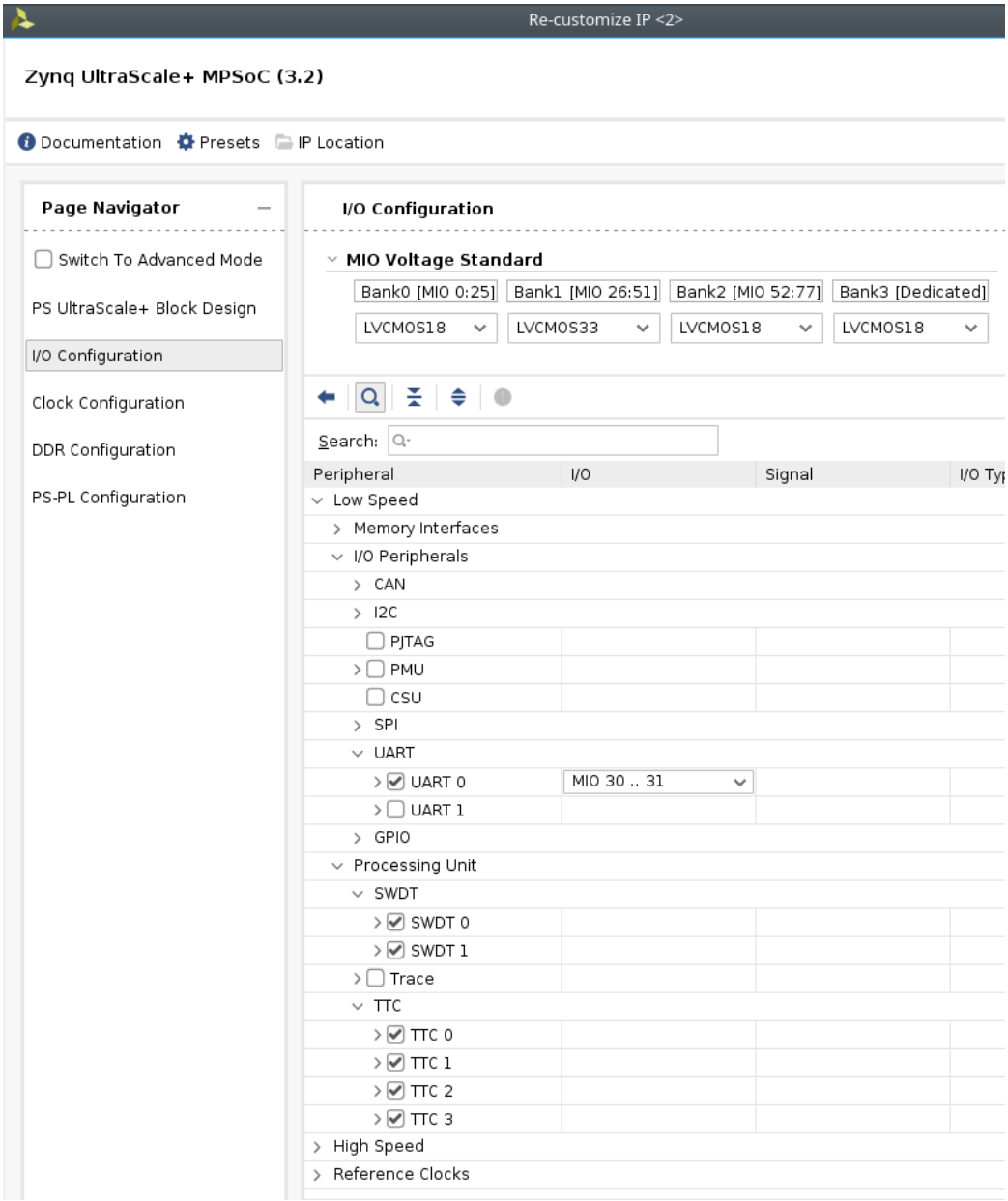


Figure 3.37: PS I/O Configuration

At the *Clock Configuration* tab, the clock we will use appears in *PL Fabric Clocks*, under *Low Power Domain Clocks*. Change it to 25MHz, as our custom IP accepts up to 25MHz, because that's the range we gave it.

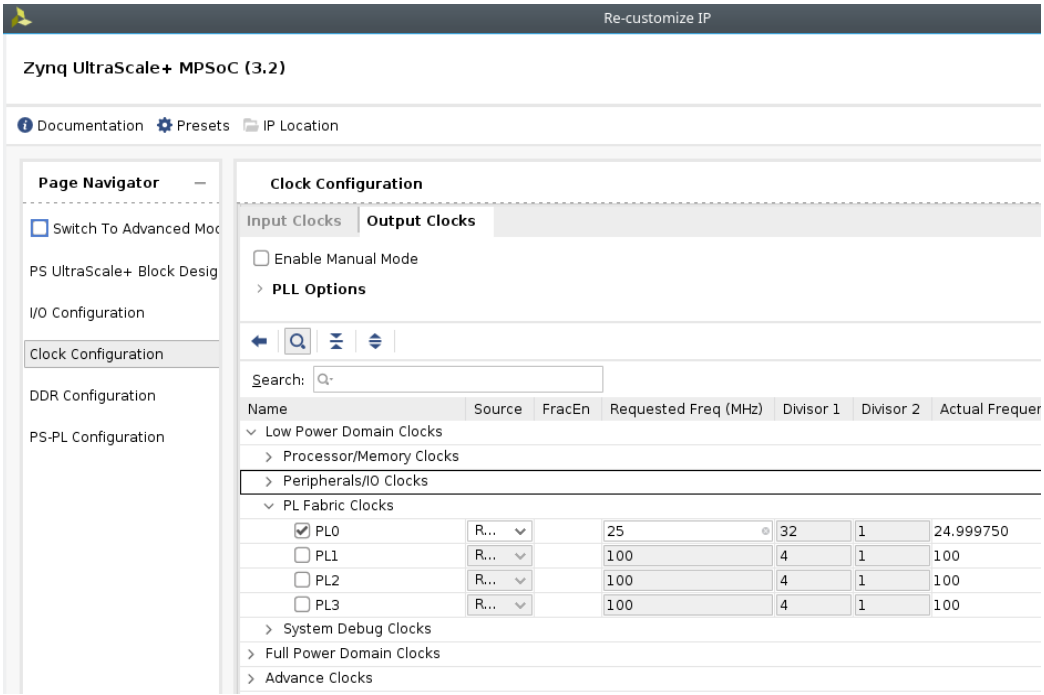


Figure 3.38: PS Clock Configuration

At *DDR Configuration*, just copy the parameters shown in Figures 3.39 and 3.40. This will configure the PS according to the external DDR memory devices for this specific hardware.

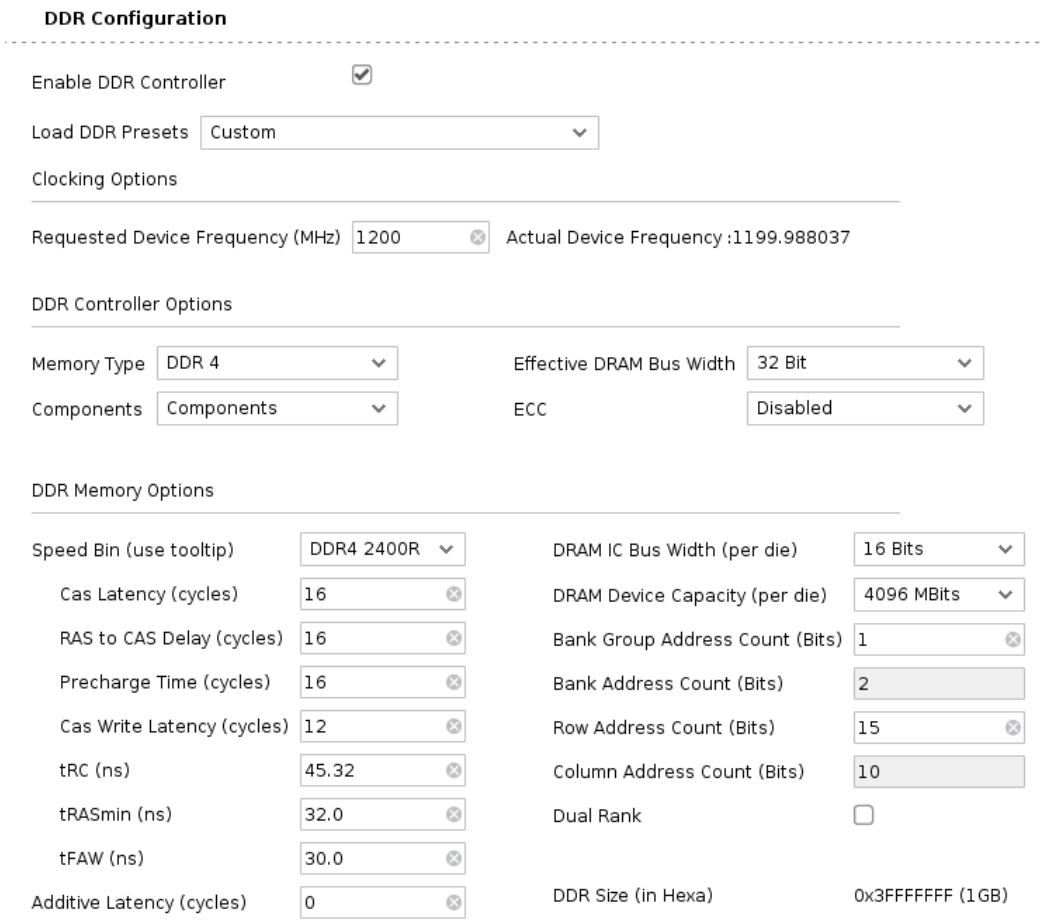


Figure 3.39: DDR Configuration (a)

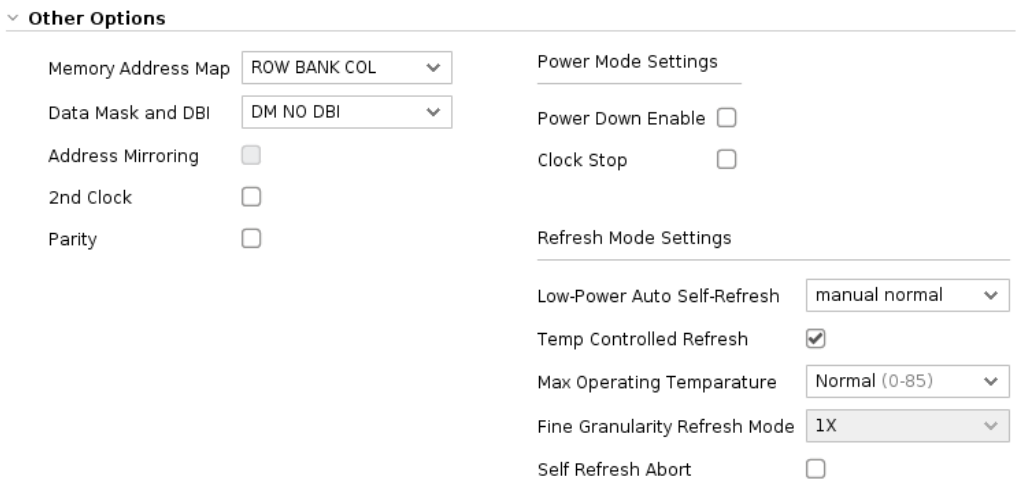


Figure 3.40: DDR Configuration (b)

Finally, click on *OK*, and add the custom IP, which should be available, since the repository was added in IP Catalog. Connect the clock accordingly, and make the led output external.

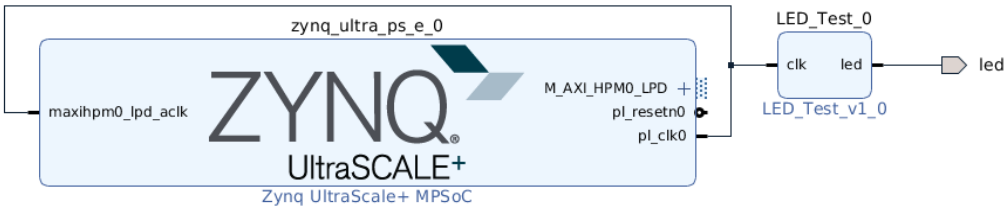


Figure 3.41: Block design with PS

For this method, there is no input clock, so the only required constraints are the ones related to the output led. Create a constraints file, and use this code:

```
##############################################################################

# LED
# PIN LOCATION
set_property PACKAGE_PIN B5 [get_ports led]
# I/O STANDARD
```

```
set_property IOSTANDARD LVCMOS18 [get_ports led]

##############################################################################
```

Create an HDL wrapper, and build the bitstream. When the process finishes, click on *File* → *Export* → *Export Hardware....* Select *Include Bitstream* and *OK*.

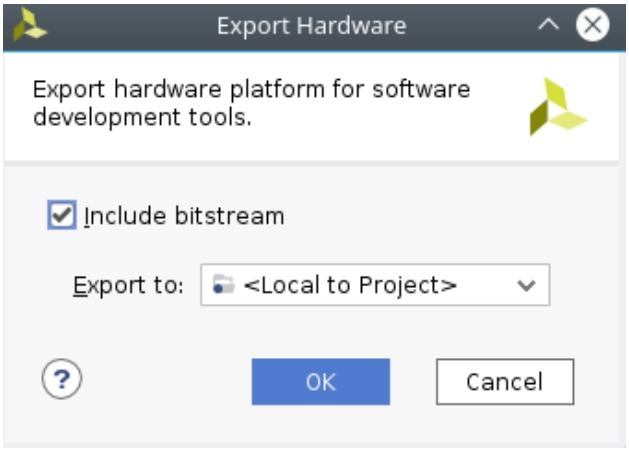


Figure 3.42: Export design

Just to leave things clear before continuing. We've just created a project, which uses the PS to generate the clock for our custom IP, which divides down a 25MHz clock to toggle a LED. In order for the PS to generate the clock, it has to be initialised. We will do that manually with Xilinx SDK.

Click *File* → *Launch SDK*, and this will launch the tool.

As you can see, the tool already generated a hardware platform. What does this mean? As it was explained in Method 1, following the hierarchy of the project files, at this point we've exported a design, which is summarised in a file with *hdf* extension, within the <ProjectName>.sdk folder. This file is enough for Xilinx SDK to generate a basic platform, which defines the different components described. If you have more interest in this, read the *psu_init.c* file within the platform. As you will see, there are a lot of definitions, that are applied in the function *psu_init(void)*, incldiung the MIOs, peripherals, etc.

This platform is automatically generated, but if you want, it's possible to create it from scratch. To do so, go to *File* → *New* → *Project....* Under *Xilinx*, you can find *Hardware Platform Specification*. The only requirement to build it, is to provide the *.hdf* file of your project.

As we just need the PS to be initialised, the project we need is FSBL, which stands for First Stage Boot Loader. In the same way as before, *File → New*, and from here, we can directly choose *Application Project*. Otherwise, it can be found under *Xilinx* where the Hardware Platform Specification project was.

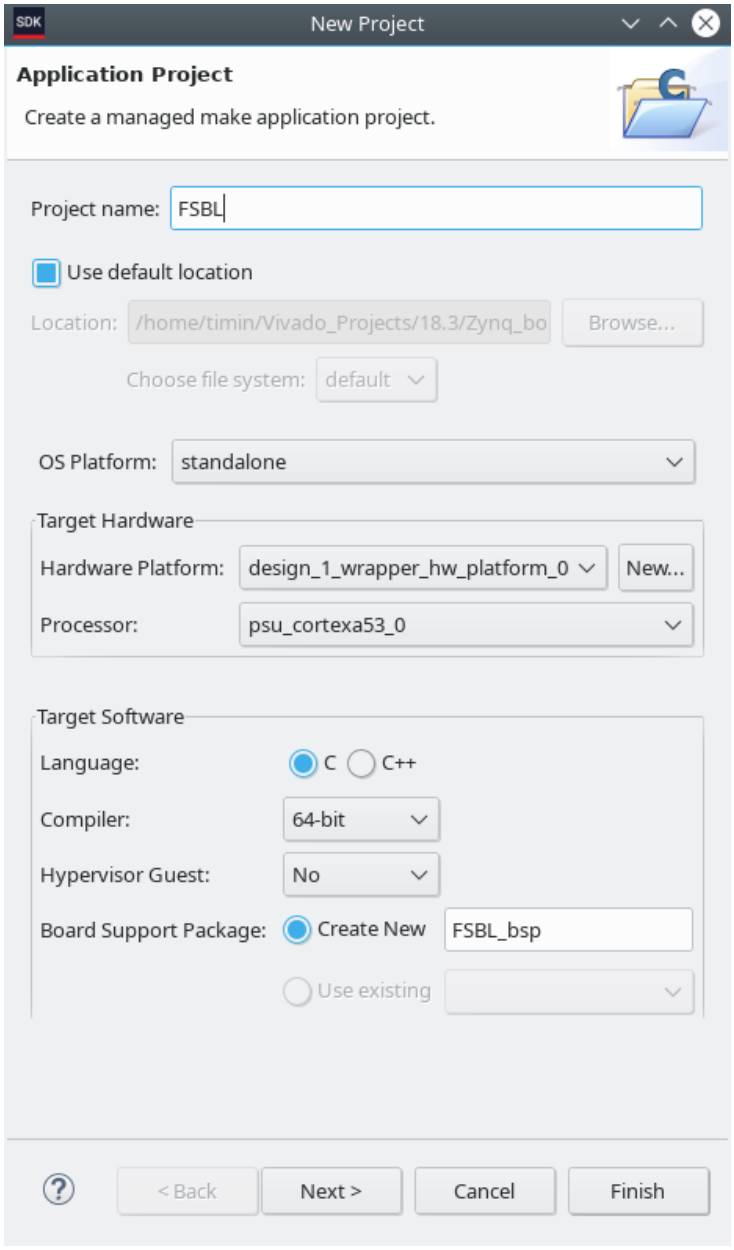


Figure 3.43: FSBL Creation

Give the project a name, FSBL for example, and select *Zynq MP FSBL* before clicking on *Finish*.

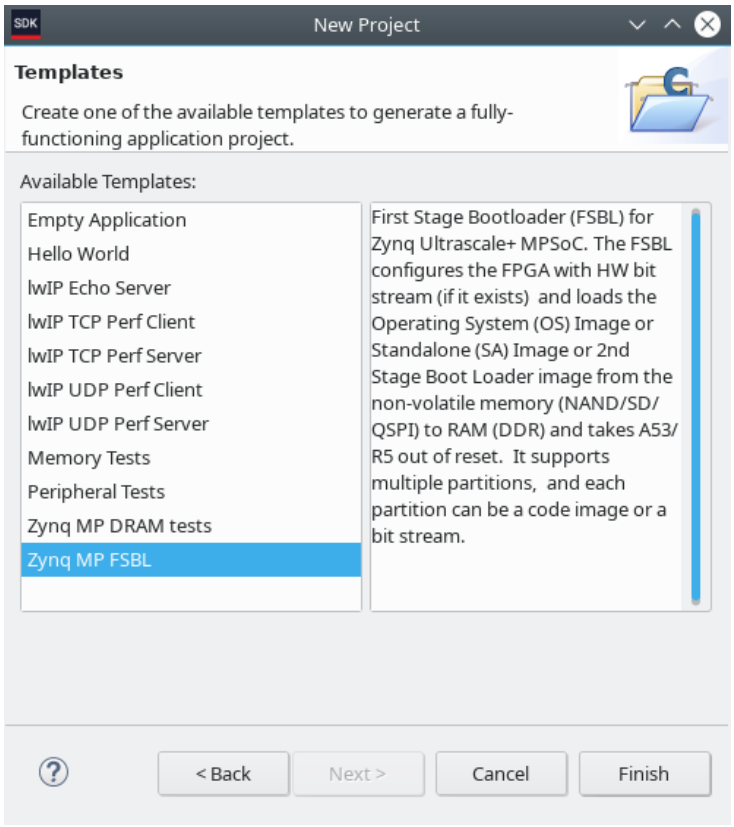


Figure 3.44: FSBL Creation

Now, in your workspace there should be an FSBL project along with the Hardware Platform Specification.

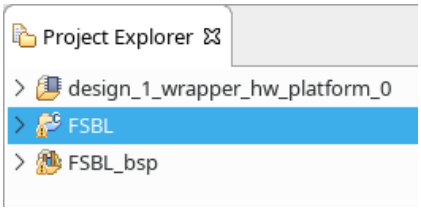


Figure 3.45: SDK workspace

The project might build itself. Otherwise, right click on it, and click on *Build Project*. You should see something like this in the console:

```
Finished building: FSBL.elf.size
16:07:12 Build Finished (took 4s.991ms)
```

It's time to initialise the PS, and configure the FPGA. Assumming that the hardware is connected through JTAG, go to *Xilinx* → *Program FPGA*, and press *Program*.

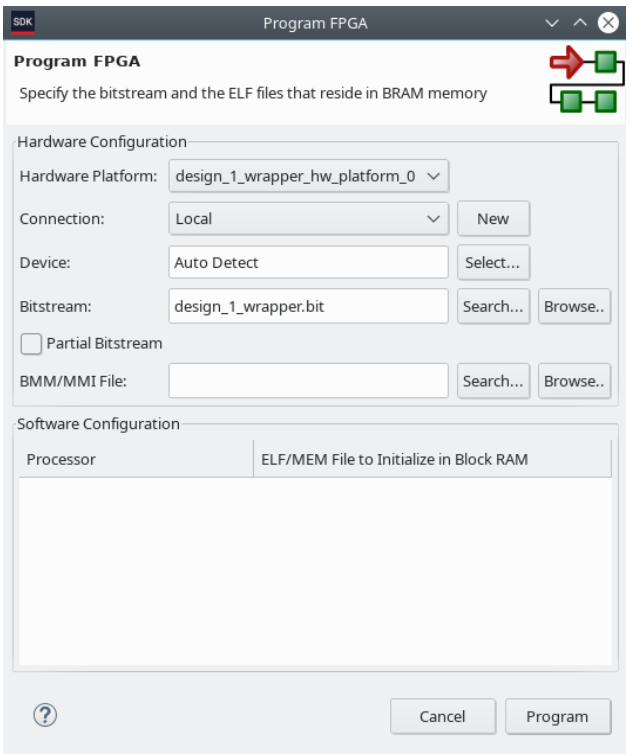


Figure 3.46: SDK Program FPGA

Being the PL programmed, the only thing left, is initialise the PS. To do so, right click on the FSBL project, and select *Debug As* → *Launch on Hardware (System Debugger)*. Xilinx SDK will ask you to change the perspective, and you can say *Yes* to it.

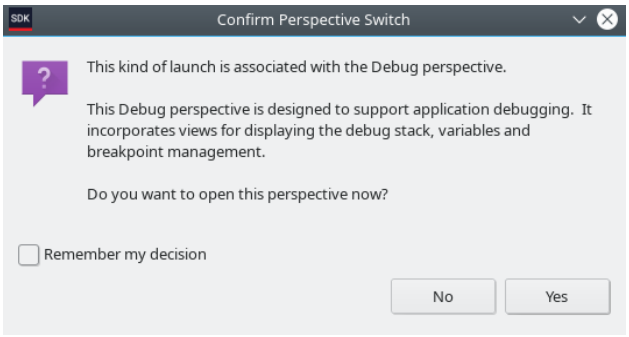


Figure 3.47: C/C++ Perspective

The program will launch, initialising the ARM core.

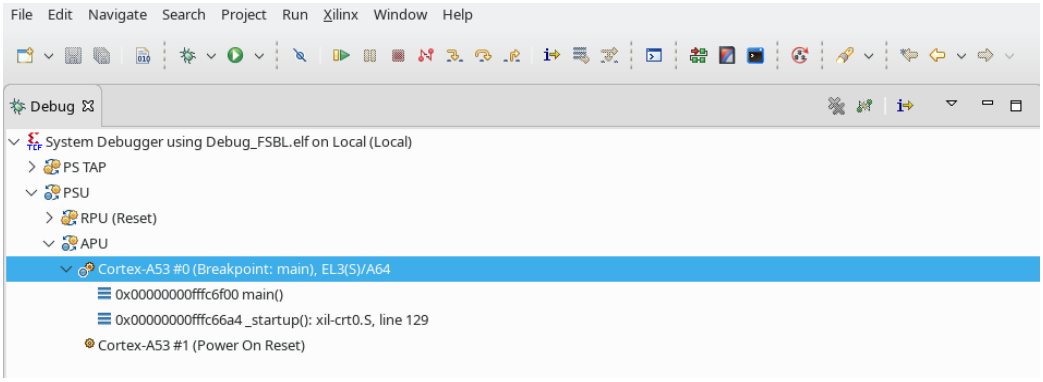


Figure 3.48: FSBL launched

You won't even need to run the FSBL code, the internal PLL will generate the clock we need as soon as the program is launched, and you will see the LED blinking as expected.

3.5 Method 5: Using C, programming the PS/PL, standalone, using JTAG

We have covered some methods to blink a LED, from a low-level perspective. I want to clarify that the reason why I used such methods was to set your minds into some basic concepts, before actually getting to know how to, with any piece of hardware based on Zynq, get a LED blinking in maybe five or ten minutes. The methods above are nice for specific projects, specific design, when addressing custom devices or developing around certain requirements, especially when the PL is the main character, or even when you are using an FPGA, with no hard or virtual processors.

However, in the Zynq architecture, we have one or more ARM cores available, with a programmable logic with plenty of resources for us to use. How does the communication between this two different environments work?

The protocol developed by ARM, used in this architecture, is AXI. I won't dig too deep into AXI, but there are some concepts that will help you out from now on, if for any reason you see AXI, and wonder what it is, or if it scares you.

The reason why AXI is important for us, is because it's our channel to do something very important in any embedded system: transfer information. The concepts I want you to have very clear about AXI are these:

- AXI transactions happen in two different ways (in general)
  - Memory mapped
  - Stream
- AXI transactions happen between a master and a slave
- Assuming the two concepts above, it's necessary to understand that, sometimes, AXI transfers require a protocol conversion

In order to understand the concepts, let's jump into Vivado, and work through them, while learning the 5th method.

Following the same steps as in method 4, add a Zynq Ultrascale+ MPSoC block, and configure it. You should have a similar design as in Figure 3.41, although you don't need the custom IP or the led port, just the Zynq IP configured.

At this point, we will try to make the led blink, using the ARM core. The

led is connected to an I/O pin of the PL or FPGA, so, how to do the link between the ARM core and the I/O pin? With AXI.

The block we will use is called AXI_GPIO. Right click on the block design window, *Add IP*, and select *AXI_GPIO*. At this point, you should have something similar to this:

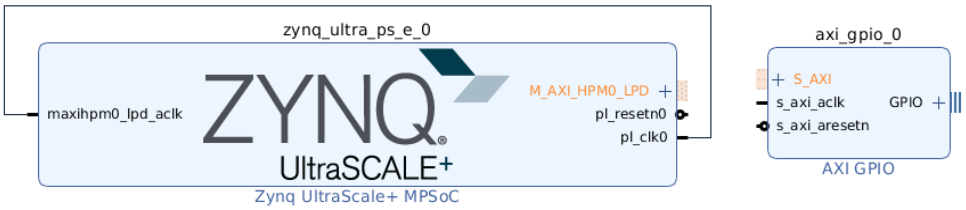


Figure 3.49: Zynq and GPIO IPs

I highlighted the ports related to AXI communication. The first thing I thought the first time I used AXI was, why can't I just connect them straight away? Well, because there is not only one type of AXI transaction, and the communication can be defined in many ways, not only depending on if it is a memory mapped transaction or a stream of data, but also the width of the bus, and the allocation of the data within the PS (use of cached memory, etc). Because of this, every relationship between a master (Zynq) and a slave (GPIO block) occurs through an IP called AXI_Interconnect.

Add an interconnect block to the design:

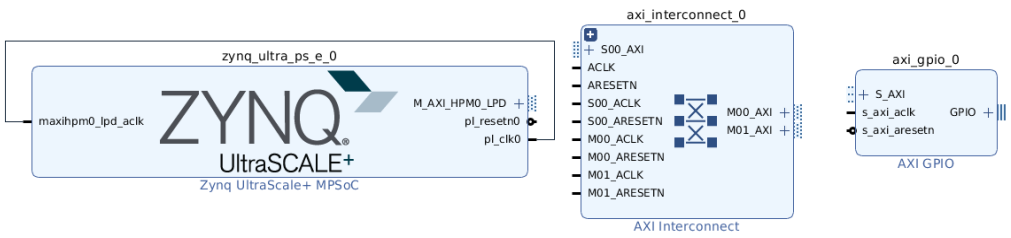


Figure 3.50: AXI Interconnect

What does adding an interconnect mean at this point? Well, instead of

having a system where the Zynq PS is the master, and the AXI GPIO block is the slave, now we have a system where the Zynq PS is the master, an interconnect block is the slave, and also, this interconnect is a master, and the GPIO block is the slave. Thus, indirectly, the Zynq PS is master of the GPIO block.

Why is this an advantage?

- The PS is going to work under a clock domain, in this case a clock generated by an internal PLL in the device, but, what if we wanted to run our GPIO block with a different clock? The interconnect is useful for separating these two clock domains, without introducing meta-stability issues.
- The PS is not limited to be the master of one GPIO block. It can master many slaves through the same interconnect block (up to 16 per interconnect). Likewise, the PS can be a slave, and be mastered by other AXI blocks in the design.

Now you can try to connect the Zynq PS to the interconnect, and the interconnect to the AXI GPIO block. You will see that it is possible.

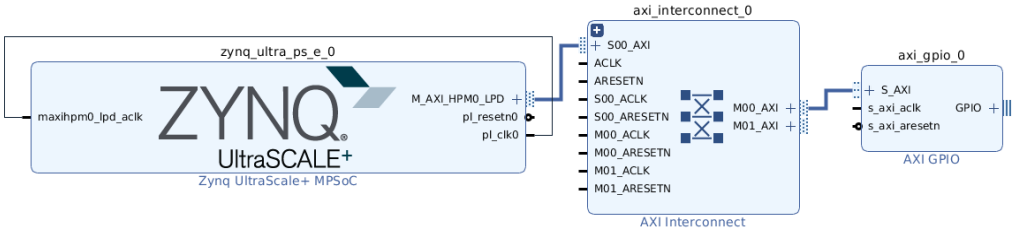


Figure 3.51: Connect the AXI Interconnect

By default, the interconnect comes with a configuration of 1 slave port and 2 master ports. As the Zynq PS is master and the interconnect is a slave, and also, the interconnect is a master, and the AXI GPIO is a slave, the interconnect requires 1 master port and 1 slave port. Double click on it, and configure it.

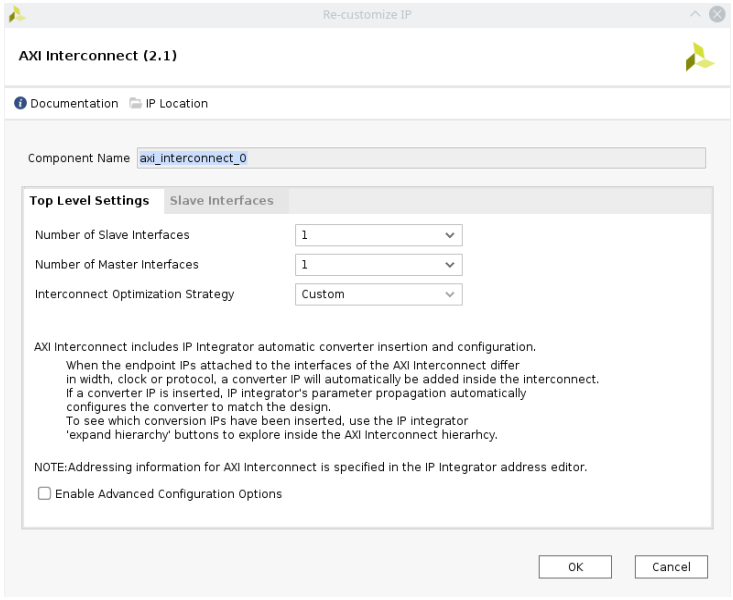


Figure 3.52: Configure the AXI Interconnect

Now, it would make sense to connect all the clock pins together:

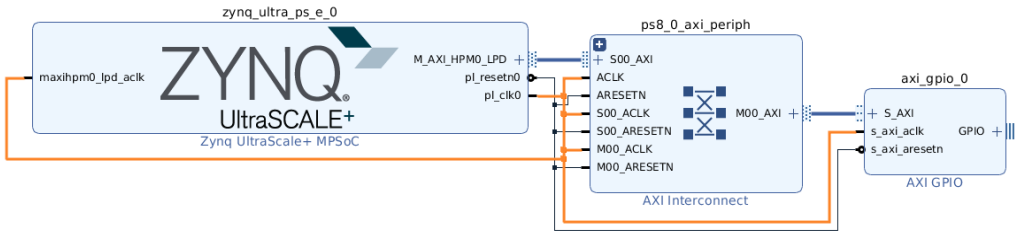


Figure 3.53: Connect the clocks

And the reset pins:

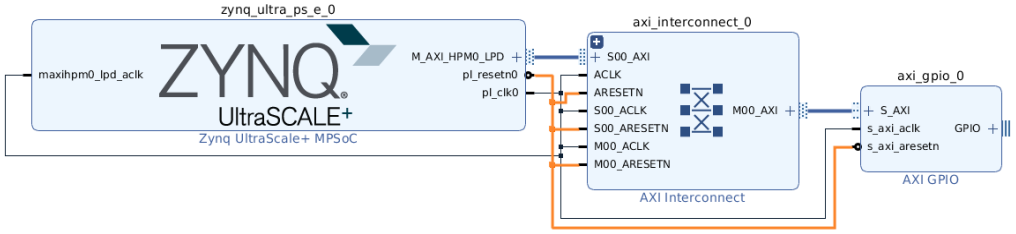


Figure 3.54: Connect the resets

But, is this correct? It actually is, assuming you are happy with all your blocks in the PL having an asynchronous reset connected to them. Vivado will warn you about this if you don't do anything about it, but the IP Xilinx provides for the reset control is the Processor System Reset. Add one to your design.

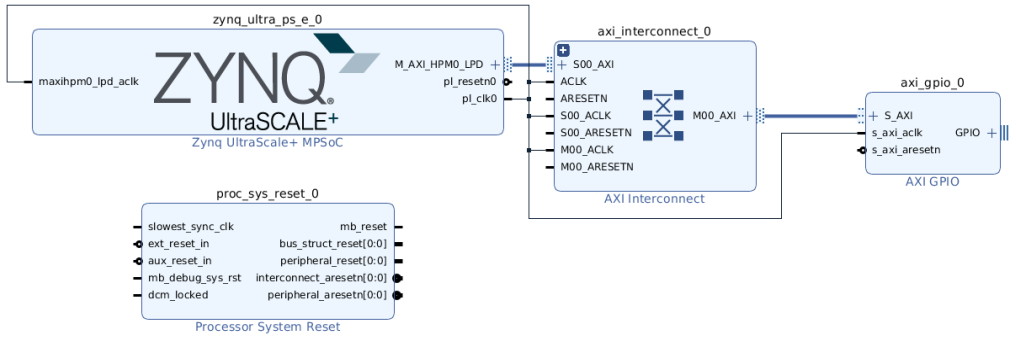


Figure 3.55: Add Processor System Reset

Now connect the Zynq PS reset to the external reset pin of the PSR block, and the peripheral reset to the interconnect and GPIO blocks.

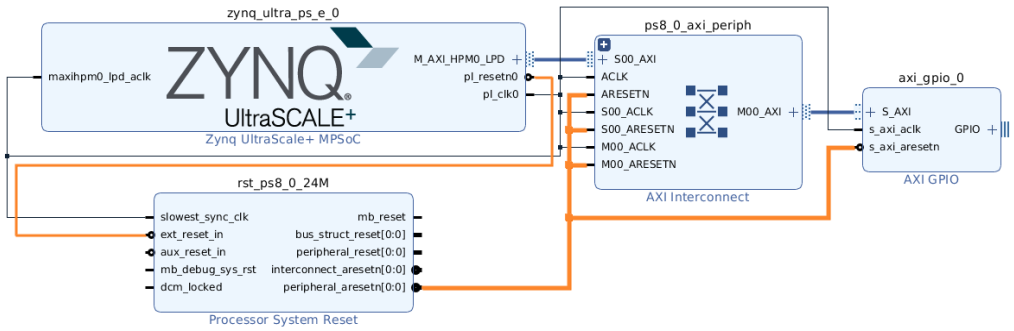


Figure 3.56: Add Processor System Reset

The only thing left is the led port. Before, we were using an output port, which corresponds to an output (std_logic in VHDL) from our code. For the GPIO block, each GPIO interface defined can be either an input or an output. If you are not familiar with tri-state buffers, it is basically a signal that can be buffered out from one system to another, but also, the first system is able to set the output buffer in high impedance mode (tri-state), so that the same signal can be read back from the second system:

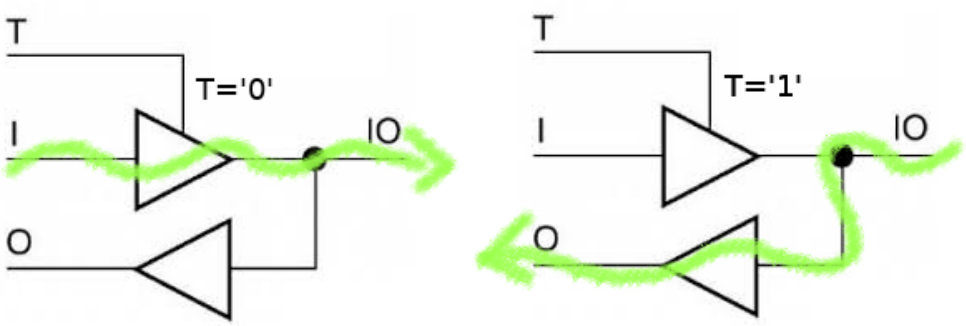


Figure 3.57: Output and Input configurations respectively

The signals in the GPIO port:

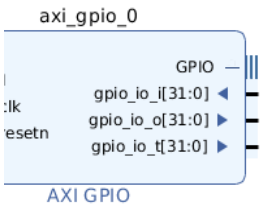


Figure 3.58: GPIO I/O port

As you can see, there is an input (i), output (o) and tri-state signal (t), for a bus of 32 signals. As we don't have 32 leds, but just 1, and also, as we don't want to read back from a led, we can configure the AXI GPIO block with just one output.

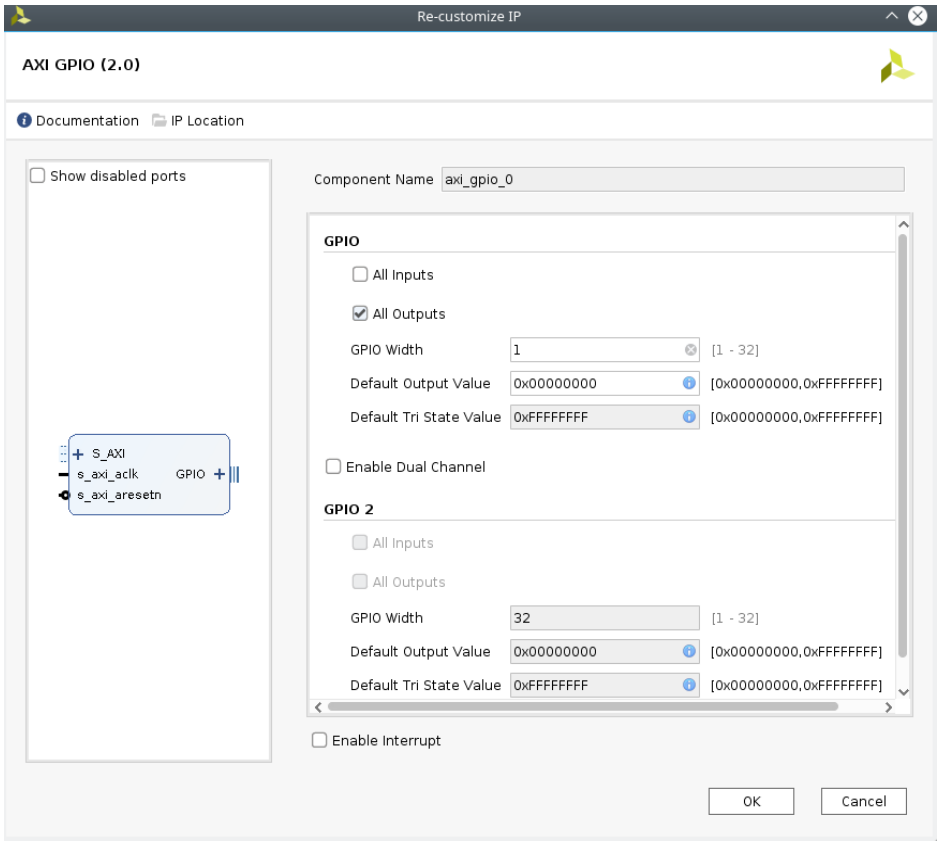


Figure 3.59: GPIO configuration

When you press *OK*, you will see that the GPIO has been configured accordingly,

and there is only one output (o):

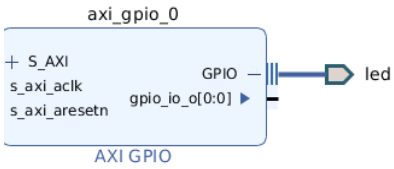


Figure 3.60: GPIO output

Making the port external, and changing its name to *led*, you should have your design finished:

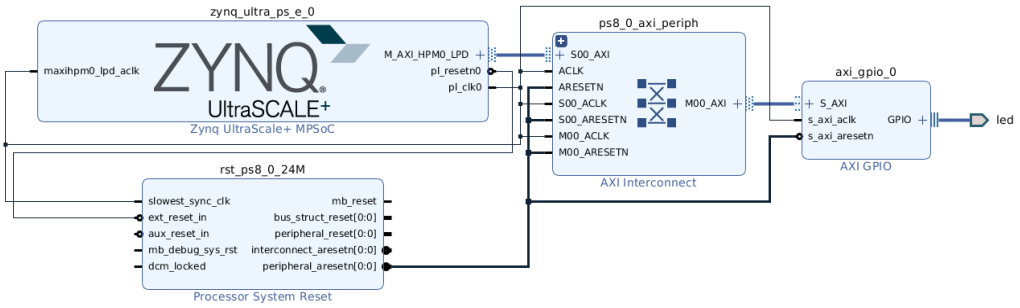


Figure 3.61: Design using AXI GPIO

Now, let's imagine that the Zynq PS is master of 10 slaves through the interconnect. How does the master know the rules to talk to one or another? Each slave has an address range assigned in memory, to which the processor can talk to. (this is a memory mapped interface). The interconnect is the one in charge to send the information to the corresponding slave.

As a brief thought, although it's not applicable in this example, there are systems that transmit data (converters, video...) which is treated as a stream, and therefore, it doesn't require memory mapping. This type of interfaces are AXI Stream interfaces, and, if for any reason you need to move that stream of data into memory, a conversion between AXI and AXI Stream must be done. Blocks like DMA (or VDMA for video stream data) can do this for you.

Back to our example, we have to define the memory ranges. Go to *Address Editor* (tab next to *Diagram*), and right click on *Unmapped Slaves*. Select *Auto Assign Address*, and a memory range will be automatically given.

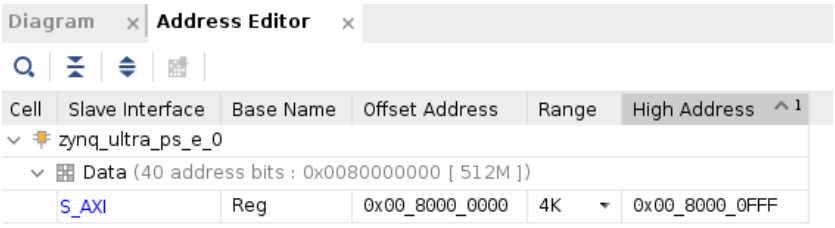


Figure 3.62: Address assignment

Go to *Tools* → *Validate Design*. You will see that there are not errors.

Figure 3.63: Validate design

Now, I want to show you something important. Delete the design in a way you are back to the state in Figure 3.49. Why? Because all the steps we have just followed, can be done automatically! Sometimes, it's not wise to do everything with the automation tools, but in this case, this design can be done with a couple of clicks. Having added the Zynq Ultrascale+ MPSoC and AXI GPIO IPs (and configured the ZU+ IP), click on *Run Connection Automation*. Select *All Automation*, and *OK*.

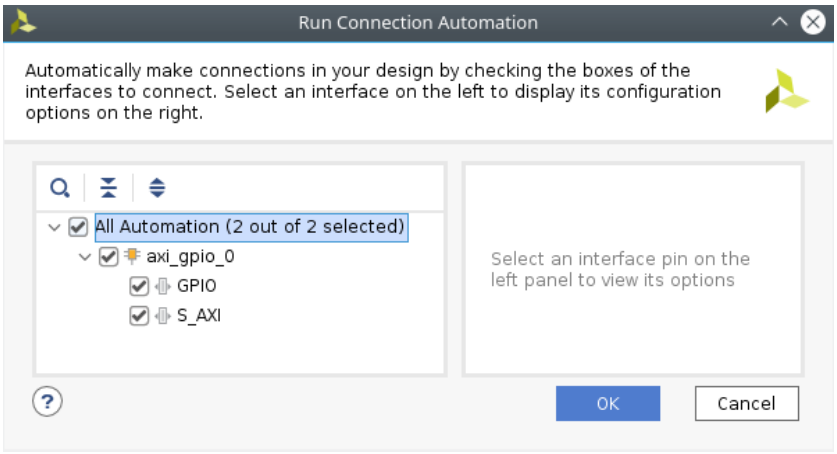


Figure 3.64: Automate the connections

This will reproduce the exact design we did before, including the address editor (maybe you will have to rename the GPIO output port). At this point, the only thing to modify, is the constraints file. Because the GPIO block can define its ports as inputs or outputs, they are defined as vectors. For signals like ours (single signal), is a vector of 1 position (0 to 0). This slightly changes our constraints file, which should be in this case:

```
##############################################################################

# LED
# PIN LOCATION
set_property PACKAGE_PIN B5 [get_ports {led_tri_o[0]}]
# I/O STANDARD
set_property IOSTANDARD LVCMOS18 [get_ports {led_tri_o[0]}]

##############################################################################
```

Create an HDL wrapper, build the bitstream, and export the design. Launch SDK.

Having SDK open, go to *File* → *New* → *Application Project*. Give it a name, and *Next*. Select *Hello World*, and *Finish*.

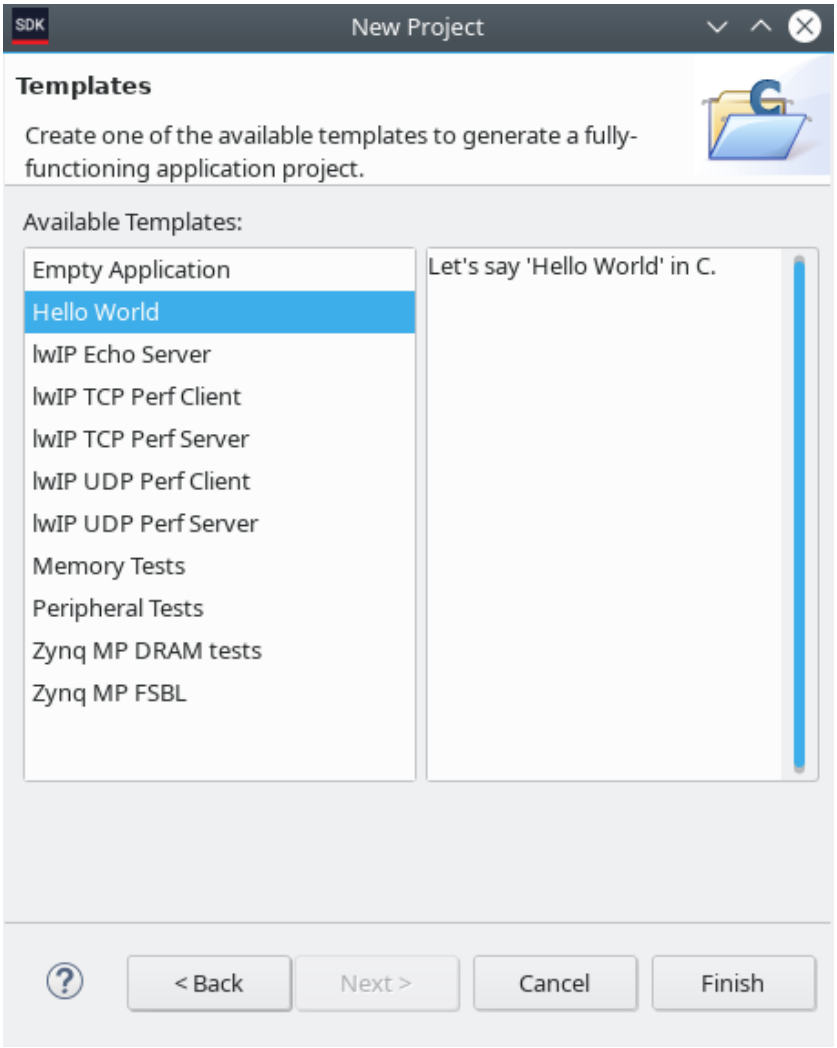


Figure 3.65: Hello World project

Having the project in the workspace, inside *src*, you will find *helloworld.c*, whose code is as follows:

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


int main()
{
    init_platform();
```

```
    print("Hello World\n\r");

    cleanup_platform();
    return 0;
}
```

From this moment, we are going to work in C/C++. But, how do we relate what we have designed in Vivado with our code in C?

The simplest approach is using the Xilinx standalone drivers. You will see that getting a led blinking is very straight forward, as long as you know where to search things.

Along with the application project created, a .bsp (board support package) project was created. Inside it, there is a file called *system.mss*. Double click on it, and you will see that there is a list of components, some of them accessible (the ones that have been added in Vivado), and some not accessible (the ones that we didn't add).

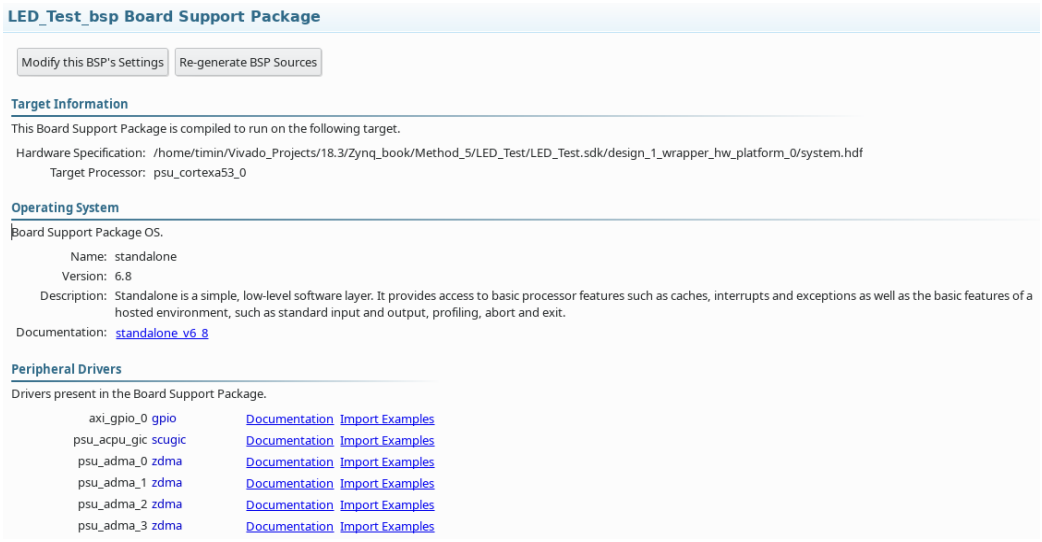


Figure 3.66: Standalone drivers

Also, inside the hardware platform project automatically created by SDK (otherwise, it can be created manually), you can find *system.hdf*. If you open it, and look up our AXI GPIO block, you can see that the addresses set in Vivado can be verified here.

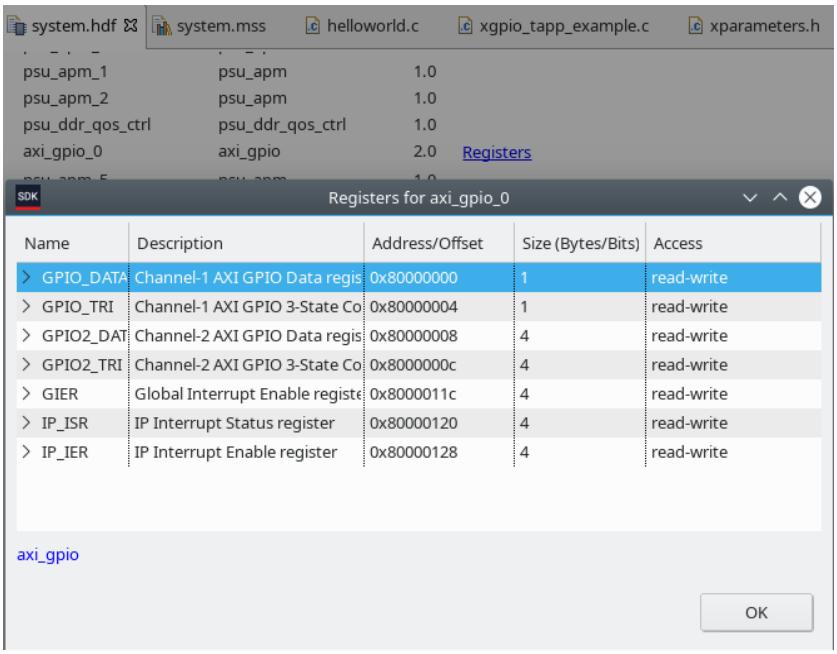


Figure 3.67: AXI GPIO in hardware platform

Back to the standalone drivers in *system.mss*, click on *Import Examples*, at the axi_gpio's row. Select *xgpio_tapp_example*, and *OK*. This will import an example project, from which we can take useful functions. In *src*, open *xgpio_tapp_example.c*.

These are the pieces of code we will be interested in:

- #include "xparameters.h".

  This file, if you look it up in *<ProjectName>.bsp/psu_cortexa53_0/include/xparameters.h* contains all the information of all the peripherals involved in the design. In fact, if you open it, and search (Control + F) *AXI_GPIO_0*, you will find this piece of code:

```
/* Definitions for driver GPIO */
#define XPAR_XGPIO_NUM_INSTANCES 1

/* Definitions for peripheral AXI_GPIO_0 */
#define XPAR_AXI_GPIO_0_BASEADDR 0x80000000
#define XPAR_AXI_GPIO_0_HIGHADDR 0x80000FFF
#define XPAR_AXI_GPIO_0_DEVICE_ID 0
#define XPAR_AXI_GPIO_0_INTERRUPT_PRESENT 0
#define XPAR_AXI_GPIO_0_IS_DUAL 0
```

  Again, the base and high address values from Vivado appear here,

and, what we will need for all our functions, the device id: #define XPAR_AXI_GPIO_0_DEVICE_ID 0

- #include "xgpio.h"

  This file contains the definitions of the functions used later on in the example project.

- XGpio_Initialize

  This function initialises the GPIO driver, assigning the corresponding device ID. If for example we had many AXI GPIO blocks in our design, this is the function that determines that the ARM core is going to talk to a specific one. Remember, the device ID is going to point to the address range we defined!

- XGpio_SetDataDirection

  Do you remember that the GPIO port could be an input or an output? As we defined in Vivado to just use our port as an output, this step is not really needed, but with this function we are able to set by software if we want to read or write to the port.

- XGpio_DiscreteWrite

  With this function, we will make the led blink!

So, open your application, and change your *helloworld.c* code (you can change the file name too if you want) for this one:

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

/*
 * The following are declared globally so they are zeroed and so they are
 * easily accessible from a debugger
 */
XGpio GpioOutput; /* The driver instance for GPIO Device configured as O/P */

int main()
{
        int Status;
        /*
        * Initialize the GPIO driver so that it's ready to use,
        * specify the device ID that is generated in xparameters.h
        */
```


```
        Status = XGpio_Initialize(&GpioOutput, XPAR_AXI_GPIO_0_DEVICE_ID);
        if (Status != XST_SUCCESS)  {
                return XST_FAILURE;
        }

        /* Set the direction for all signals to be outputs */
        XGpio_SetDataDirection(&GpioOutput, 1, 0x0);

        while(1){
                /* Set the GPIO outputs to low */
                XGpio_DiscreteWrite(&GpioOutput, 1, 0x0);
                /* Wait a second */
                sleep(1);
                /* Set the GPIO Output to High */
                XGpio_DiscreteWrite(&GpioOutput, 1, 0x1);
                /* Wait a second */
                sleep(1);
        }

}
```

The code talks by itself, it is very simple. Save, which will build the project, and then run it. The LED will blink as expected.

3.6 Method 6: Booting from SD card

Until this moment, I have shown how to program the FPGA either using Vivado or SDK, using JTAG. There are two other ways of booting the PS and loading the bitstream into the PL. One of them is using QSPI, and the other one is using SD card. In this method I will cover the SD card boot methodology. Generating the files to boot from SD card is very simple, and this knowledge can be applied to boot a linux kernel, or generate SD boot files in SDSoC.

Having the project from method 4, you will need to activate the SD boot in the ZynqMP IP. Also, put the PL clock we changed to 25MHz, back to 100MHz.

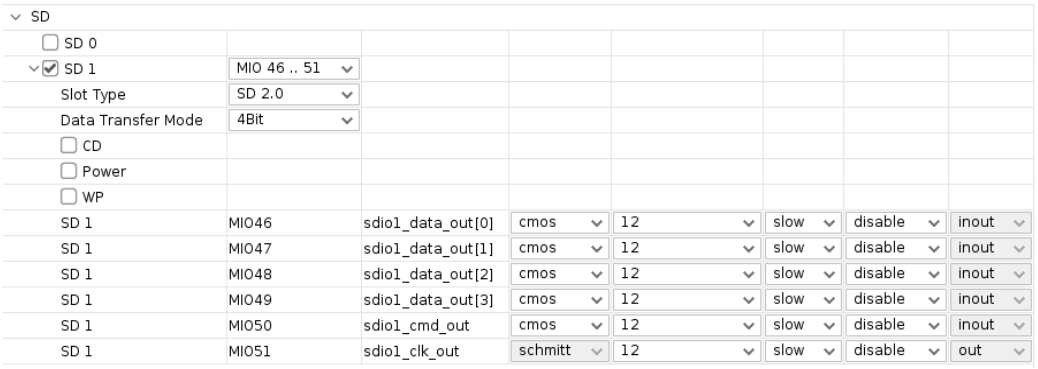


Figure 3.68: Activate SD boot

Generate the bitstream, export hardware, and launch SDK. If you had SDK already open, it might react, as a new *.hdf* file has been generated. If so, accept the changes. Otherwise, you might have to re-create the hardware platform and application project.

Create an FSBL application project. After the FSBL is created and built, select the standalone project that blinks the LED (*LED_Test* in my case), and then go to *Xilinx* → *Create Boot Image*.

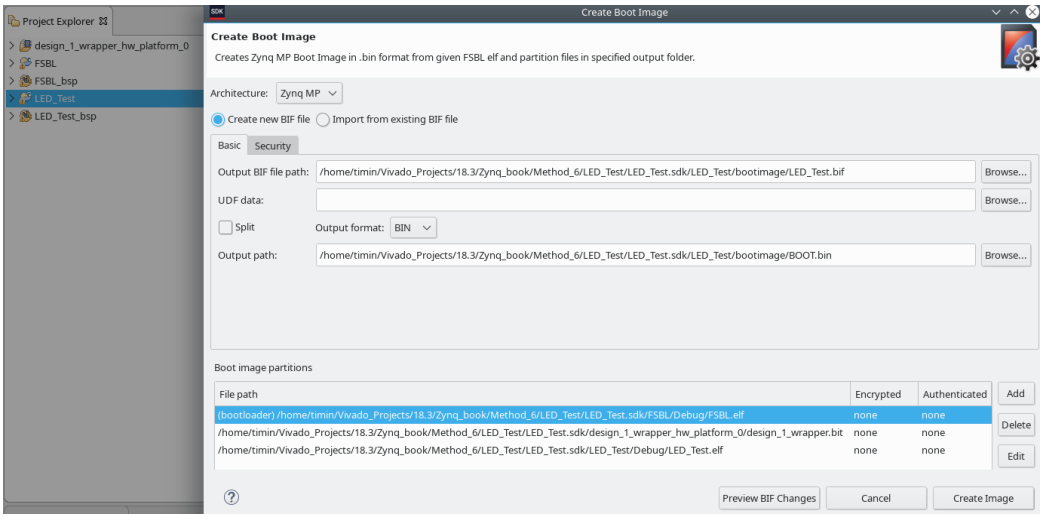


Figure 3.69: Create boot image

As you can observe, Xilinx SDK automatically selects 3 files to merge into one *BOOT.bin* file (you can always do this manually).

- The first step, is the First Stage Boot Loader. This *.elf* file will boot the PS, mapping all the devices declared in our design in memory, and initialising the device.
- The second step, as you might imagine already, is to program the PL with the *.bit* file.
- Finally, the third step is to run the *.elf* file with our application, in this case, the blinking LED.

These 3 steps will occur one after another when the SD card is plugged into our hardware, containing this *BOOT.bin* file.

Click on *Create Image* and your files should be in the path specified in this last window. Copy the *BOOT.bin* file into an SD card, plug it into the carrier board, turn on, and the LED should start blinking after the devices boots.

3.7 Method 7: Using custom board files from Sundance, using a board interface

I want to introduce an important feature, that will buy us a lot of time. Until now, we were selecting a part for our project, and synthesise/implement a design, based on that part's architecture, defining certain constraints.

There is a way to avoid configuring everything manually for every project you create, and that's what the board files are for. In this book, I'm not going to show how to create board files from scratch, but I will show how the information "travels" throughout the tool, to give you a deeper understanding of what the potential of using board files is.

Board files are stored at: *<installationpath>/Xilinx/Vivado/2018.3/data/boards/board_files/* Any board files from different vendors can be allocated there, and will be available to use in Vivado. Also if you create your own!

Board files from Sundance Multiprocessor Technology LTD can be found in our GitHub repository.

https://github.com/SundanceMultiprocessorTechnology/VCS-1/tree/master/Hardware/Xilinx/BoardFiles

Download the required board files, and place them in the path mentioned. Before opening Vivado, let's briefly cover how the board files are structured:

- *board.xml* file.

  This is the file that declares (essentially) components and connections.

  For example, the most important component defined in our board files would be the Zynq device. It's called *part0*, and as you can see, it has a display name, description, etc. Inside *part0*, all its interfaces are declared. In addition to the Zynq device, I added two more components: *ps8_fixedio* and *onboardleds*, the PS and the LEDs on-board respectively.

  Two interfaces have been created as part of the Zynq device (*part0*), and their names are the same as the PS and LEDs components.

  ps8_fixedio and *onboardleds* are the Zynq IP we instantiated in IP Integrator in the last methods and the interface that allows you to access the led we were using until now. (Beware that I modified the indentation of the code so that some things fit in this document).


```
<components>
  <component
  name="part0"
  display_name="EMC2-DP + TE0820-4CG (ZYNQ-UltraScale+)"
  type="fpga" part_name="xczu4cg-sfvc784-1-e"
  pin_map_file="part0_pins.xml"
  vendor="xilinx.com"
  spec_url="www.xilinx.com">
<description>FPGA part on the board</description>

<interfaces>
  <interface
    mode="master"
    name="ps8_fixedio"
    type="xilinx.com:zynq_ultra_ps_e:fixedio_rtl:1.0"
    of_component="ps8_fixedio"
    preset_proc="zynq_ultra_ps_e_preset">
    <preferred_ips>
      <preferred_ip
      vendor="xilinx.com"
      library="ip"
      name="zynq_ultra_ps_e"
      order="0"/>
    </preferred_ips>
  </interface>

  <interface
    mode="master"
    name="onboardleds"
    type="xilinx.com:interface:gpio_rtl:1.0"
    of_component="onboardleds"
    preset_proc="leds_preset">
        <preferred_ips>
          <preferred_ip
          vendor="xilinx.com"
          library="ip"
          name="axi_gpio"
          order="0"/>
        </preferred_ips>
        <port_maps>
          <port_map
          logical_port="TRI_O"
          physical_port="Onboard_LEDs_tri_o"
          dir="out"
          left="1"
          right="0">
                <pin_maps>
                  <pin_map port_index="0" component_pin="LED_1"/>
                  <pin_map port_index="1" component_pin="LED_2"/>
```


```
                </pin_maps>
            </port_map>
          </port_maps>
  </interface>
...
  <component
  name="ps8_fixedio"
  display_name="PS8 fixed IO"
  type="chip"
  sub_type="fixed_io"
  major_group=""/>

  <component
  name="onboardleds"
  display_name="On-board LEDs"
  type="chip"
  sub_type="led"
  major_group="General Purpose Inputs or Outputs">
        <description>LEDs available at the EMC2-DP, next to the Trenz module.
        Same as the top 2 LEDs on the SEIC (extension board).
        They will lit up when high active</description>
  </component>
...
```

Interesting things to mention here? You can see that the preferred IP for the *onboardleds* interface is *axi_gpio*. Also, that the port is defined as an output of 2 signals (for two LEDs available on the board, although we are playing with one only). Also, notice that *onboardleds* points to *leds_preset*. We will see this later.

What about the connections? They are defined as follows:

```
...
<connection name="part0_onboardleds" component1="part0" component2="onboardleds">
  <connection_map
  name="part0_onboardleds_1"
  typical_delay="5"
  c1_st_index="2"
  c1_end_index="3"
  c2_st_index="0"
  c2_end_index="1"/>
</connection>
...
```

This means that the LEDs (index number 0 and 1 of *onboardleds*) have been mapped into index number 2 and 3 of *part0*. Remember this for the next point.

- *part0_pins.xml* file.

This is where the connections defined in *board.xml* are configured

For example, for the *onboardleds* interface declared within *part0* in *board.xml*

```
<part_info part_name="xczu4cg-sfvc784-1-e">
<pins>
  <pin index="0" name ="PHY_LED_1" iostandard="LVCMOS18" loc="A2"/>
  <pin index="1" name ="PHY_LED_2" iostandard="LVCMOS18" loc="A1"/>

  <pin index="2" name ="LED_1" iostandard="LVCMOS18" loc="B5"/>
  <pin index="3" name ="LED_2" iostandard="LVCMOS18" loc="B3"/>

  <pin index="4" name ="LVTTL_0" iostandard="LVCMOS18" loc="T8"/>
  <pin index="5" name ="LVTTL_1" iostandard="LVCMOS18" loc="R8"/>
...
```

Here you can verify, that the constraint we were using until now (LVCMOS18, pin B5), corresponds to the index 2 of *part0*.

- *preset.xml* file.

  This is a very useful file, where preset configurations are defined.

  Do you remember when we used *Run Connection Automation* in Method 5, to apply automatic connections? Similarly, this can be done to apply default configurations. Those configurations are defined in this file. Following the example, this is where the AXI GPIO preset is defined. As I said before, *onboardleds* points to *leds_preset* in *board.xml*.

```
...
<ip_preset preset_proc_name="leds_preset">
  <ip vendor="xilinx.com" library="ip" name="axi_gpio" ip_interface="GPIO">
    <user_parameters>
      <user_parameter name="CONFIG.C_GPIO_WIDTH" value="2"/>
            <user_parameter name="CONFIG.C_ALL_OUTPUTS" value="1"/>
        <user_parameter name="CONFIG.C_ALL_INPUTS" value="0"/>
      </user_parameters>
  </ip>
...
```

  As you can see, the interface is pre-defined with an AXI GPIO block, with 2 signals specified as outputs.

Open Vivado, and create a new project, but this time, select the corresponding board files.

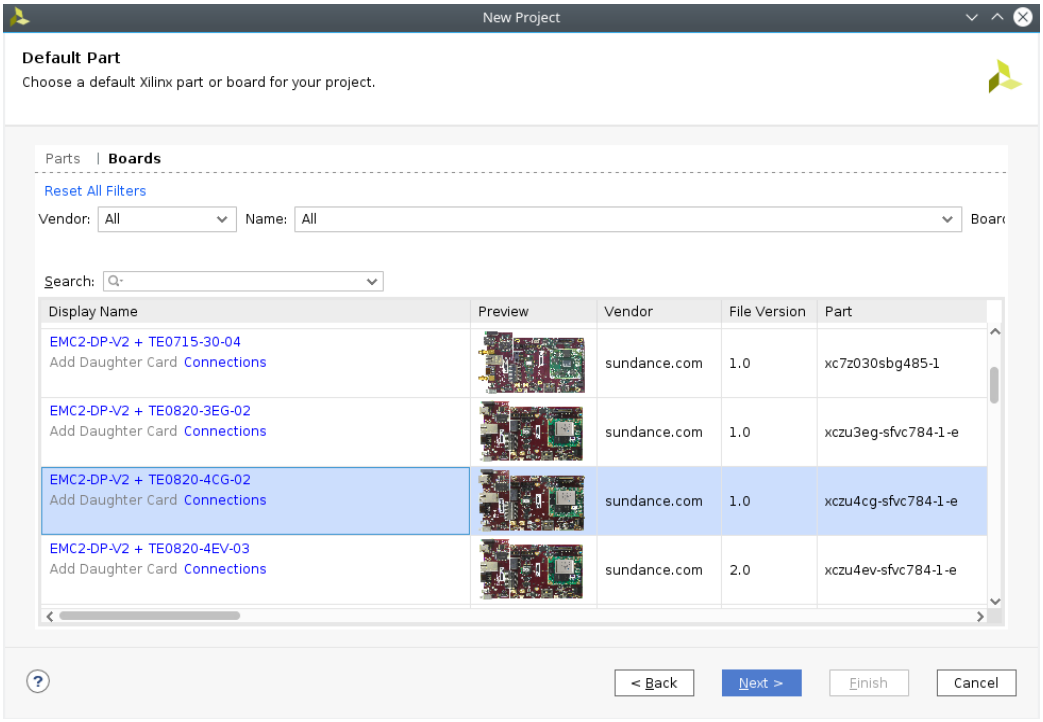


Figure 3.70: Select board files

Create a block design, and add a Zynq Ultrascale+ MPSoC IP. Click on *Run Block Automation*, and *OK*.

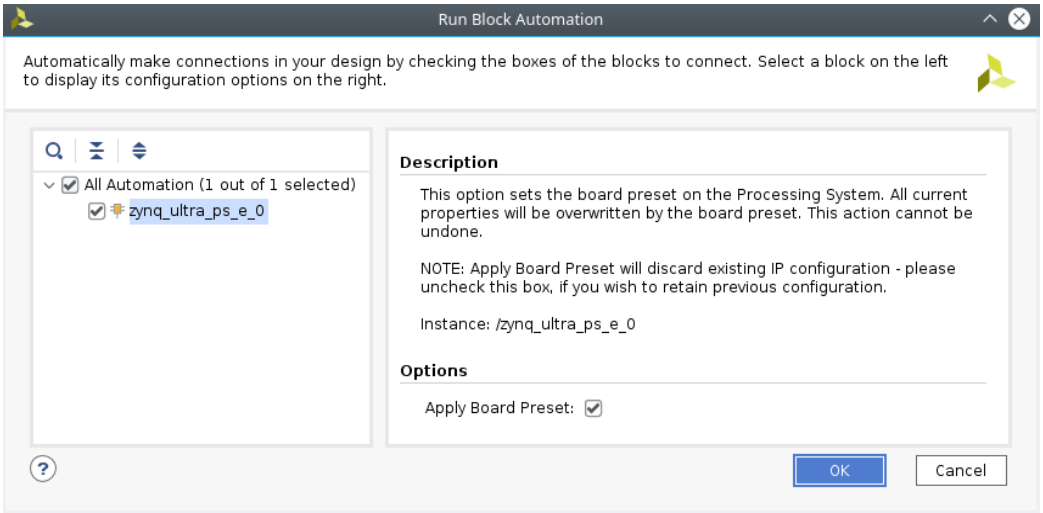


Figure 3.71: Apply preset configuration to the Zynq IP

Add an AXI GPIO block, and click on *Run Connection Automation*. Selecting *GPIO*, choose *onboardleds* as the interface.

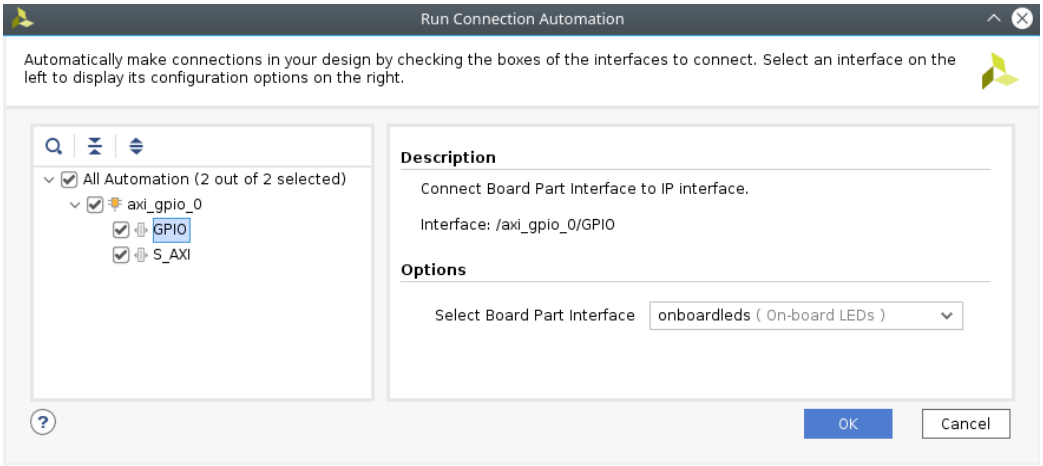


Figure 3.72: Automate the interface connection

Click *OK*, and change the port name from *onboardleds* to just *leds*.

With this, we have just created a project that we can build. No constraints file required. No more steps. Before moving on, I wanted to show another

way of doing this. Going to the tab *Board*, you will see that the *On-board LEDs* interface has been used, and you have many others available. All these are the interfaces defined in the board files.

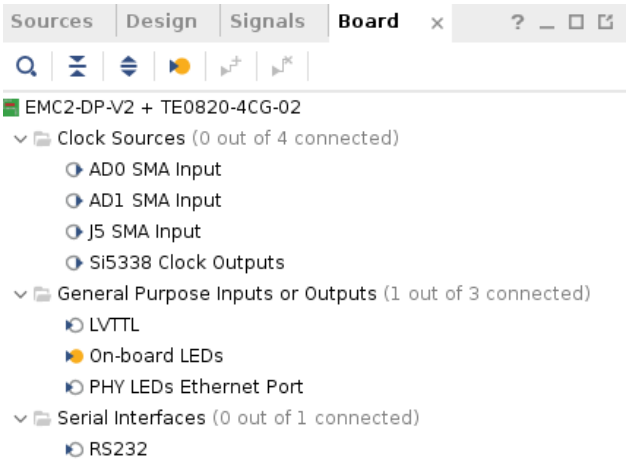


Figure 3.73: Board files' interfaces

Create an HDL wrapper, generate the bitstream, and export the design. In SDK, create an FSBL, Application project, and use the code shown in Method 5. Generate SD boot files as shown in Method 6. This method should make the LED blink as expected.

3.8 Method 8: Automating all the previous methods using scripts

If you followed all the methods, and reached this point, I'm sure that you realised that some steps are always repeating. Creating a project, creating a new block diagram, etc. Also, when you are certain a project works, and you want to store that information somewhere, or you want to share that information with other people, size becomes an issue, as some projects can be very heavy.

I want to share a way of automating the creation of a project, in a simple way. Create a new project, selecting the board files like in Method 7. As soon as you click on *Finish*, before doing anything else, look at the *tcl* console.

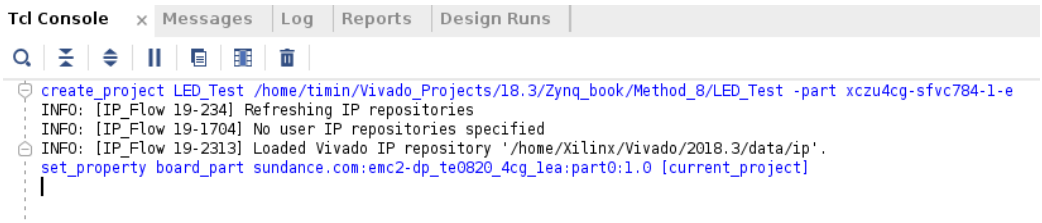


Figure 3.74: Tcl console

Copy those commands (*create_project, set_property*) in an empty text file. Select VHDL as target, create a new block design, etc. One by one, all the movements you do with the tool, copy them into your text file.

```
#Create project to blink a LED
create_project LED_Test /home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test -part xczu4cg-sfvc784-1-e

#Board part selection
set_property board_part sundance.com:emc2-dp_te0820_4cg_1ea:part0:1.0 [current_project]

#Set target to VHDL
set_property target_language VHDL [current_project]

#Create block design in IP Integrator
create_bd_design "design_1"
```

Figure 3.75: Tcl commands in a text file

Continue copying all the steps in the script, and it should look similar to this at the end:


```
#Create project to blink a LED
create_project LED_Test /home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test -part xczu4cg-sfvc784-1-e

#Board part selection
set_property board_part sundance.com:emc2-dp_te0820_4cg_1ea:part0:1.0 [current_project]

#Set target to VHDL
set_property target_language VHDL [current_project]

#Create block design in IP Integrator
create_bd_design "design_1"

#Add Zynq Ultrascale+ MPSoC IP
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:zynq_ultra_ps_e:3.2 zynq_ultra_ps_e_0
endgroup

#Apply preset configuration
apply_bd_automation -rule xilinx.com:bd_rule:zynq_ultra_ps_e -config {apply_board_preset "1" }  [get_bd_cells zynq_ultra_ps_e_0]

#Add AXI GPIO block
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:axi_gpio:2.0 axi_gpio_0
endgroup

#Automate connections
startgroup

apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config { Clk_master {Auto} Clk_slave {Auto} Clk_xbar {Auto}\
Master {/zynq_ultra_ps_e_0/M_AXI_HPM0_LPD} Slave {/axi_gpio_0/S_AXI} intc_ip {New AXI Interconnect} master_apm {0}}\
[get_bd_intf_pins axi_gpio_0/S_AXI]

apply_bd_automation -rule xilinx.com:bd_rule:board -config { Board_Interface {onboardleds ( On-board LEDs ) }\
Manual_Source {Auto}}  [get_bd_intf_pins axi_gpio_0/GPIO]
endgroup

#Regenerate layout
regenerate_bd_layout
regenerate_bd_layout -routing

#Create VHDL wrapper
make_wrapper -files\
[get_files /home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test/LED_Test.srcs/sources_1/bd/design_1/design_1.bd] -top
add_files -norecurse\
/home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test/LED_Test.srcs/sources_1/bd/design_1/hdl/design_1_wrapper.vhd

#Build project and generate bitstream
launch_runs impl_1 -to_step write_bitstream -jobs 2
wait_on_run impl_1

#Export .hdf and .bit
file mkdir /home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test/LED_Test.sdk
file copy -force /home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test/LED_Test.runs/impl_1/design_1_wrapper.sysdef\
/home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test/LED_Test.sdk/design_1_wrapper.hdf
```

Notice that the use of backslash is a continuation line in *tcl*.

You can delete the project, and from the main Vivado window, just paste your script in the Tcl console. The project should be created, built, and exported successfully. You can also go to *Tools* → *Run Tcl Script...* and browse your script to run it.

Now, the next step would be to launch SDK, create the hardware platform, FSBL, application, and generate SD boot files, true? All this can be done with simple scripts.

Assuming that our project is called *LED_Test*, and the *.hdf* file is stored in *LED_Test/LED_Test.sdk*, I want you to create the following files, at the same level as the project's directory:

- *LED_Test.c*

  This file will be the application code, seen in Method 5. Create this file including that code, which makes the LED blink.

- *LED_Test.bif*

  This file will be the one we normally create in SDK when generating SD boot files (figure 3.69, *Output BIF* file). Write this in it:

```
//arch = zynqmp; split = false; format = BIN
the_ROM_image:
{
  [fsbl_config]a53_x64
  [bootloader]/home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test/LED_Test.sdk/FSBL/Debug/FSBL.elf
  [destination_device = pl]/home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test/LED_Test.sdk/HardwarePlatform/design_1_wrapper.bit
  [destination_cpu = a53-0]/home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test/LED_Test.sdk/LED_Test/Debug/LED_Test.elf
}
```

  As you can see, the files specified are the same as the ones we can see in figure 3.69, pointing to the FSBL, bitstream and application.

- *SDK_Script.sh*

  This file will be a script to run from our OS. I use Linux, but if you use Windows, you can create your own script (*.bat* for example), and use it in the same way. What we will do with this script, is the same steps we do in SDK through the GUI. This, is simply faster.

```
# Script to generate HW Platform, FSBL, and App standalone based on .hdf file.

#!/usr/bin/tclsh
# Create workspace
setws /home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test/LED_Test.sdk
# Create Hardware Platform project
createhw -name HardwarePlatform -hwspec /home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test/LED_Test.sdk/design_1_wrapper.hdf
# Create FSBL project
createapp -name FSBL -app {Zynq MP FSBL} -proc psu_cortexa53_0 -hwproject HardwarePlatform -os standalone
# Create Hello World project
createapp -name LED_Test -app {Hello World} -proc psu_cortexa53_0 -hwproject HardwarePlatform -os standalone
# Delete helloworld.c and replace
file delete -force /home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/LED_Test/LED_Test.sdk/LED_Test/src/helloworld.c
cd /home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/
file copy -force ./LED_Test.c ./LED_Test/LED_Test.sdk/LED_Test/src
# Build projects
projects -build
# Generate SD boot files
cd /home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/
exec bootgen -arch zynqmp -image ./LED_Test.bif -w -o BOOT.bin
exit
```

  The command *setws* locates the workspace folder, where all our projects will be created.

  The command *createhw* will create our hardware platform.

  The command *createapp* can be used to create our FSBL and Hello World applications. As you can see, the *helloworld.c* file is deleted and replaced by our *LED_Test.c* file.

  The command *projects -build* builds all the projects in the workspace.

The command *bootgen* will generate the *BOOT.bin* file, based on what we specified in our *LED_Test.bif* file.

Launch SDK in batch mode, in Linux is as follows:

```
source /home/Xilinx/SDK/2018.3/settings64.sh
xsdk -batch
```

Run *SDK_Script.sh* within *xsdk*:

```
cd /home/timin/Vivado_Projects/18.3/Zynq_book/Method_8/
source ./SDK_Script.sh
```

When it finishes, place the *BOOT.bin* file in an SD card, and you will see that the LED blinks as expected.

NOTE: In order to use the scripts without path dependencies, they can be written based on the script location. For *tcl* or *tclsh* scripts, simply add these two lines in the beginning:

```
#Use script's path as project's path
set script_path [file dirname [file normalize [info script]]]
```

After these lines, replace the paths for *$script_path* throughout the script.

For other scripts, you can simply use *$PWD* instead of the path

3.9 Method 9: Creating a kernel out of the design, and using Embedded Linux

I remember the first time I used Linux. I didn't understand the need of having another operating system that wasn't Windows, as I grew up using W95, W98 and WXP.

When it comes to embedded systems, it's very important to know why an operating system is needed. In fact, in this book I don't attempt to explain too much theory about operating systems, but it's important to see when an application requires it, and also, what the advantages and disadvantages are, especially in terms of complexity.

The first question to address is: do we need an operating system for an application as simple as toggling an LED? The immediate answer is no, and there is no other answer, it's simply not needed. But, the essence of this book is to show the simplest and most graphical way to understand an application in a hardware device, and make it work in many different ways, touching different areas in the levels of abstraction in embedded systems.

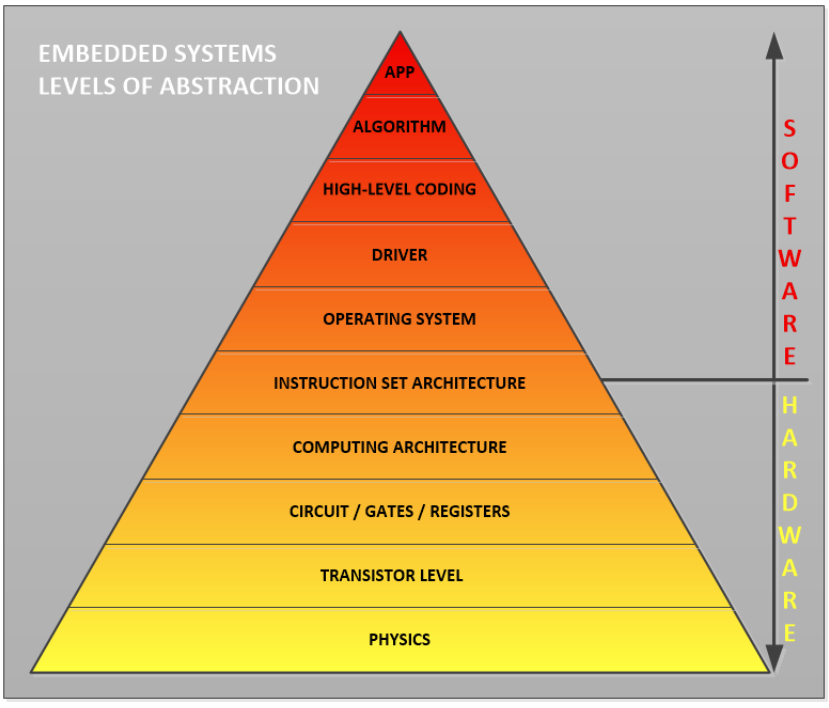


Figure 3.76: Levels of abstraction

Why is it not needed? In other words, why do we need an operating system in an application?

Making an LED blink is a simple application, that requires, in the case of using the PS in the Zynq device, access to a certain memory mapped address for the AXI GPIO controller. The memory address is mapped in DDR, memory that is accessible by the ARM core.

But, what if we want to add several applications to run at the same time, in addition to the LEDs? Baremetal could still be a solution, handling interrupts, and prioritising the applications to our convenience, but, is this efficient? To which extent? The answer to this question is essentially memory. If there is not enough memory available in hardware to manage all the applications, "someone" has to decide which application has access to DDR in a certain moment in time (memory management). This "someone" is the operating system, or OS.

On top of that, an OS is an inner layer between the hardware and thousands of frameworks, drivers, and filesystem support, which is not available sometimes for baremetal applications. Also, the OS allows the use of multithreading, which means that you can exploit the performance of all the ARM cores available in the Zynq device, and not just one.

Now, why Linux? I don't want to go too far in this matter, but Linux:

- Provides lots of device drivers. You don't need to write a USB stack or an I2C library, it's given.
- Provides network and filesystem support. TCP/IP, UDP, etc, also given.
- It's portable. It works in Android devices, microblaze, x86 architectures, etc.
- It's free and open source. It also comes with a large community from which you can get a lot of information, and solutions to known issues.

The tool that Xilinx offers to generate a Linux image based on our hardware design in Vivado is called Petalinux. This book has been written using Petalinux 18.3

Before we dig into Petalinux, I want to leave clear the data flow we've followed until now, and what is new from now on:

- Standalone:

  - Vivado → Hardware design, configuring the PS, and adding an AXI GPIO block in the PL.
  - Export hardware design.
  - SDK → Description in C for the ARM core to command the AXI GPIO controller, to toggle the LED.
  - SD Card booting → Requires an FSBL, bitstream and application files, all merged in a *.bin* file

  NOTE - The AXI GPIO block has an address range assigned, to which we can have direct access in our code. When we read/write from/to that address range, is the real range in hardware.

Figure 3.77: SD Card for standalone applications

- OS-based:
  - Vivado → Design of PS/PL distribution, configuring the PS, and adding an AXI GPIO block.
  - Export hardware design.
  - Petalinux → Generates Linux image based on our design, adding the drivers/modules we want.
  - SD Card booting → Requires two partitions:
    * *BOOT* partition → *BOOT.bin* file (which merges the FSBL, bitstream and u-boot), *image.ub* (Linux image), and the device tree (it can be merged within *image.ub*).
    * *rootfs* partition → File system (group of folders where all our files will be stored, being *"/"* the root.

– Write an application in a Linux compiler, that makes the LED blink, using the Linux GPIO driver (alternatively, you can write your own driver for your own application).

NOTE 1 - Some definitions:

1- u-boot: launches the Linux kernel (*image.ub*)

2- Linux kernel: manages the demands from the different controllers to efficiently use the resources available in hardware.

3- Device tree: file that "tells" the kernel some information regarding the hardware. The idea of the device tree, is that, knowing that there are so many ARM-based boards out there, a kernel can be re-used for different boards, just changing the device tree file.

NOTE 2 - The AXI GPIO block has an address range assigned, to which we do NOT have direct access in our code. When we read/write from/to an address range in our Linux application, is a virtual address range. The translation of physical/virtual address is managed by the MMU. The physical address is given to the kernel by the device tree, or mapped through the FSBL.

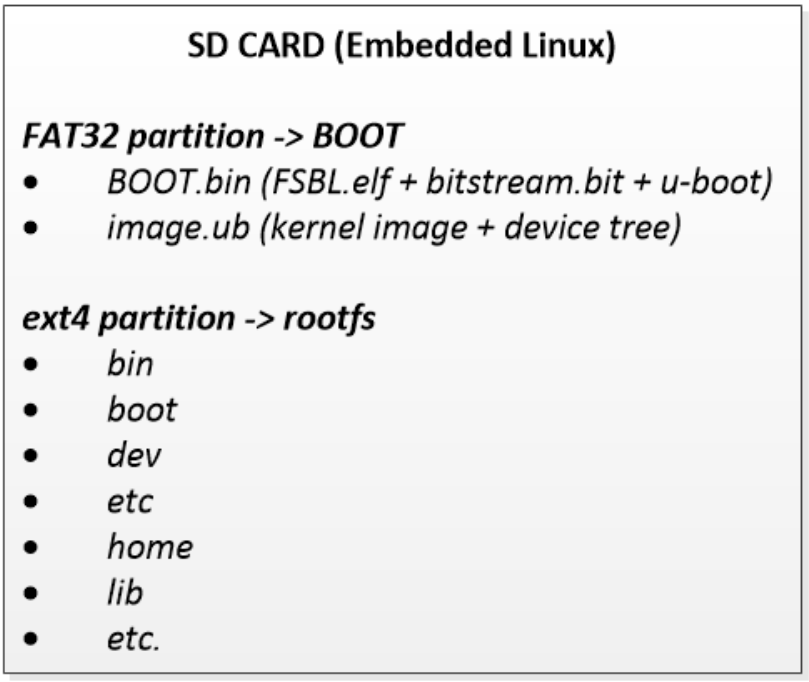


Figure 3.78: SD Card for linux applications

Now it's time to build the kernel, using Petalinux, and generate the files we need to boot from SD card, and make the LED blink from Linux.

First of all, assuming Petalinux is installed, we need to create a Petalinux project:

```
$ petalinux-create --type project --template zynqMP --name LED_Test_kernel
```

Then, we can place the *.hdf* file generated in Method 8 into the project folder. From that directory, we can configure the project settings:

```
$ petalinux-config --get-hw-description -p .
```

A window similar to this one should appear:

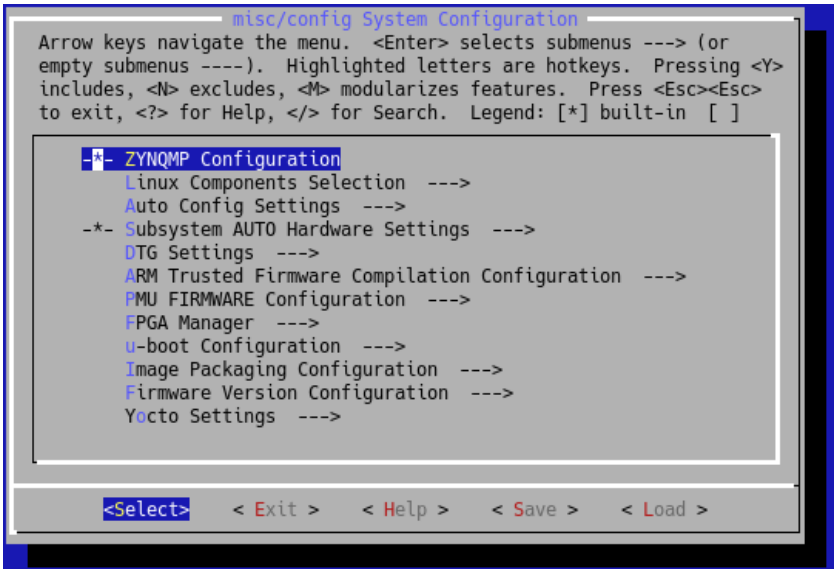


Figure 3.79: Petalinux project configuration

We should change the configuration so that the Linux kernel is launched from SD card:

Subsystem AUTO Hardware Settings → Advanced bootable images storage Settings → boot image settings → image storage media → primary SD

We can also merge the device tree with the kernel image:

Subsystem AUTO Hardware Settings → Advanced bootable images storage Settings → dtb image settings → image storage media → from boot image

Also, the file system will be in the SD card:

Image Packaging Configurations → Root filesystem type → SD card

Save the configuration and exit.

Before building the kernel, we can configure it, and select the modules we want included. To do so:

```
$ petalinux-config -c kernel
```

This time you will see a window like this one:

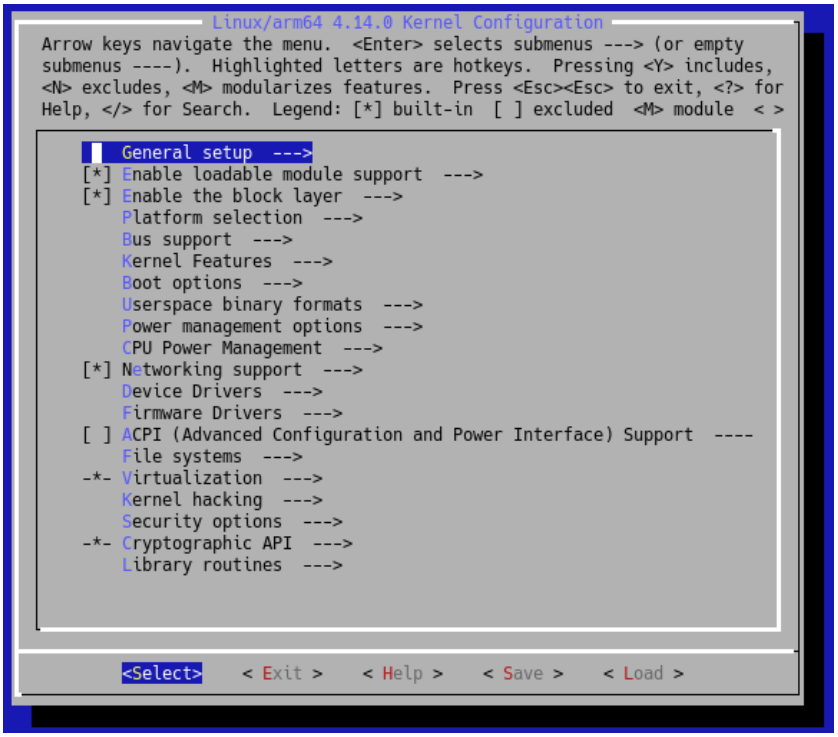


Figure 3.80: Petalinux kernel configuration

Disable initramfs in kernel configuration GUI:

General setup → Initial RAM file system and RAM disk (initramfs/initrd) support

Include userspace I/O platform driver:

Device Drivers → Userspace I/O drivers → <> Userspace I/O platform driver with generic IRQ handling*

Save the configuration and exit. After some work done by Petalinux, the kernel should be successfully configured.

At this point, you have to edit the device tree, to define the peripherals included in the design, in this case, the AXI GPIO block. Edit the file *system-user.dtsi*:

```
$ nano project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi
```

Change the contents of the file, including this instead:

```
/include/ "system-conf.dtsi"
/ {
chosen{
 bootargs = "earlycon clk_ignore_unused console=ttyPS0,115200 earlyprintk root=/dev/mmcblk1p2 rw rootwait uio_pdrv_genirq.of_id=generic-uio";
};
};

&axi_gpio_0 {
  compatible = "generic-uio";
};

&usb0 {
  status = "okay";
};

&dwc3_0 {
    status = "okay";
    dr_mode = "host";
};

&sdhci1 {
 disable-wp;
 no-1-8-v;
};

&smmu {
  status ="okay";
};

&fclk0 {
  status = "okay";
};

/* ETH PHY */
&gem0 {
  status = "disabled";
};

&gem1 {
  status = "disabled";
};

&gem2 {
  status = "disabled";
};

&gem3 {
  status = "okay";
  //reg = <0x0 0xff0e0000 0x0 0x1000>;
  //clock-names = "pclk", "hclk", "tx_clk", "rx_clk";
  //clocks = <0x3 0x1f 0x3 0x34 0x3 0x30 0x3 0x34>;
  phy-mode = "rgmii-id";
  phy-handle = <&ethPhy>;
  ethPhy:phy@1{
    reg = <0x1>;
    ti,rx-internal-delay = <0x8>;
    ti,tx-internal-delay = <0xa>;
    ti,fifo-depth = <0x1>;
    ti,rxctrl-strap-worka;
    marvell,reg-init = <3 16 0xff00 0x12 3 17 0xfff0 0x00>;
  };
};
```

The important thing to know here, is that AXI GPIO is declared as generic-uio. This means that the UIO driver can be used to control the IP. But we will not use this driver. Also, the declaration of the SD controller, USB or Ethernet, for additional features.

At *bootargs*, some parameters are defined as environment variables, to point to the correct source (SD card partitions) in order to boot properly.

You can now build the kernel:

```
$ petalinux-build
```

Be patient, it can take a while, but it should build successfully. From now on, the folder we care about in the project files, is the one called *images*. Inside, we can find the root file system and the kernel image. We just need the *BOOT.bin* file that merges the FSBL, bitstream and u-boot. In order to do so, we have to copy the *FSBL.elf* file (we have seen how to generate it in previous methods) into *./images/linux/*. Then:

```
$ petalinux-package --boot --format BIN --fsbl ./images/linux/FSBL.elf
--fpga ./images/linux/system.bit --u-boot --pmufw ./images/linux/pmufw.elf
```

This command generates the *BOOT.bin* file, placed in *./images/linux/*. Now we have all the required files to boot Linux from an SD card. To prepare the SD card, software tools like *gparted* can be used. As I mentioned earlier, it's necessary to make two partitions. Executing *gparted*, and having an empty SD card connected to our host PC, select the SD card in the tool. Please, make sure it's the SD card, otherwise you can do irreversible damage and lose files from your hard disk.

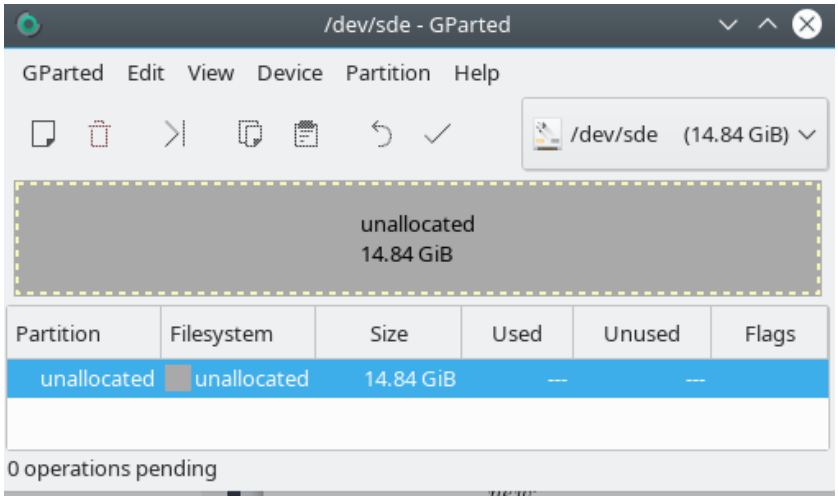


Figure 3.81: Gparted, SD Card

Make sure the SD card is unmounted (you can right click on any partition of the SD card and select *unmount*), and delete the partition of the SD

card so that it displays *unallocated*. Right click the unallocated space and create a new partition (clicking on *new*):

Free Space Proceeding (MiB): 4
New Size (MiB) : 512
Free space following (MiB): (it will automatically select the rest)
Align to: MiB
Create as: Primary Partition
Filesystem : fat32
Label : BOOT

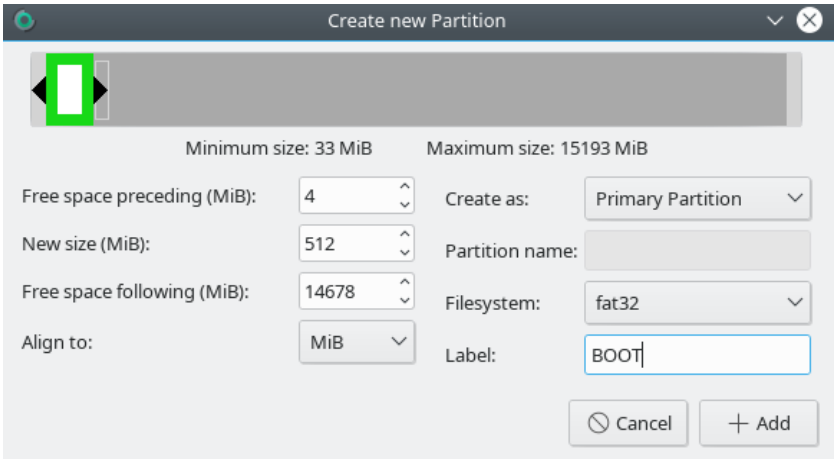


Figure 3.82: Gparted, BOOT partition

Click on "Add". Right click the remaining unallocated space and click on *new*:

Free Space Proceeding (MiB): 0
New size (MiB) : it automatically shows the full size
Free Space Following(MiB): 0
Align to: MiB
Create as: Primary Partition
Filesystem : ext4
Label : rootfs

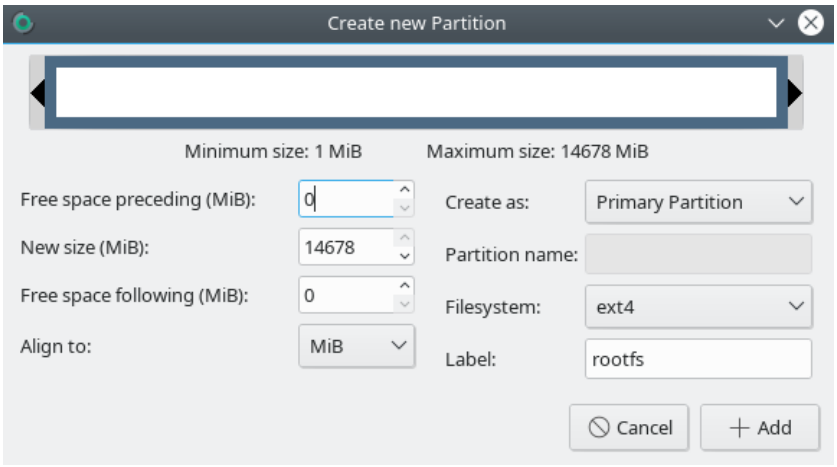


Figure 3.83: Gparted, rootfs partition

Click on *Add*. Apply all operations to create the partitions (*Edit* → *Apply All Operations*). Click on *Apply* when asked if you are sure. Click on *Close* when it finishes. Close *gparted*.

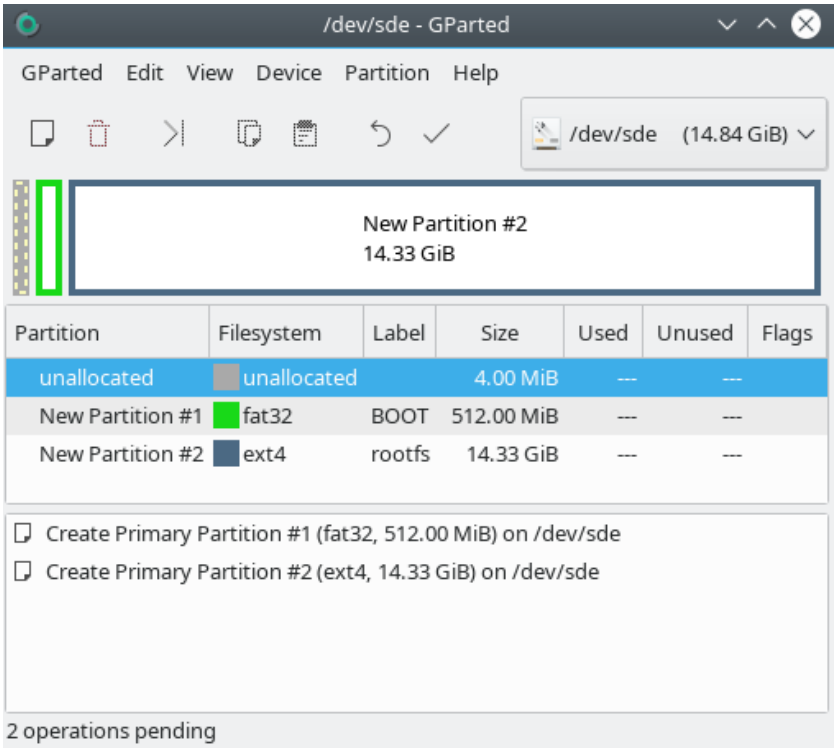


Figure 3.84: Gparted, SD Card set up

Now, assuming that you are in the project's directory, you should copy the boot files into the *BOOT* partition, and the file system into the *rootfs* partition. Give root permissions to the *rootfs* partition, and extract the file system:

```
$ sudo cp ./images/linux/BOOT.BIN /media/BOOT
$ sudo cp ./images/linux/image.ub /media/BOOT
$ sudo cp ./images/linux/rootfs.tar.gz /media/rootfs
$ sudo chown root:root /media/rootfs
$ sudo chmod 755 /media/rootfs
$ cd /media/rootfs
$ sudo tar -xvzf ./rootfs.tar.gz
```

The SD card should have this structure now:

```
timin@timinPC:/media/timin$ tree ./BOOT/ -L 1 && tree ./rootfs/ -L 1
./BOOT/
├── BOOT.BIN
└── image.ub

0 directories, 2 files
./rootfs/
├── bin
├── boot
├── dev
├── etc
├── home
├── lib
├── lost+found
├── media
├── mnt
├── proc
├── rootfs.tar.gz
├── run
├── sbin
├── sys
├── tmp
├── usr
└── var
```

Figure 3.85: Hierarchy of folders inside the SD card

Placing the SD card in the hardware, and turning on, Embedded Linux boots. In order to see and interact with the board, it can be done using a MicroUSB to USB cable, and a terminal. Using *picocom* for example:

```
$ sudo picocom -b 115200 /dev/ttyUSB0 -l
```

Log in using *root* as username and *root* as password. As we already know from the design we've made in Vivado, the AXI GPIO block is mapped at 0x80000000 in memory.

Using a the following command, we are able to turn the LED on:

```
devmem 0x80000000 8 1
```

Where *8* is the number of bits of the data we are writing, and *1* is the value. To turn the LED off:

```
devmem 0x80000000 8 0
```

3.10 Method 10: Using C and Python running Ubuntu

Ubuntu is one of the most used distributions of Linux. For us, it opens doors to install multiple packages that allow us to develop applications in high-level programming languages, having direct access to our peripherals instantiated in the FPGA.

In order to have Ubuntu running in the board, and be able to install *gcc* and *python*, and make the LED blink using C and Python code, we have to use the Ubuntu file system.

Download the ubuntu base file system for *arm64*. It can be found here:

http://cdimage.ubuntu.com/ubuntu-base/releases/bionic/release/ubuntu-base-18.04.2-base-arm64.tar.gz

Now, after deleting all the files from the *rootfs* partition of the SD card, we can extract the downloaded Ubuntu base file system in the emtpy partition:

```
$ sudo tar xvzf ubuntu-base-18.04.2-base-arm64.tar.gz /media/rootfs
```

For us to install packages in file system, we will require internet connection. An easy way to install packages in this file system for ARM, which is in the SD card, but "pretending that" this file system is actually the one in our computer (although our computer is probably not ARM-based!) is doing what is known as cross-compiling.

A tool that allows us to do cross-compiling is *qemu*. Having *qemu* installed, we can use *chroot* to set up the file system of the SD card:

```
$ sudo cp -av /usr/bin/qemu-aarch64c-static /media/rootfs
```

We need to copy certain files from our computer, before doing *chroot*. This file will allow internet connection:

```
$ sudo cp -av /run/systemd/resolve/stub-resolv.conf /media/rootfs/etc/resolv.conf
```

Now, mount *proc*, *sys*, *dev*, *dev/pts* to the new fileystem and enter into *chroot* environment:

```
$ cd /media/rootfs
$ sudo mount -t proc /proc proc/
$ sudo mount -t sysfs /sys sys/
$ sudo mount -o bind /dev dev/
$ sudo mount -o bind /dev/pts dev/pts
$ sudo chroot ./
```

At this point, we are using the root file system of our SD card. Let's do some configurations:

Add an admin user with *sudo* permissions:

```
$ adduser ubuntu
$ addgroup ubuntu adm && addgroup ubuntu sudo && addgroup ubuntu audio && addgroup ubuntu vide
```

Change *root* password:

```
$ passwd root
```

Setup the hostname:

```
$ echo Ubuntu_VCS-1 > /etc/hostname
$ echo 127.0.0.1 localhost > /etc/hosts
$ echo 127.0.1.1 Ubuntu_VCS-1 >> /etc/hosts
```

Fetch the latest package lists from server, and then, upgrade.

```
$ apt-get update
$ apt-get upgrade
```

Install some packages:

```
$ apt-get install dialog perl
$ apt-get install locales
$ apt-get install sudo net-tools ethtool wireless-tools network-manager wpasupplicant nano  wg
```

Install *gcc*, *devmem2*, *python* and *usb utils*:

```
$ apt-get install gcc
$ apt-get install devmem2
$ apt-get install python
$ apt-get install usbutils
```

Exit and unmount:

```
$ exit
$ sudo umount proc
$ sudo umount sys
$ sudo umount dev/pts
$ sudo umount dev
```

The Ubuntu file system is ready. Place the SD card again in the hardware, and the kernel should boot, showing Ubuntu 18.04 LTS for ARM.

We can turn on the LED again, in a similar way to Method 9, using *devmem2*:

```
$ sudo devmem2 0x80000000 b 1
```

Or turn it off:

```
$ sudo devmem2 0x80000000 b 0
```

Let's do a simple application in C. Create a file called *axi_gpio.c*:

```
$ nano axi_gpio.c
```

Copy this code inside:

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
        int i=0;
        int *led;
        printf("Blinking test\n\r");

        int fd = open("/dev/mem", O_RDWR | O_SYNC);
        printf("Device opened\n\r");

        void *virt_addr = mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0x80000000);
        led=(int *)(virt_addr);
        printf("Memory mapped at address %p.\n", virt_addr);

        printf("Device mapped in memory\n\r");

        printf("Blinking LED...");
        for(i=0;i<5;i++)
        {
                printf(".");
                *led=0x1;
                sleep(1);
                *led=0x0;
                sleep(1);
        }

        printf("\n\rTest finished\n\r");

        munmap(virt_addr, 1000);
        close(fd);
        return 0;
}
```

As you can see, the code is simple: -Opening the device called *mem*, which allows us to access physical addresses in memory -Mapping the physical address *0x80000000* into a virtual address, managed by the kernel -Using a variable that points to that virtual address, to blink the LED in a loop.

Save the file, and then compile it using *gcc*:

```
$ gcc axi_gpio.c -o test
```

Run the output file:

```
$ sudo ./test
```

The LED should blink as expected.

Now, let's apply this concept to make the LED blink using Python. Instead of using the same code as before, where there was a *main* function that maps the address and makes the LED blink, we can just use the same code to define a general function, that can be declared in a header file. Modify *axi_gpio.c* with the following code:

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "axi_gpio.h"

int gpio(int value)
{
        int *led;

        int fd = open("/dev/mem", O_RDWR | O_SYNC);

        void *virt_addr = mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd,0x80000000);
        led=(int *)(virt_addr);

        *led=value;

        munmap(virt_addr, 1000);
        close(fd);
        return 0;
}
```

As you can see, we are just declaring a function called *gpio*, with a parameter called *value*. On top, we include a file called *axi_gpio.h*. Create the header file adding this code in it:

```
int gpio(int value);
```

Compile the file *axi_gpio.c* again using *gcc*, and generate a shared object or *.so* file:


```
$ gcc -shared -o axi_gpio.so -fPIC axi_gpio.c
```

With this file, we can import C functions in Python. Run Python:

```
$ sudo python
```

Run the following commands:

```
>>> from ctypes import *
>>> gpio =CDLL("./axi_gpio.so")
```

Finally, you can just turn the LED on with a simple command:

```
>>> gpio.gpio(1)
```

Or turn it off:

```
>>> gpio.gpio(0)
```

CHAPTER 4

What have I learnt?

I hope this book has been useful to you, as much it has been to me. FPGAs, low-powered devices capable of running applications in real time at high frequencies, are a passion for many, me included.

Zynq and Zynq Ultrascale+ architectures from Xilinx opened doors towards an infinity of possibilities, where users take the baton and implement solutions taking the best out of these devices.

So far, in this book we could make the simplest signal available, a LED, blink using the Programmable Logic alone and using the Processing System, both standalone and OS-based. We covered most of the varieties the tools offer us to design and obtain the same result, using VHDL, C and Python languages, and learning how to manually do each step, or even use board files, create Integrated Packages or scripting.

This knowledge is the result of years of experience, and just an injection of morale for what is to come!