525.742 SoC FPGA Design Lab

Lab 1A - Getting Started Tutorial

# Purpose/Goals

The purpose of this tutorial is to familiarize you or provide you a refresher with the design tools, introduce the basic design flow we will be using in this course, and build a basic project with custom software running in the processing system (PS) and custom hardware in the programmable logic (PL).

Table of Contents

*This tutorial is a little long with all the screenshots, so here’s a table of contents to give an overview of the components and help you navigate around as you get into it. Alternatively you can turn on the Navigation Pane to see this outline alongside the document as you read.*

[Purpose/Goals 1](#_Toc1665213977)

[Part I – Vivado & Vitis Installation 2](#_Toc1120096702)

[Part II – Simple PL design 5](#_Toc1671981345)

[Step 1 - Project Creation 5](#_Toc563118901)

[Step 2 – Language Settings 8](#_Toc146074048)

[Step 3 – Design Entry 9](#_Toc911650312)

[Step 4 – Simulation 16](#_Toc1150653688)

[Step 5 – Synthesis / Implementation 19](#_Toc1402522030)

[Step 6 – FPGA Configuration 20](#_Toc1890246097)

[Step 7 – Design Update 22](#_Toc380100110)

[Conclusion 22](#_Toc105192007)

[Part III – Adding PS software 23](#_Toc1023494527)

[Step 1 – Adding a processing system 23](#_Toc1882499458)

[Step 2 – “Instantiating” the processing system in the top level FPGA design 33](#_Toc1439027676)

[Step 3 – Designing Software 36](#_Toc1662793657)

[Step 4 – Debugging 42](#_Toc865659097)

[Step 5 – Writing some custom software 46](#_Toc1792862906)

[Step 6 – Booting from Flash 46](#_Toc1663450177)

[Booting from SD Card 46](#_Toc1546659949)

[Booting from QSPI 47](#_Toc1571304125)

[Part IV – Experimenting with UARTs 48](#_Toc1552396107)

[Option 1 48](#_Toc1630478778)

[Option 2 49](#_Toc1394504025)

# Part I – Vivado & Vitis Installation

Vivado downloads are available from Xilinx at <https://www.xilinx.com/support/download.html>. You may need to create a free account and provide some information for Xilinx to verify export compliance but you will not need a license to use Vivado with the Zybo board / Zynq part in this course.

We want to download the unified installer for version 2021.2. It is important that we all use this version as moving projects between versions can introduce a number of issues. The tools are available for Windows and Linux. The Windows installer includes everything you need for setup. f you want to install for Linux, it will likely work seamlessly if you have previously installed/used Vivado before, but if not there will be a few missing packages to install before Vivado/Vitis will run properly. If you are working on a Mac you will need to set up a virtual machine.

Graphical user interface, text, application

Description automatically generated

Download and run the installer for your operating system. If you see any popups from your firewall asking whether the installer should be able to access the network you need to say yes. So far we’ve just downloaded a “small” package that will help us choose what to download and install. The complete tools installation is very large as we will see shortly.

The first screen will be a welcome message. After you click next you will need to enter login information for the Xilinx account you used/created when downloading the installer.

Graphical user interface, text, application, email

Description automatically generated

On the following page we want to choose to install the complete Vitis development kit with Vivado Design Suite

Graphical user interface, text, application

Description automatically generated

The components you should install to be able to complete our labs are

* “Vitis Unified Software Platform” (Vitis, Vivado, Vivado HLS)
* Vitis Model Composer
* DocNav
* Device support for “Zynq-7000”
* JTAG/”Cable” drivers (per the note make sure your board is powered off and/or disconnected during installation)

Graphical user interface, text, application, email

Description automatically generated

At this point you can leave the remaining options at their defaults and click next/accept through the remainder of the screens. If you have other Xilinx tool versions installed at the same installation location (eg. C:\Xilinx) don’t worry – the tools are grouped in subfolders by version and generally coexist without issue. One thing you *don’t* want to do is try to move the installation location to Program Files. Xilinx tools continue to have issues with spaces and special characters in paths. (This goes for your project directories too…stick to something simple like C:\work\lab1, C:\work\lab2, etc.)

# Part II – Simple PL design

Here we will create an exceptionally simple FPGA design that lights the LEDs on the Zybo in a counting pattern incrementing at 2Hz. Note that like many advanced design tools Vivado provides a number of methods to accomplish many tasks including several of the steps below. We’ll present one method for each step here. You may be familiar with or find others over the course of the semester.

## Step 1 - Project Creation

Start Vivado 2021.2 and click Create Project in the Quick Start panel

Graphical user interface

Description automatically generated with medium confidence

A popup will welcome you to the project creation wizard. Click next and choose a name and location for your Vivado project. I like to create a directory for each lab with two subdirectories, one named vivado and the other named vitis, which will hold my hardware and software designs, respectively.

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

At the next dialog, select the option to create an RTL Project and skip specifying existing sources in the wizard. We will add/create our sources later on for this first project.

Graphical user interface, text, application, email

Description automatically generated

After this we need to tell the tools what device we are targeting. Specifying the board (Zybo Z7-20) rather than just the Zynq part (xc7z020clg400-1) on the Parts tab allows the tools to preconfigure the project with some settings such as DDR memory parameters that are specific to the board.

Click the Boards tab, then enter the text “zybo” in the search box. You should see a list of boards including the Zybo Z7-20. If you don’t, click the Refresh button to get an updated list of board definitions from the git repository where they are hosted[[1]](#footnote-2). In the status column you’ll see an icon to download the board definition. Once you have clicked and downloaded the Next button should be enabled.

Graphical user interface, text, application, email

Description automatically generated

Now you should see a project summary as follows with a Finish button.

Graphical user interface, text, application

Description automatically generated

## Step 2 – Language Settings

Now we should have an empty project and an opportunity to check/fix two important settings that will help us down the road. If we open the menu Tools -> Settings you should see the following in the Project Settings -> General section.

Graphical user interface, application

Description automatically generated

If the target language is not VHDL change it to VHDL now. Additionally, move to the Tool Settings -> Project section and do the same for the default target language for new projects so that you don’t have to worry about this in the future.

Graphical user interface, text, application

Description automatically generated

## Step 3 – Design Entry

Now that the project has been created, it is time to describe our design. This will be done using two HDL files: **toplevel.vhd** which we will create from scratch, and **clkdivider.vhd** which is provided for you at this link: [clkdivider.vhd](https://livejohnshopkins-my.sharepoint.com/:u:/g/personal/dwenstr3_jh_edu/EeNjLx_XjzBHgOwdM2YEj84BKQ-VtJwte0BGSqD_jbcBAw?e=clwn3c). This way you’ll see the steps to add both new and pre-existing code files to your project.

Download clkdivider.vhd to any convenient location, as Vivado will copy it to the project directory shortly, then click “Add Sources” under “Project Manager” in the Flow Navigator in the left hand pane to get started.

Graphical user interface, text, application, email

Description automatically generated

We’ll start with the existing source, clkdivider.vhd

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Make sure that you’ve checked “Copy sources into project” so that this existing file gets copied from wherever you saved it into your project directory. When we’re done with the lab you’re going to zip the whole project directory to turn in and we don’t want broken links to elsewhere on your computer when I try to open your projects.

Before you click finish to add the file(s) to our project, though, let’s also take care of the new source file toplevel.vhd we want to create. Click the “Create File” button and specify the name for your new file.

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Now we have both files and we can click Finish. At this point, a popup dialog about “toplevel” (and any other new sources) will appear, asking us to specify ports. Whether to fill this out now is up to you. If you enter the ports here, you won’t have to type them in the entity description. I prefer to just type/paste in the text editor and I’m going to give you the code to paste, so let’s click through this box and acknowledge any warnings.

Graphical user interface, text, application

Description automatically generated

After you click Finish your project will contain two VHD files, clkdivider (which has something in it) and toplevel (which is essentially empty). Let’s now double click to open and edit toplevel to do what we want.

Graphical user interface, text, application, email

Description automatically generated

Copy and paste the code between the ---labels--- into toplevel.vhd replacing any template text that Vivado auto-generated for us.

-----Toplevel.vhd Code-----

**library** IEEE**;**

**use** IEEE**.**STD\_LOGIC\_1164**.ALL;**

**use** IEEE**.**NUMERIC\_STD**.ALL;**

**entity** toplevel **is**

**Port** **(** reset\_pb **:** **in** STD\_LOGIC**;**

sysclk **:** **in** STD\_LOGIC**;**

led **:** **out** STD\_LOGIC\_VECTOR **(**3 **downto** 0**));**

**end** toplevel**;**

**architecture** Behavioral **of** toplevel **is**

**signal** ctr **:** unsigned**(**3 **downto** 0**);**

**signal** clk**,**rst**,**ena\_2Hz **:** std\_logic**;**

**component** clkdivider **is**

**generic** **(**divideby **:** natural **:=** 2**);**

**port** **(** clk **:** **in** std\_logic**;**

reset **:** **in** std\_logic**;**

pulseout **:** **out** std\_logic**);**

**end** **component;**

**begin**

clk **<=** sysclk**;**

rst **<=** **not** reset\_pb**;**

make2Hz**:** clkdivider **generic** **map** **(**divideby **=>** 62500000**)**

**port** **map** **(**clk **=>** clk**,** reset**=>** rst**,** pulseout **=>** ena\_2hz**);**

-- just a silly counter for demo purposes

**process(**clk**,**rst**)**

**begin**

**if** rst**=**'1' **then**

ctr **<=** **(others=>**'0'**);**

**elsif** **rising\_edge(**clk**)** **then**

**if** ena\_2hz**=**'1' **then**

ctr **<=** ctr **+** 1**;**

**end** **if;**

**end** **if;**

**end** **process;**

led **<=** std\_logic\_vector**(**ctr**);**

**end** Behavioral**;**

-----End Toplevel.vhd Code-----

Once you save the toplevel.vhd file if you’ve done this correctly you should see the Sources view update to show that toplevel contains an instance of clkdivider called make2Hz.

Graphical user interface, text, application, email

Description automatically generated

At this point, our behavioral design is complete, and we can move to Simulation and/or Synthesis.

## Step 4 – Simulation

Though this design is simple enough that we could skip simulation without much risk, we will go through the steps of simulating so that you are again familiarized with the tools

Click on “Add Sources” again. This time specify that you are adding **simulation** sources. (This way the tools won’t try to synthesize the un-synthesizable testbench later).

Click Create File as before…

Graphical user interface

Description automatically generated

Then click through the remaining dialogs as before. Open the newly created file and replace its contents with the simple testbench code that follows. Note that a lot of this is boilerplate and a little online utility for making that boilerplate can be found here: <http://vhdl.lapinoo.net/testbench/tb.php>

----Begin testbench code----

**library** ieee**;**

**use** ieee**.**std\_logic\_1164**.all;**

**entity** tb\_toplevel **is**

**end** tb\_toplevel**;**

**architecture** tb **of** tb\_toplevel **is**

**component** toplevel

**port** **(**reset\_pb **:** **in** std\_logic**;**

sysclk **:** **in** std\_logic**;**

led **:** **out** std\_logic\_vector **(**3 **downto** 0**));**

**end** **component;**

**signal** reset\_pb **:** std\_logic**;**

**signal** clk\_in **:** std\_logic**;**

**signal** led **:** std\_logic\_vector **(**3 **downto** 0**);**

**constant** TbPeriod **:** time **:=** 8 ns**;**

**signal** TbClock **:** std\_logic **:=** '0'**;**

**begin**

dut **:** toplevel

**port** **map** **(**reset\_pb **=>** reset\_pb**,**

sysclk **=>** clk\_in**,**

led **=>** led**);**

TbClock **<=** **not** TbClock **after** TbPeriod**/**2**;**

clk\_in **<=** TbClock**;**

stimuli **:** **process**

**begin**

reset\_pb **<=** '0'**;**

**wait** **for** 232 ns**;**

reset\_pb **<=** '1'**;**

**wait;**

**end** **process;**

**end** tb**;**

----End testbench code----

Again, the sources tree should update to show the expected hierarchy.

Graphical user interface, text, application, email

Description automatically generated

And now that you have a testbench, let’s simulate. First, change the “divideby” generic in your top level VHD instantiation of the clkdivider from 62500000 to 62500. Otherwise, you will have to run the simulation for 500 ms just to see the LEDs move once!

In the Flow Navigator (left hand pane) under Simulation, click “Run Simulation” and select “Behavioral Simulation”. The Xilinx simulator should start and it will have run for some default period of time (1us). The controls for this simulator should be exactly like what you used in 525.642, so this tutorial won’t dwell on the details. Type “run 10 ms” in the command window, or use the GUI to run the simulation for 10ms. You should see the LEDs count from 0-F,0-4 in that time.

Graphical user interface, application

Description automatically generated

## Step 5 – Synthesis / Implementation

Assuming your design worked properly, you can now close the simulator and move on to Synthesis. Remember to change your timebase back by changing the generic back to 62500000. After saving, Click “Run Synthesis” under Synthesis in the Flow Navigator to synthesize your design -- hopefully there will be no errors and you will be prompted for next steps.

Graphical user interface, application

Description automatically generated

After synthesizing, you can see some reports if you like. Otherwise, we need to do a little preparation before we move on to implementation and put the design in our FPGA by setting some pin locations. We certainly don’t want these randomly chosen since external hardware like the LEDs are attached to specific fixed pins on our development board. There are a variety of ways to set these pin assignments, from looking at the schematic to reading the labelling on the board itself. Good documentation is available here and in the linked Reference Manual:

<https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/start>

For this tutorial, however, we will take an easier approach and make use of the fact that Digilent has already created a constraints file for the board with all of the pin assignments defined. You can download the master constraints file from the Files->Class Materials section of the General channel in Teams or the direct link from Digilent here:

<https://github.com/Digilent/digilent-xdc/blob/master/Zybo-Z7-Master.xdc>

Download the constraints file to a temporary location, and again click on “Add Sources” in Vivado. This time you will click on “Add or Create Constraints” and navigate to the downloaded XDC. Edit this file by simply uncommenting the pin assignments that you want to use and changing the name of the port to match your top level design port name. Your XDC will end up looking like this (commented lines removed for clarity).

set\_property -dict { PACKAGE\_PIN K17 IOSTANDARD LVCMOS33 } [get\_ports { sysclk }]; #IO\_L12P\_T1\_MRCC\_35 Sch=sysclk

create\_clock -add -name sys\_clk\_pin -period 8.00 -waveform {0 4} [get\_ports { sysclk }];

set\_property -dict { PACKAGE\_PIN K18 IOSTANDARD LVCMOS33 } [get\_ports { reset\_pb }]; #IO\_L12N\_T1\_MRCC\_35 Sch=btn[0]

set\_property -dict { PACKAGE\_PIN M14 IOSTANDARD LVCMOS33 } [get\_ports { led[0] }]; #IO\_L23P\_T3\_35 Sch=led[0]

set\_property -dict { PACKAGE\_PIN M15 IOSTANDARD LVCMOS33 } [get\_ports { led[1] }]; #IO\_L23N\_T3\_35 Sch=led[1]

set\_property -dict { PACKAGE\_PIN G14 IOSTANDARD LVCMOS33 } [get\_ports { led[2] }]; #IO\_0\_35 Sch=led[2]

set\_property -dict { PACKAGE\_PIN D18 IOSTANDARD LVCMOS33 } [get\_ports { led[3] }]; #IO\_L3N\_T0\_DQS\_AD1N\_35 Sch=led[3]

This tells the synthesizer the proper location for all of your external ports and informs it of the speed of your main clock. Leaving the clock unconstrained would almost certainly be fine for this simple project, but it is always good practice to constrain the clock for static timing analysis. Future projects may not be placed/routed in a way that works at 125MHz if we don’t require it from the tools by setting a constraint. This is something that you don’t want to forget.

## Step 6 – FPGA Configuration

Now that the design is completed, click "Generate Bitstream" in the Left Pane. This should trigger the tools to go through Synthesis again (it is out of date since you changed the constraints) as well as Implementation and Bitstream generation. When completed it will create a file toplevel.bit to be used for configuration of the FPGA and you should see “write\_bitstream Complete” at the top right corner of the Vivado window.

Graphical user interface, text, application

Description automatically generated

The Zybo contains onboard USB – JTAG. So, to load our bitfile to the FPGA (and use all other features which are available over JTAG) we need not carry around a Xilinx platform cable brick. Instead, we will just connect a USB cable from our computer to the **“PROG/UART”** Micro-USB jack on the board**. Make sure that the power switch on the Zybo is in the on position, the power jumper by the power switch is set to receive power over USB as pictured, and the boot mode jumper above the PGOOD and PDONE LEDs on the right side is set to JTAG (rightmost two pins, not as pictured).**

**A picture containing text, circuit, electronics

Description automatically generated**

Programming in Vivado is done through the “Hardware Manager”. In the Flow Navigator under “Program and Debug” choose “Open Hardware Manager” then “Open Target” and select Click to open it, select “Open Target” then “Open New Target” and choose “Auto Connect.”

The hardware manager will be open, connected to the programming circuitry on the Zybo. Here you can see that you have your xc7z020 fpga (the PL in our Zynq), and it is currently “Not Programmed.”

Graphical user interface

Description automatically generated

Note – if your boot mode jumper is not set to JTAG this will likely say “Programmed” as the board will have booted the factory demo image (which is flashing all of the LEDs for you) or a previous student image which could be doing any number of things. You can still program a new bitstream over JTAG but it’s best to fix the jumper and power cycle the board in case the image that booted configured any external devices in a way that you don’t know and aren’t controlling.

To program the FPGA with your new bitfile, click Program Device in the Flow Navigator below the option you used to open the target. The default bitfile chosen for you on the next dialog should be correct, so say “Program.”

Graphical user interface, text, application, email

Description automatically generated

At this point - your design is on the FPGA, but no lights are blinking. Press and hold BTN0, (our “reset\_pb”) and the lights will begin counting. We’ve just realized that we made a mistake in our reset polarity. We thought that the button was active low, but it was active high (should have looked more carefully at the schematic!). Let’s go through the correction of that error to practice the tools one more time.

## Step 7 – Design Update

First, change the rst signal assignment in toplevel.vhd to read

rst <= reset\_pb ; --used to say “not”

After saving toplevel.vhd, the top corner of the window should (eventually) update to indicate that the design has been changed

Graphical user interface, text, application, email, website

Description automatically generated

Click again on Generate Bitstream, and the whole process should run again. After bitstream completion select Program Device again to load your new bitstream (note the filename is the same, it has just been overwritten with your changes).

## Conclusion

At this point, you have successfully built a very small, simple FPGA design and tested it on your board. You have stepped through all of the design phases including simulation and are capable of creating HDL based projects for the FPGA/PL.

# Part III – Adding PS software

Next we’ll take what we’ve built in Part II and extend it with software running on the processing system (PS or “PS7”) in order to understand the design flow for integrating and debugging software as well as how to package a design into a binary that can be loaded from flash without use of JTAG / development tools.

*Note: We will cover the basics you need to prepare for the following labs. A much more in depth treatment is available from Xilinx here:* [*https://www.xilinx.com/support/documentation/sw\_manuals/xilinx2021\_2/ug898-vivado-embedded-design.pdf*](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_2/ug898-vivado-embedded-design.pdf)

Preparation

## Step 1 – Adding a processing system

In Part I, you completed an FPGA design that used a 125MHz external oscillator as a clock, and placed a counting pattern on the board LEDs. This step will extend that design, adding a microprocessor with a serial console, and a GPIO peripheral which will read the slider switches on the board, as well as take control of the LEDs to be lit under software control. To that end, we must first go through the construction and instantiation of that microprocessor block. This is done using a feature called “IP Integrator”. The IP Integrator is a sub-tool that allows the user to graphically connect together pieces of pre-packaged IP into a design that can be treated as a single block by the rest of your design. We are going to use it to build a block which has:

1) A representation of our processor/processing system

3) GPIO (for reading switches and writing LEDs)

4) UART (for printf statements and user interface)

5) A debugger module to allow single stepping and code debug

To start, click on the “Create Block Design” in the left pane, give the block a name, and click OK to create your new block design.

Graphical user interface, text, application

Description automatically generated Graphical user interface, application

Description automatically generated

At this point you should see an empty block design page with a recommendation to click the button and add some IP. Let’s start by adding a PS7. Note that as discussed in lecture this IP is a little different than the others in that it won’t be built in fabric but instead is a description of the physical on-chip processor with which we will be interfacing.

Graphical user interface, application

Description automatically generated

Once you add the IP you should see the symbol for the PS7 as well as a link from Designer Assistance to “Run Block Automation.”

Graphical user interface, text, application

Description automatically generated

Let’s go ahead and do that. Click the link and you should see the following dialog. Unlike when you configured a Microblaze in 642 there isn’t a whole lot to do here. The processor exists physically on the chip so we don’t to customize how it is implemented in hardware. We will get to set some (lots of?) registers that affect its behavior but that comes later. For now we are simply saying that (1) we want to bring out the IO and memory interfaces from the processor to ports on our block design and (2) want to apply board presets (from the Zybo Z7-20 target that we chose way back in Part 2) to those configuration registers we’ll look at in a bit.

Graphical user interface, text, application

Description automatically generated

A picture containing graphical user interface

Description automatically generated

Note that the processor is providing an output clock FCLK\_CLK0, synchronous to its own clock that can be used in the PL. It’s not too important now, but let’s go ahead and configure that clock to run at 125 MHz like the external clock to our PL so that if we want to use this clock in it’s place later we won’t need to change any of our calculations, dividers, etc. Double click the block. Before we make any changes let’s take a quick look around.

A picture containing diagram

Description automatically generated

Here we can see all of the various pieces of the processing system, remembering that they are hard-ip – meaning that these peripherals are actually on the silicon whether we ask for them or not. Green blocks can be clicked on to customize that section. Note – any changes you make here are actually register settings in the PS7. You are not changing the FPGA configuration / bit file. More on this later.

As an example, let’s look at the configuration of the DDR controller by double clicking on the green box that says “DDR2/3/LPDDR2 Controller”. As you can see there are a wide variety of parameters that need to be programmed into the controller before it will successfully be able to use the external DDR (what part? How long are the traces?...etc.) Here is where those parameters would be entered. Fortunately, they have been set properly for us based on the board specification we provided when we ran block automation.

**Table

Description automatically generated**

Next, navigate to the “Clock Configuration” page, expand the “PL Fabric Clocks” section, and change the “Requested Frequency” to 125 (MHz).

Graphical user interface, text

Description automatically generated

When you click OK you’ll see four “Critical Warning” messages about DDR signals. These are safe to ignore and would have appeared later even if we had left the clock setting as it was. The story behind them is in the hardware errata for our board here: <https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual?redirect=1#hardware_errata>

Graphical user interface, text, application, email

Description automatically generated

Now we have a processor but no peripherals to do anything with. Our hard processor core includes a lot of built-in peripherals, but we’re going to hold off on exploring those until a bit later. For now, while it may not be the most efficient use of resources, we are going to explore implementing some peripherals in the PL that we can control using AXI interfaces.

First find the “Axi Uartlite” and add it to the design. Graphical user interface, text

Description automatically generated with medium confidence

Again, right after adding, click on the “Run Connection Automation” to help make some of the connections for you. All of these things can be done by hand, but it is much easier (as you’ll see from the block diagram when you finish) to have the tools set things up in the most common configuration.

Graphical user interface, text, application, email

Description automatically generated

If you carefully read what it is offering to do for you, you’ll be more familiar with the process. It is offering to

1: hook the AXI slave interface on the UART to the AXI Master on the CPU.

2: Make the IP interface of the UART external. In other words, take the TX/RX signals which come out of the UART, and hook them to ports so that they can make it outside this block design and into your top level (where we will hook them to pins!)

Diagram

Description automatically generated

Take a minute to trace through the connections and see what the tools have done for you!

Now, before we forget, we’ll want to set the baud rate of the UART. The AXI\_UARTLITE is a peripheral with fixed baud rate. Double click on the UART in the block diagram to configure it.

Graphical user interface, application

Description automatically generated

We will set its baud rate to 115200, which is commonly supported by most terminal programs / serial ports. *Note, the AXI Clock frequency will be set to Auto. By keeping it as such, it will automatically set the divider rate from your main clock down to the baudrate. Note that it might still say “100MHz” even if you have changed the clock rate to 125 as we did previously. That’s ok - this will get updated shortly when we “validate” our design.*

Now, let’s add the GPIO peripheral to the system as our final change to our block design. Add more IP as before, this time selecting “AXI GPIO**”.** Don’t run connection automation quite yet as the tools know some things about our board design and can help us with some naming and configuration items.

Double click the GPIO peripheral to customize. Use the dropdowns to tell the tools we are going to connect GPIO to the slider (dip) switch inputs and GPIO2 to the led outputs.

Table

Description automatically generated

Now click OK, run the connection automation, and note what was done for you so that in the future you can make manual changes:

1. It added another master port to the AXI interconnect
2. It connected the GPIO peripheral to that new port
3. It made the GPIO signals external so that they could go outside of the block.

Using your mouse, you can drag components around to get a better layout if you like. Additionally, (and this is useful) you can change the names of the peripherals to suit you. Later when we write software, we will be referring to the peripherals by name, so a name like “uartlite” might be more convenient than “axi\_uartlite\_0”. To change the name of a block, right click on the block and select “Block Properties”. For convenience, rename those peripherals to “uartlite”, and “dips\_and\_leds” (our GPIO).

Finally if you click on the “+” next to the GPIO interface ports, you’ll see that for GPIO2, a variety of different signals are coming in/out.

A picture containing icon

Description automatically generated

Conveniently using the board parameters the tools have determined that you only need a 4-bit interface for both GPIO and GPIO2. They have also correctly determined that the switches are inputs. For unknown reasons they have decided that the LEDs are input/output and included signals for a tristate configuration. To keep our lives simple let’s change GPIO2 to be a custom configuration. We’ll have a plenty of opportunity to find out how tristate signals work in a later lab. Double click on the block to customize it.

Graphical user interface, text, application, email

Description automatically generated

Select “Custom” in the board interface dropdown for GPIO2. Then switch to the IP Configuration tab.

Graphical user interface, text, application

Description automatically generated

In the GPIO 2 section, check “All Outputs” and just so that we have some indication our configuration settings are working go ahead and change the default output value to 0xA. This way we’ll see an alternating pattern on the LEDs when we program our design before our software has written anything to the peripheral. Click OK and check that your peripheral symbol now shows a simple 4-bit input.

Icon

Description automatically generated

At this point, we are done with the hardware! Much more can be done (we can rename the external ports, change our PS7 register customizations…etc.), but we will get into that later. For now, pressF6 to validate your design. This will check and make sure you haven’t left any important inputs hanging…etc. After that passes, Save (Ctrl+S) and close (if you like) the block design.

## Step 2 – “Instantiating” the processing system in the top level FPGA design

Step 1 was fairly involved, but when you consider what you have constructed wired together (a 32-bit ARM processor with UART, GPIO, JTAG debugger, and external DDR memory) you can start to appreciate the power of block designs and IP integration! Now all that is left to do is place this newly created block (proc\_system) into our top level design.

The easiest thing to do here is to bring up a template that the tools will create for you that shows how to easily instantiate the block design (it saves lots of typing). There are other ways to auto-generate a wrapper but these add an extra layer of hdl files and complexity of choosing (or not choosing) to let the tools auto-update the wrapper when the block design changes. In our suggested approach, if you change the ports on the block design you just need to bring up the wrapper again, copy/paste the component definition, and update the port map accordingly with connections for your changed ports.

Graphical user interface, application

Description automatically generated

When you see the template, you can see that it is an example of how to instantiate your proc\_system in another design. We’ve got the 53 pin MIO, a clk for the PS, and a power on reset and system reset. All of these are connections from the PS to the outside world. They aren’t accessible in our PL design and unlike other signals from our PL they are there always, whether we like/need it or not, so we will put them in our top level HDL to model that truth. Also on the component are our 4 bit input, 4-bit output, and UART TX/RX pair. Finally, occupying all the space and looking complicated are the DDR pins. Though there are a lot of them, it is important to remember that the situation here is just the same as the other fixed IO. These pins are here, they are part of the PS7 and are hooked to specific pins on the package (you can’t change that). The board designer has hooked those pins to the memory already, so the fact that they are in your top level VHDL is only so that your HDL models what is actually there.

Grab the component declaration and copy that to the declarative section of your top level design.

component proc\_system is

port (

DDR\_cas\_n : inout STD\_LOGIC;

DDR\_cke : inout STD\_LOGIC;

DDR\_ck\_n : inout STD\_LOGIC;

DDR\_ck\_p : inout STD\_LOGIC;

DDR\_cs\_n : inout STD\_LOGIC;

DDR\_reset\_n : inout STD\_LOGIC;

DDR\_odt : inout STD\_LOGIC;

DDR\_ras\_n : inout STD\_LOGIC;

DDR\_we\_n : inout STD\_LOGIC;

DDR\_ba : inout STD\_LOGIC\_VECTOR ( 2 downto 0 );

DDR\_addr : inout STD\_LOGIC\_VECTOR ( 14 downto 0 );

DDR\_dm : inout STD\_LOGIC\_VECTOR ( 3 downto 0 );

DDR\_dq : inout STD\_LOGIC\_VECTOR ( 31 downto 0 );

DDR\_dqs\_n : inout STD\_LOGIC\_VECTOR ( 3 downto 0 );

DDR\_dqs\_p : inout STD\_LOGIC\_VECTOR ( 3 downto 0 );

FIXED\_IO\_mio : inout STD\_LOGIC\_VECTOR ( 53 downto 0 );

FIXED\_IO\_ddr\_vrn : inout STD\_LOGIC;

FIXED\_IO\_ddr\_vrp : inout STD\_LOGIC;

FIXED\_IO\_ps\_srstb : inout STD\_LOGIC;

FIXED\_IO\_ps\_clk : inout STD\_LOGIC;

FIXED\_IO\_ps\_porb : inout STD\_LOGIC;

uart\_rtl\_rxd : in STD\_LOGIC;

uart\_rtl\_txd : out STD\_LOGIC;

leds\_4bits\_tri\_o : out STD\_LOGIC\_VECTOR ( 3 downto 0 );

sws\_4bits\_tri\_i : in STD\_LOGIC\_VECTOR ( 3 downto 0 )

);

end component proc\_system;

Then grab the instantiation and paste it into your toplevel.vhd. We will need to replace all of the signals on the right hand side with our top level signals (as I’ve done here because I’m going to shorten the names for the led, sw, and uart signals) if we don’t use matching names, but this still saves work.

proc\_system\_i: component proc\_system

port map (

DDR\_addr(14 downto 0) => DDR\_addr(14 downto 0),

DDR\_ba(2 downto 0) => DDR\_ba(2 downto 0),

DDR\_cas\_n => DDR\_cas\_n,

DDR\_ck\_n => DDR\_ck\_n,

DDR\_ck\_p => DDR\_ck\_p,

DDR\_cke => DDR\_cke,

DDR\_cs\_n => DDR\_cs\_n,

DDR\_dm(3 downto 0) => DDR\_dm(3 downto 0),

DDR\_dq(31 downto 0) => DDR\_dq(31 downto 0),

DDR\_dqs\_n(3 downto 0) => DDR\_dqs\_n(3 downto 0),

DDR\_dqs\_p(3 downto 0) => DDR\_dqs\_p(3 downto 0),

DDR\_odt => DDR\_odt,

DDR\_ras\_n => DDR\_ras\_n,

DDR\_reset\_n => DDR\_reset\_n,

DDR\_we\_n => DDR\_we\_n,

FIXED\_IO\_ddr\_vrn => FIXED\_IO\_ddr\_vrn,

FIXED\_IO\_ddr\_vrp => FIXED\_IO\_ddr\_vrp,

FIXED\_IO\_mio(53 downto 0) => FIXED\_IO\_mio(53 downto 0),

FIXED\_IO\_ps\_clk => FIXED\_IO\_ps\_clk,

FIXED\_IO\_ps\_porb => FIXED\_IO\_ps\_porb,

FIXED\_IO\_ps\_srstb => FIXED\_IO\_ps\_srstb,

leds\_4bits\_tri\_o(3 downto 0) => led(3 downto 0),

sws\_4bits\_tri\_i(3 downto 0) => sw(3 downto 0),

uart\_rtl\_rxd => uart\_rxd,

uart\_rtl\_txd => uart\_txd

);

Add the new signals to your top level entity definition :

entity toplevel is

Port ( --DOES NOT NEED CONSTRAINTS

DDR\_addr : inout STD\_LOGIC\_VECTOR ( 14 downto 0 );

DDR\_ba : inout STD\_LOGIC\_VECTOR ( 2 downto 0 );

DDR\_cas\_n : inout STD\_LOGIC;

DDR\_ck\_n : inout STD\_LOGIC;

DDR\_ck\_p : inout STD\_LOGIC;

DDR\_cke : inout STD\_LOGIC;

DDR\_cs\_n : inout STD\_LOGIC;

DDR\_dm : inout STD\_LOGIC\_VECTOR ( 3 downto 0 );

DDR\_dq : inout STD\_LOGIC\_VECTOR ( 31 downto 0 );

DDR\_dqs\_n : inout STD\_LOGIC\_VECTOR ( 3 downto 0 );

DDR\_dqs\_p : inout STD\_LOGIC\_VECTOR ( 3 downto 0 );

DDR\_odt : inout STD\_LOGIC;

DDR\_ras\_n : inout STD\_LOGIC;

DDR\_reset\_n : inout STD\_LOGIC;

DDR\_we\_n : inout STD\_LOGIC;

FIXED\_IO\_ddr\_vrn : inout STD\_LOGIC;

FIXED\_IO\_ddr\_vrp : inout STD\_LOGIC;

FIXED\_IO\_mio : inout STD\_LOGIC\_VECTOR ( 53 downto 0 );

FIXED\_IO\_ps\_clk : inout STD\_LOGIC;

FIXED\_IO\_ps\_porb : inout STD\_LOGIC;

FIXED\_IO\_ps\_srstb : inout STD\_LOGIC;

--NEEDS CONSTRAINTS

reset\_pb : in STD\_LOGIC;

sysclk : in STD\_LOGIC;

led : out STD\_LOGIC\_VECTOR ( 3 downto 0 );

sw : in STD\_LOGIC\_VECTOR ( 3 downto 0 );

uart\_rxd : in STD\_LOGIC;

uart\_txd : out STD\_LOGIC);

end toplevel;

For convenience I’ve grouped the signals at the top level into signals that need constraints (actual PL signals that need to get routed to a pin) and signals that don’t (signals that are physically routed from the PS7 but described here anyway to make the hardware description complete).

Note that I’ve changed the names of the UART, LED and switch signals from the template for simplicity (and to match our constraints file). And since we have just hooked LEDs to the block design peripheral we need to comment out our HDL that drives the LEDs from the counter - having them driven by the counter as well would be an error (multi-source).

At this point with all of the copying and pasting your code may be looking pretty messy! If you want you can right click in the editor and select “Reformat Code” to let Vivado fix all your indentation and some other formatting elements. May not make everything 100% consistent but helps a lot with readability in a fraction of the time it would take to fix everything by hand.

Now that we have created new top level signals (SW, UART), we will need to add new pin assignments to the XDC file. Lets just edit the text file to uncomment these lines.

set\_property -dict { PACKAGE\_PIN G15 IOSTANDARD LVCMOS33 } [get\_ports { sw[0] }]; #IO\_L19N\_T3\_VREF\_35 Sch=sw[0]

set\_property -dict { PACKAGE\_PIN P15 IOSTANDARD LVCMOS33 } [get\_ports { sw[1] }]; #IO\_L24P\_T3\_34 Sch=sw[1]

set\_property -dict { PACKAGE\_PIN W13 IOSTANDARD LVCMOS33 } [get\_ports { sw[2] }]; #IO\_L4N\_T0\_34 Sch=sw[2]

set\_property -dict { PACKAGE\_PIN T16 IOSTANDARD LVCMOS33 } [get\_ports { sw[3] }]; #IO\_L9P\_T1\_DQS\_34 Sch=sw[3]

and for the UART, we’ll be connecting those signals to a PMOD-USBUART which will plug into JE, so we will assign pins as below. (Note on the schematic that this is “JE2” and “JE3”, pins 2 and 3 of the JE PMOD connector). Let’s find the appropraite lines and change the signal names….

#set\_property -dict { PACKAGE\_PIN W16 IOSTANDARD LVCMOS33 } [get\_ports { je[1] }]; #IO\_L18N\_T2\_34 Sch=je[2]

#set\_property -dict { PACKAGE\_PIN J15 IOSTANDARD LVCMOS33 } [get\_ports { je[2] }]; #IO\_25\_35 Sch=je[3]

set\_property -dict { PACKAGE\_PIN W16 IOSTANDARD LVCMOS33 } [get\_ports { uart\_txd }]; #IO\_L18N\_T2\_34 Sch=je[2]

set\_property -dict { PACKAGE\_PIN J15 IOSTANDARD LVCMOS33 } [get\_ports { uart\_rxd }]; #IO\_25\_35 Sch=je[3]

Save the constraints and your hardware design is complete. Click Generate Bitstream to run through synthesis, implementation, and bitstream generation. Wait for this to complete, but don’t program the device just yet – our hardware design no longer does anything of interest on it’s own.

## Step 3 – Designing Software

At this point, you have a complete hardware design that is ready to do just about anything that the software would like (provided that you only want to read switches, write LEDs, and communicate via. UART).

Note, however, that we did a bunch of things in Vivado that our software development environment probably should know about: we built peripherals into our programmable logic and attached them to the processor’s AXI interface, assigned memory mappings (tools did it automatically but more on that in later labs), and specified configuration settings that are controlled by registers in the processor. The tools can provide us drivers for the peripherals, automate the programming of the configuration registers, and more if we formulate a good description all that we’ve done to hand off to the software development environment.

To do this, we choose a menu option File -> Export -> Export Hardware. This opens a wizard to “Export Hardware Platform.” From here things start to diverge from what you may be familiar with if you’ve worked with processors in tool versions 2019.1 or earlier in 642/at work/elsewhere. Click Next.

In the second wizard screen you are asked to select the type of output we want to generate. We’ll select “Include bitstream” to package the FPGA configuration with our hardware description. This will make programming easier later on.

In the following dialog you can accept the default settings to create toplevel.xsa at the route of your *Vivado* project directory. This .xsa is really just a zip file with a different extension that contains several files, some of which we’ll be exploring when we get into the software tools.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Click Next, take a look at the summary of what the wizard is about to do, and click Finish.

Once the hardware description (“platform”) has been generated and saved we are ready to switch to our software development environment “Vitis.” We can launch it directly from Vivado with the menu option Tools -> Launch Vitis IDE. If you have used Eclipse or other Eclipse based IDEs portions of what follows will look familiar.

The first dialog that appears will ask you to select a workspace. Xilinx recommends against placing this workspace inside your Vivado project directory as was the default in SDK up until 2019.1 and the default location isn’t a great idea either because a) we want to have separate workspaces for each project/lab and b) we want to be able to easily zip up a folder to submit to instructors. Instead use the Vitis folder you created in Part II / Step 1. In my case that looks like this:

Graphical user interface, text, application, email

Description automatically generated

(remember the Vivado project for the hardware was in my case at C:\work\labs\_2021\_2\lab1a\vivado).

This way when I’m ready to send my project to an instructor I can zip the whole C:\work\labs\_2021\_2\lab1a folder and it will include everything needed with relative paths preserved so that (hopefully) the instructor will be able to load it easily.

When we click Launch, Vitis creates an empty workspace and brings us to the Welcome page to ask how we want to start.

Graphical user interface, application, Word

Description automatically generated

Click “Create Application Project.” This will launch a wizard which will guide us through the necessary steps to get software running, including generating a platform project based on our exported hardware definition. Take a look at the first screen of the wizard. It provides a helpful overview of the components of how Vitis structures the components of a design by processor; a platform that defines the configuration, operating system and hardware surrounding the processor; and an application. When you are ready to proceed click Next.

Vitis supports pulling platform configurations from a number of sources, but we just built our own custom design for our board/chip in Vivado and exported a .xsa file describing it. Switch to the “Create a new platform from hardware (XSA)” tab. There you can choose from some pre-built board descriptions (assuming you didn’t add custom hardware in the PL) or import your own .xsa as we want to do. Click browse and navigate to the .xsa file in your Vivado project folder.

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, application

Description automatically generated

Click OK/Next to import the hardware description. Once that’s complete you will see a dialog to configure the different required projects that we saw in that block diagram on the first screen of the wizard. Choose an application project name. The tools will recommend a system project name based on that. I used lab1a as shown:

Graphical user interface, text, application

Description automatically generated

Leave the default setting to run this project on core 0 (our PS7 is actually a dual core ARM Cortex A9 remember). Click Next, accept the default domain settings on the following page, and click Next again.

On the final screen of the wizard you get to choose from some helpful template applications to get started. Select the “Hello World” template and click Finish.

Now you have a platform project and a system project with your new Hello World application. Navigate to lab1a\_system / lab1a / src / helloworld.c and take a quick look at the software.

Graphical user interface, text, application

Description automatically generated

Not much there but it does look like it should print a little message to let you know it’s working. Next we need to figure out/set where it’s going to print that message – we want it to go to our AXI Uartlite peripheral that we wired out to the PMOD USB UART in our hardware design.

In the toplevel platform project, double click on platform.spr. In the “Main” tab you’ll see that there are two board support packages associated with our processor core in this platform. The first is a first stage bootloader project that we’ll discuss later. The second is for the standalone “operating system” that our application is running under (basically just a collection of driver libraries to make bare-metal development easier). Click on that Board Support Package, then click the “Modify BSP Settings” button in the right-hand panel.

Graphical user interface, text

Description automatically generated

In the dialog that opens select “standalone” in the left-hand panel. You should see on the right-hand side that stdin and stdout (where print and scan functions go to / come from) are set to ps7\_uart\_1. This is not what we want – it’s one of those built in peripherals we said we’d look at later. Open the drop-downs and change both stdin and stdout to point to your uartlite peripheral then click OK.

Table

Description automatically generated

Right click on lab1a\_system and select “Build Project” to build/rebuild your application and any dependencies in the workspace such as the platform project that are out of date.

*If you have experience developing in the old tool Xilinx SDK, note that you have to explicitly build the project whenever you make changes. Saving a file no longer automatically triggers a rebuild in Vitis by default.*

Graphical user interface, text, application, email

Description automatically generated

At this point you have a software binary to run on the processor and a bitfile to configure your programmable logic from step 2.

## Step 4 – Debugging

For this section you need your Zybo board. If you don’t have it yet, you can read ahead, but completion of the remaining sections are not required for Week 1’s assignment.

First, setup the Zybo-Z7 for operation. This will include the PMOD USBUART in the top of JE, and two USB connections from your computer to the board. Using a terminal program of your choice (TeraTerm, MobaXterm, Putty…etc.) open a connection to the PMODUSBUART at 115200 baud 8-N-1. Note- the other USB connection will also have a USB UART on it, so you may want to plug in one at a time in order to figure out which COM port is the PMOD. You will know you have the right one when you can type in your terminal program and lights flash on the PMOD. If you have the wrong one, lights will flash on the Zybo board itself.

***A picture containing text, electronics

Description automatically generated***

Before we can run our application we need to program our PS and PL. There are a few ways to approach this between Vivado and Vitis but we chose to export the bitfile from Vivado along with our hardware description. As a result Vitis is prepared to program both the PS and the PL for us. To see more on this, right click on your lab1a\_system project, go to “Debug As” and choose “Debug Configurations…” Select the “Target Setup” tab. There you can see the tools know where to find the bitfile you exported and are going to program the PL with it as part of the initialization process.

Graphical user interface, text, application

Description automatically generated

Click close to go back to the main screen, then right click the lab1a\_system project again, select Debug As, and this time click Launch Hardware. The tools may warn you the first time you do this that they are going to switch to the debug “perspective” which is a different arrangement of panels better suited for debugging. Up until now you have been in the Design perspective. You can switch perspectives from the window menu or the buttons at the top right of the Vitis window



If everything works correctly the Debug perspective will open, the devices will be programmed, and you software will be sitting paused at a breakpoint at the beginning of the main function. Press the “Resume” button or F8 to run the program or “Step Over” / F6 to run one line of main at a time. When the print lines run you should see the corresponding text in your terminal program.

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

You did it! Now let’s figure out how to use the GPIO peripheral we included as well.

Lets modify this program a tiny bit, and learn how to write the LEDs at the same time. Going back to hello\_world.c - #include two new files :

* xparameters.h this file contains a list of every peripheral in your system, parameters about it, its memory address…etc.
* xgpio\_l.h this file contains low level functions for reading and writing to GPIO

Now, after the printing of hello world, add this line of code, which will write the value 2 to the DATA2 register in the DIPS\_AND\_LEDS peripheral, which we know is our LEDs.

XGpio\_WriteReg(XPAR\_DIPS\_AND\_LEDS\_BASEADDR, XGPIO\_DATA2\_OFFSET,0x02);

Rebuild and run again to confirm that your LEDs are updated as expected.

Lets try one more thing before we write some of our own software.

printf("DIPs read : %x\r\n", XGpio\_ReadReg(XPAR\_DIPS\_AND\_LEDS\_BASEADDR,XGPIO\_DATA\_OFFSET));

This will read the current value of the dip switches and print it out in hex.

## Step 5 – Writing some custom software

Before we move on to other more mechanical things, lets get some practice writing our own software. We know how to print, and read the dip switches, so we now have all the tools to extend this to our own custom program. This program should **start by printing your name and then once printing the current value of the dip switches. Then, the program should run continuously in a loop forever, monitoring the slider switches for changes, and printing the new value when (and only when) they are moved by the operator. (No debouncing or anything fancy needs to be done here, so if a change happens to cause multiple transitions to be printed to the screen, no worries) This program that you have written will be what you demonstrate to show completion of this lab**

## Step 6 – Booting from Flash

We have two options to boot our system (PS and PL) from flash. We can put an image on a micro SD card (included in your kit), or we can program an image into the QSPI flash chip on the board.

### Booting from SD Card

Our software project should already be configured to generate an SD card image by default when we build.

Graphical user interface, text, application, email

Description automatically generated

Much like when we were debugging, because we included the bitfile for our PL when we exported our hardware, the SD card image includes everything needed to load and run both the PS and the PL.

Navigate to your Vitis folder, then into the system project, then into Debug/sd\_card. For the directory structure I’ve been using that winds up being C:\work\labs\_2021\_2\lab1a\vitis\lab1a\_system\Debug\sd\_card

Here you’ll find two files, a short but very helpful readme and the boot file for the SD card. Let’s open the readme.

Text, letter

Description automatically generated

Follow these simple directions (step 2 is referring to the boot mode jumper JP5 that we previously set to JTAG) and when you turn the board on you should see the DONE LED illuminate, your printout(s) on the terminal and the LEDs set just as you did when you ran from the debugger. (Unsurprisingly you can skip copying the readme and just copy the BOOT.BIN file to the SD card if you prefer.)

### Booting from QSPI

To boot from QSPI we need to program the same binary that was in BOOT.BIN along with a bootloader into the QSPI flash chip. Vitis includes and has again pre-configured everything that we need since we exported the bitfile with our hardware description from Vivado. Right click on your system project and choose “Program Flash.”

Graphical user interface, application

Description automatically generated

Everything we need has been pre-populated with one exception. The tools don’t know what type of QSPI flash is on our board.

Graphical user interface, text, application

Description automatically generated

Change the Flash Type to “qspi-x4-dual\_stacked”. Make sure that the boot mode jumper JP5 is still set to the JTAG position as in Part II and then click Program. Messages will scroll on the Vitis console as the device is programmed. When it’s finished you should see something like this.

Graphical user interface, text, application, email

Description automatically generated

You can now power off the board, change the boot mode jumper to the QSPI position (middle two pins), and power back on. You should see the DONE LED illuminate, your printout(s) on the terminal and the LEDs set just as you did when you ran from the debugger.

# Part IV – Experimenting with UARTs

One of the benefits of the hard-core system is that it has so many existing peripherals built in, including GPIO and UARTs. Since the PS7 has (2) UARTs built in that require no logic from the PL we’d prefer to use one of them for our design rather than a UART implemented in PL logic.

## Option 1

The Zybo has a USB UART converter like what’s on the PMOD built-in that can be hooked to the PS7’s UART1 via multiplexed IO. It enumerates over the same cable as the JTAG device programmer. The only inconvenience is that it is it is powered by the board, so every power cycle of the board you lose your terminal. In particular, when booting from flash sometimes the program starts running before the device has re-enumerated and you can miss messages if you don’t put a delay at the beginning of your program. As a result, this may not be the final solution we want, but let’s proceed for now.

No Hardware Needs to Change. UART1 from the PS7 was already enabled to go out to MIO pins 48,49 on the board (this was done when you let the board parameters be applied to your PS7 instance). So, using the UART is as simple as changing software.

Go into your BSP settings, and this time select stdin and stdout to be : ps7\_uart\_1.

Run your program again, and you might notice that different LEDs flash when the program prints. Open the appropriate serial port in teraterm (for help finding it, when you type a key, LEDs on the Zybo will blink instead of on the PMOD). You program should behave exactly as before, and now you need one less USB cable!

## Option 2

As we have discussed/will discuss in class, PS7 peripheral IO can often be sent not only to the MIO pins, but also into the PL itself for use in the FPGA fabric via EMIO. To solve our issue with Option 1 when power cycling, we can take the other built-in UART (UART0/ps7\_uart\_0), bring it’s tx and rx signals into the PL, and route them out to the to the PMOD USBUART like we did with the Uartlite. We will be back to using 2 USB cables and some PL routing and pins, but will avoid consuming PL resources to build a redundant UART peripheral.

To do this we need to edit the block design. Remove the AXI-UARTLITE peripheral. Then double click on the PS7 block to customize. Click on the green box that says UART0, then check the box enabling UART0, and select EMIO in the IO dropdown. This says to enable UART0, and to bring its two signals out to EMIO (note the other choices within the IO dropdown).

When you click OK a new UART port on the PS7 is created and added to the block design symbol. Hook it to the existing port left behind when you deleted the axi\_uartlite. Now you can rebuild your hardware design.

We’ve made changes to our hardware, including deleting the Uartlite, and we need to pass this info and the new bitfile to Vitis. So, when your new bitstream is built, export your hardware again leaving the path the same and overwriting the existing file. Switch over to Vitis, right click on your toplevel platform project, and choose “Update Hardware Specification” – again the path should still be correct so you can just click OK. Lastly modify your BSP/standalone to point to the new UART, ps7\_uart\_0. (Note there wasn’t an option for the Uartlite anymore since we exported a new hardware description after we deleted it.) Then rebuild your software, wait for the bitstream generation to complete, and run again to confirm everything is working.

Part V : Packaging for Submission

For this lab, you will not need to submit the entire project – only a single BOOT.BIN file which can be used by the instructor to see your Hardware/Software in action. When put on an SD card (or in flash) it should print your name to the PMOD USB-UART, and then monitor and report all changes to the slider switches.

1. If Vivado can’t automatically refresh from the repository you can get Digilent board definitions/instructions to install manually here: https://github.com/Digilent/vivado-boards [↑](#footnote-ref-2)