FPGA Acceleration Ramp-UP Guide

By: Iyad Alhasan

Table of Contents

Introduction:

In the ever-going pursuit of "better performance ", researchers have come up with plenty of novel ideas that kept transforming the world we live in into a smarter & more intelligent world and one of the emerging research interests nowadays is acceleration by FPGA. But what does that mean anyway? Let me explain this with an example: Video games. Yes, video games nowadays require powerful GPUs and if you look at it closely, the GPU is a hardware accelerator! Because video games graphics require lots of math functions calculation which would take the CPU forever to perform, GPUs got into picture providing a huge number of specialized circuits to do the math functions required to render the graphics in a fraction of the time required by the CPU. So, hardware acceleration is just a way to run software code with the help of special hardware that implements some time-consuming function to make it run faster, thereby reducing runtime to meet performance criteria.

The guide is meant for beginners who want to learn about FPGA and start using it. It is tailored for explaining the first steps to get into the world of FPGA acceleration. The guide will point to resources that need to be looked at, providing them in a topic-oriented manner with few comments about them.

Understanding what an FPGA is -in a brief-:

A Field-Programmable Gate Array (FPGA) is a reconfigurable integrated circuit that allows users to customize its functionality after manufacturing. Meaning that you can implement any hardware design you have in mind and change it later. That's why FPGAs are widely used in prototyping hardware designs, but discussing FPGA in detail is beyond our scope (Refer to how does an FPGA work for more details). An FPGA board would consist of two main elements, the processing system (PS) & the Programmable Logic (PL), which can talk to each other. The processing system basically does the job of running an operating system (Linux). On the operating system you can create software codes and execute them like a normal computer does. Now moving to Programmable logic, it is a reconfigurable set of building blocks that can be programmed to do any function desired but in hardware, not software and that is what makes FPGAs special.

Getting into the world of FPGA acceleration:

The FPGA board that we will be using to start our journey in FPGA hardware acceleration is the PYNQ-Z2 board. Other boards may have slight variations on how to use them, so the knowledge learnt is beneficial in using lots of other FPGAs. Please read your specific FPGA board's manual after you finish reading this tutorial. I will refer to you the tutorials that I found helpful and need to be read/watched from its original source and try to add any knowledge I have acquired that is not available easily. Let's begin with an overview of what we will be learning:

1. Setting up and connecting our board to the computer.
2. Write and run a code on the processing system (PS): will use python to play around with LEDs on the board.
3. Write a hardware code and use it to configure our programmable logic (PL): will write a Verilog code to play around with LEDs, but this time without interference of the processing system.
4. Accelerate a simple function using both PS & PL.

Setting up the PYNQ-Z2 board, and toggling your first LEDs:

In the following blog post made by Umer Farooq, you will be introduced to the PYNQ-Z2 board, get help setting up the OS image, know the main two ways of connecting to the board and help you use the Jupyter interface (or server as they say). After getting all of that done, the post will guide you to write your first code.

Note: Jupyter interface is an interface that helps you navigate through the files on your board, supports something called notebooks which are the place where you write and execute your python code. This interface also provides access to board using command line terminal (BASH).

Here is the link to blog Post:

PYNQ-Z2 Guide Part I

Connecting board to computer directly and sharing internet connection:

This method aims to provide the best of both worlds, a direct connection with internet access for the PYNQ board. While most of the time the internet connection is not necessary to operate the board, a need to have internet connection to download a library for example might arise with no access to any router. This is where this method of connection might come in handy. I don't recommend using this method for regular use cases and advise skipping to page 9).

Disclaimer: These steps might mess up your internet connection, or ability to access some websites, so whenever you do them, come back and undo these steps once you are done using the board or you might face trouble with your computer connection.

1. Connect your Ethernet cable to your board and your computer.
2. Connect the Micro-USB wire between your board and your computer.
3. Assuming you are using Windows, Open Network and Sharing Center (Figure 1).
4. Click "Change Adapter Settings" Highlighted by red box (Figure 1).

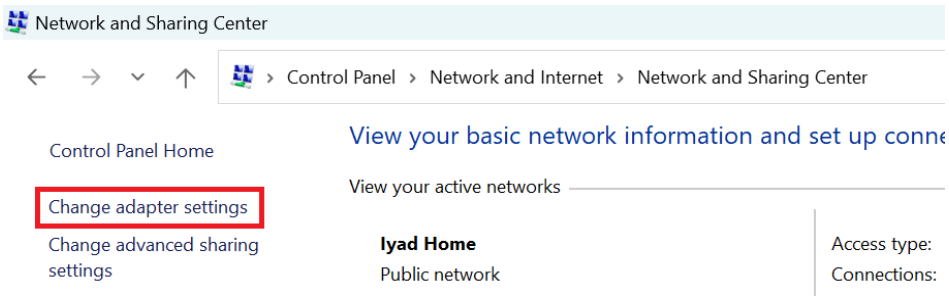

Figure (1)

5. Now the new window that has just opened contains all your network adapters. Highlight your Wi-Fi (**First**) and Ethernet (**Second**) Adapters and do a right-click on the Wi-Fi adapter, then click "bridge connections" (figure 2). Order specified is important.

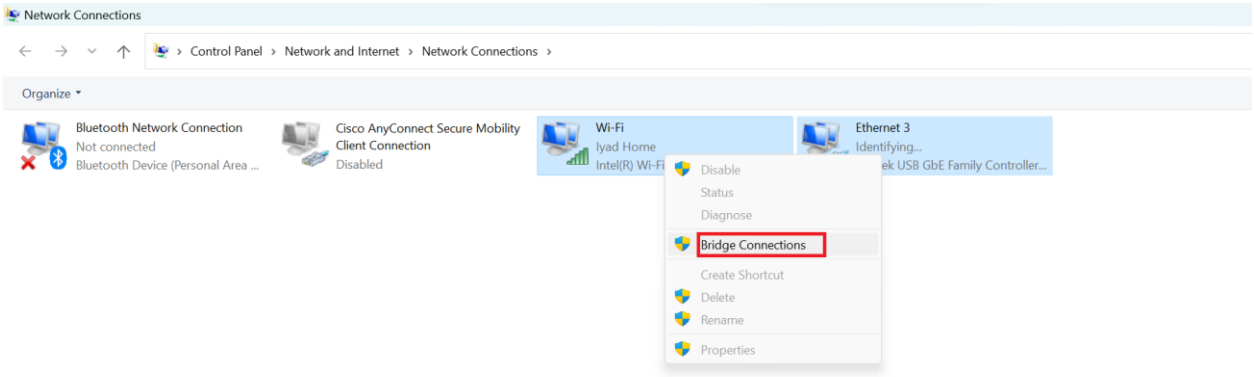

Figure (2)

6. If an error gets shown when bridging the connections (Figure 3), proceed to step 7. If no error is there, skip to step 8.

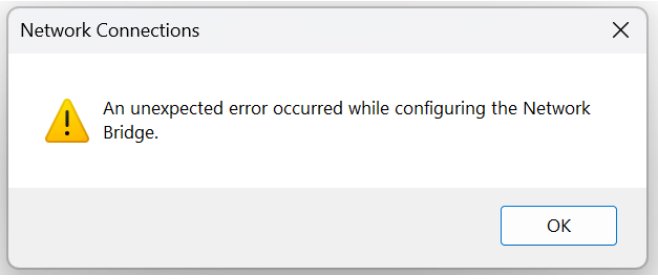

Figure (3)

7. Right click on the new bridge icon and click on “properties”. In the properties, make sure both Wi-Fi & Ethernet adapters are checked (Figure 4). By default, Ethernet would be unchecked (disabled) so check it (enable it). In case you lose your internet connection after this step, remove the bridge connection to restore it and try the previous steps again carefully.

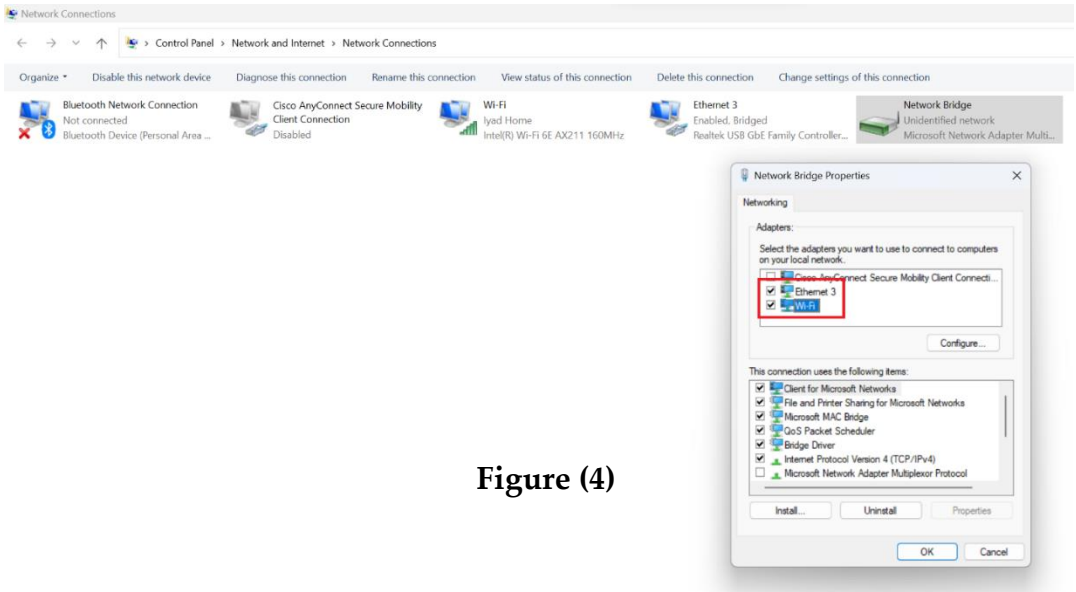

Figure (4)

8. Now click on “Internet Protocol version 4 (TCP/IPv4)” then click “Properties” (green boxes in Figure 5). Make sure the properties tab has the same settings as in Figure 5.

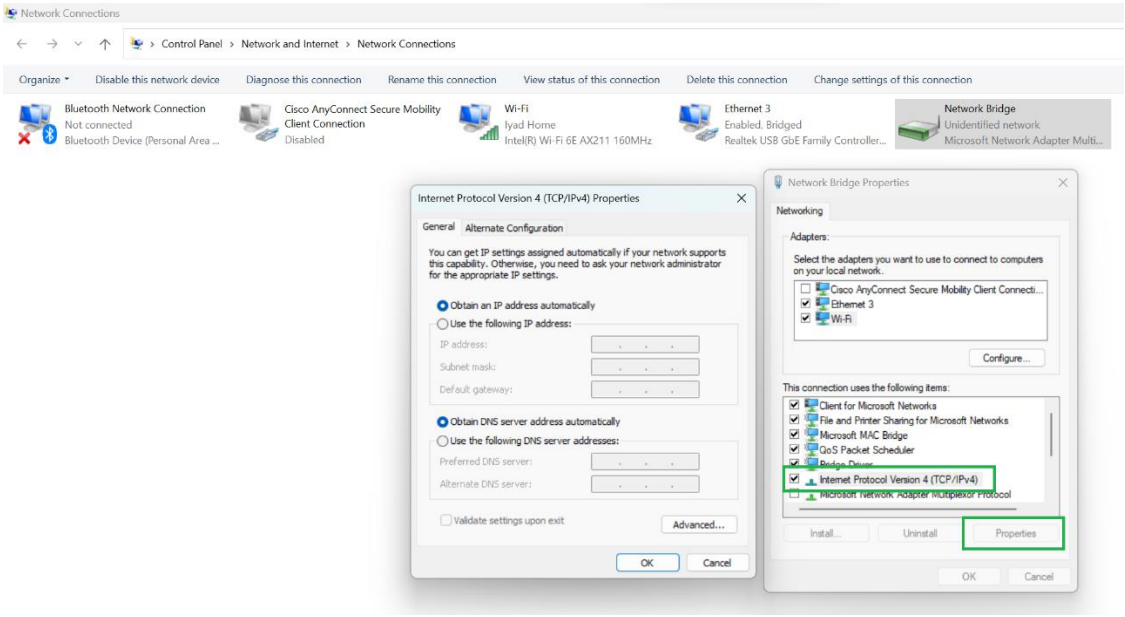

Figure (5)

9. Now go back to adapters tab, right-click on the Wi-Fi adapter and click on “status” and in the small tab that pops, click on the “Details” button then write down the “IPv4 Address” (Green boxes in Figure 6).

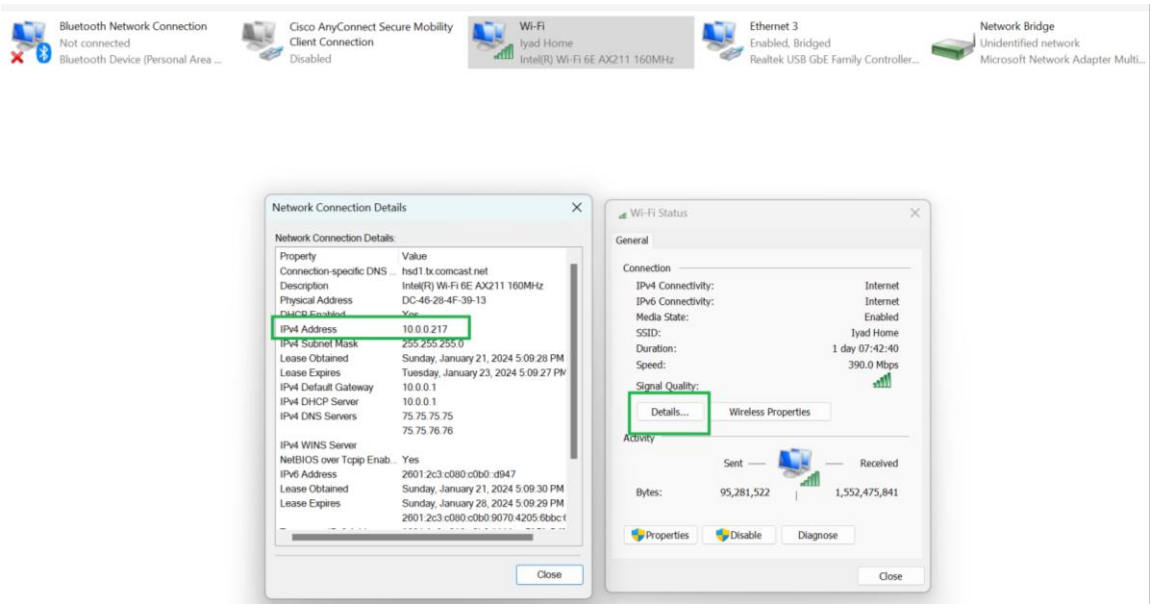

Figure (6)

10. Now we will try to access the Board Serially using Putty software (can be downloaded for free). After you install it and run it, we will select “Serial” connection type and specify Baud Rate as 115200 (Green boxes in Figure 7). We also need to find the COM port number for the board using “Device Manager”.

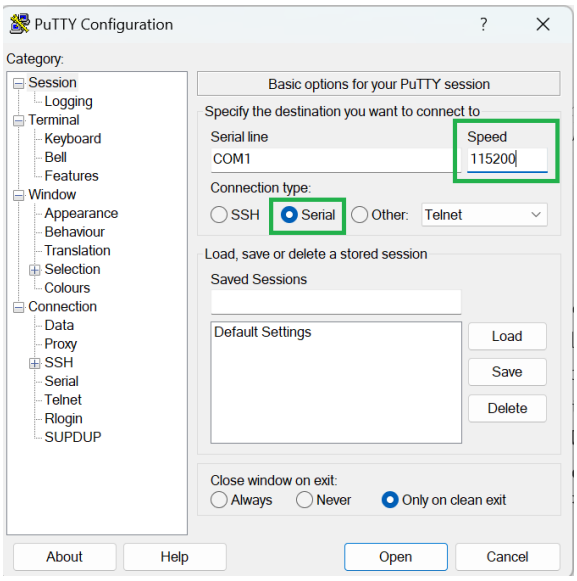

Figure (7)

11. To find the COM port number, we first type “Hardware Manager” in Windows search and once it is open, scroll towards “Ports (COM & LPT)” pull down and expand it (Figure 8). If there are multiple COM ports and you can’t figure which one is for the board, disconnect the boards Micro-USB cable from the computer, in a moment the “Device Manager” will refresh and one port will be gone, now reconnect the Micro-USB cable to the computer and the “Device manager” will refresh and a COM port will be added. Now going back to Putty, insert the COM number in the serial line box (Green Box in Figure 9).

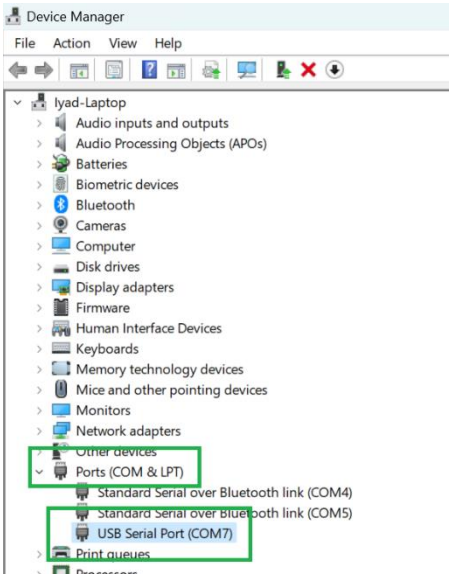

Figure (8)

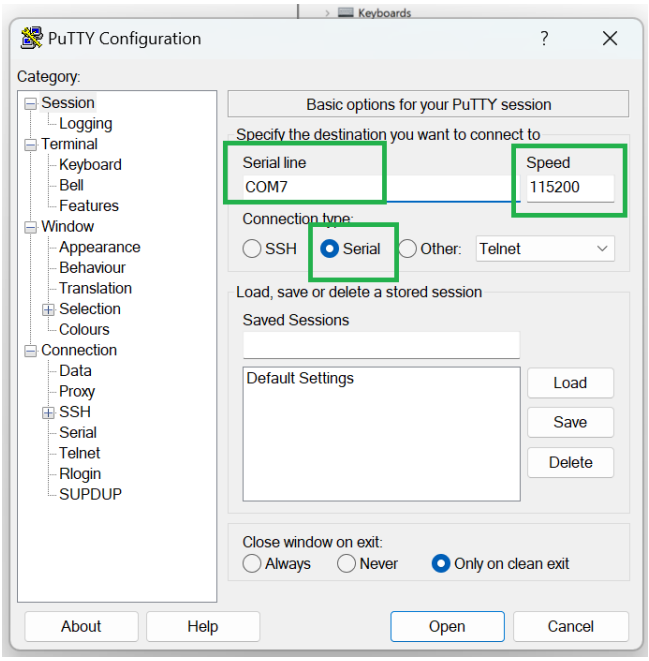

Figure (9)

12. Now click "Open". A black screen should pop up. Now inside this screen we will type the following:

- Click "Enter", When doing so, the cursor will move to a new line and a string (xilinx@pynq:~$) will exist before the cursor.
- Run the following command ***(sudo ifconfig eth0 <IP address you wrote down with the last segment incremented by 1>).*** My IP Address is 10.0.0.217 so when I type the command it should be (***sudo ifconfig eth0 10.0.0.218***). The system will ask you for the password which is "***xilinx***" by default.

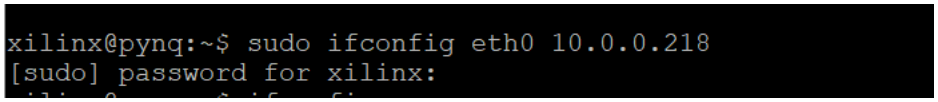

Figure (10)

13. Now to login to jupyter, you need to type in the new IP address you have assigned for the board in your browser (10.0.0.218 in my case).

Blinking your first LEDs**:**

Now that the board is connected to the computer and the Jupyter interface is running, **you can go back to the blog post** and start learning how to write your first code. **Note:** In the end of the article, there is a challenge to use RGB LEDs existing on the board, don't skip it. Use the PYNQ documentation page to find necessary information on how to load the RGB LEDs objects from PYNQ library. Here is the documentation link:

RGB LED Documentation

Done blinking LEDs the PS way? Let's do it the PL way now:

Now that you have created your first code and got exposed to using Jupyter Notebook, we can start learning how to implement a Verilog code to control LEDs without the intervention of Python. In the following blog post (Which is Part II of the previous blog post), Umer Farooq starts by explaining the high-level concept of FPGA acceleration. Then he provides the steps needed to set up Vivado IDE, a tool in which you can write Verilog code, synthesize it then upload it to your board. You can also run simulations in Vivado (if you have created a testbench). After setting up Vivado, he will guide you to write, synthesize & upload your Verilog code with as much clarification as possible. In the end of the post, there is a challenge to use the Ethernet clock to control the LEDs, go for it and if it felt challenging then I would suggest going to Part III of this blog post, understand it then come back to challenge problem in Part II (Since part III helps understanding the challenging part in the problem at the end of part II). Find the links to parts II & III below:

PYNQ-Z2 Guide Part II

PYNQ-Z2 Guide Part III

In Case you are not very familiar with Verilog:

If you find yourself struggling with Verilog HDL, you can find below a very good website tutorial that you can study Verilog from:

Verilog Tutorial

Once you feel more confident, I urge you to do the following project (using a simulator first, then maybe try to port it to FPGA):

Car Parking System using Verilog

Note: Simulation and synthesis results may vary depending on how you code your circuit. In other words, a design that works in simulation may not be synthesizable. Therefore, you need to practice by doing a few projects as a simulation then try uploading to FPGA. One error in synthesis (in Vivado IDE to say the least) that I can think of which does not occur in simulation is when you drive a wire/register in multiple procedural blocks meaning that you can't assign a wire/register in two different ***always*** blocks, even if you made sure they are mutually exclusive.

Accelerating a simple function on the FPGA Board:

Now that we have explored both PS and PL and worked with each one individually for a bit, let's move to the interesting stuff. I need to recommend a specific tutorial which I found to be important. This tutorial goes through the steps to interface PS & PL using GPIO and turn off other interfaces like AXI (if not needed) while also teaching how to use ***concat*** and ***slice*** blocks in Vivado which I find very important to know. GPIO Tutorial Link (from the previous list):

PS GPIO - Part I

PS GPIO - Part II

The Video list these two videos belong to has a few tutorials regarding PYNQ FPGA board. The first 3 videos talk about PYNQ and the environment. I deliberately kept these for now so that when you watch them, you will refresh your information and have a better image overall. Below is the list link:

PYNQ Tutorials - Cathal McCabe

AXI Protocol Based Communication:

Now let's get to real business. When creating a hardware accelerator, one thing that we need is the ability to send large amounts of data to our accelerator then receive the outputs in a an almost automatic way (when I say automatic, I mean that the user does not need to trigger a transfer for every byte of data). In Xilinx boards, the protocol to be used for this purpose is the AXI Protocol. AXI Protocol consists of 3 types:

- Full (Memory Mapped) AXI (AXI-4).
- AXI-Lite.
- AXI-Stream.

Full AXI is usually used to read and write onto memory, while AXI-Lite is mainly used in programming register spaces of IPs such as DMAs in the design (i.e. Processing System configure the DMA settings using AXI-Lite). When developing our accelerator, we will mostly use AXI-Stream Protocol. AXI-Stream is unidirectional (one way communication) standardized way to stream data efficiently between different components, it is less complex than full AXI and has high performance. It has 3 main control signals (It can support more controls for extra functionality, but I will not discuss that since these functionalities are not of interest for now) and a bus signal that carries the data. These signals are:

- TREADY: indicates that the Subordinate[1] (Receiver/Consumer) can accept data when asserted.
- TVALID: indicates that the data on the bus is valid when asserted by the Manager[1] (Sender/Producer).
- TLAST: gets asserted along with the last fragment of data sent to let the subordinate know transmission has ended (gets asserted for one clock cycle, TVALID should de-assert once TVALID de-asserts).
- TDATA: Bus signal that carries data from manager to subordinate.

(1): In some literatures, The Manager and Subordinate are referred to as Master and Slave respectively.

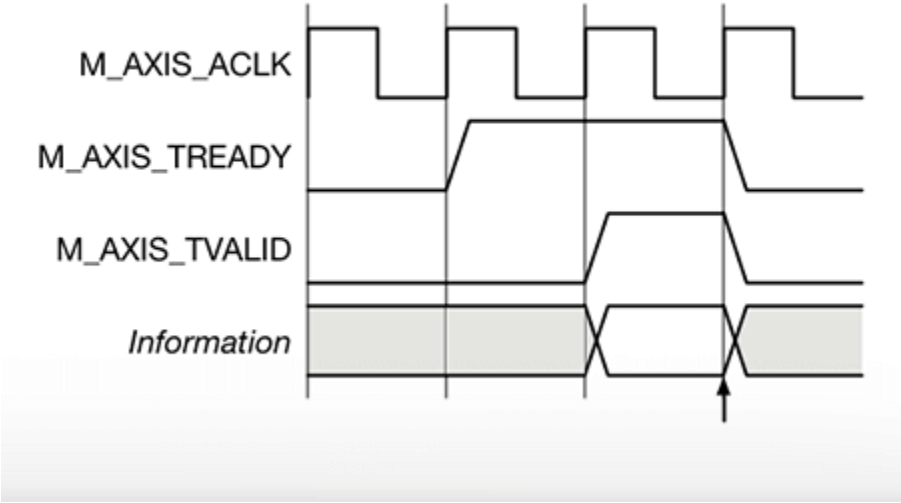

Figure (12): AXI-Stream working principle (TLAST behavior not included)

For more information on AXI Protocol (Full), This link is a good start:

AXI Basics - Xilinx

I made a list of videos where I design an adder that uses AXI-Stream protocol to communicate with PS. In case you find yourself unable to understand any part of the videos, I've got you covered! you will find below in Additional resources section some professional videos. Link to my video list:

FPGA Acceleration Using PynQ

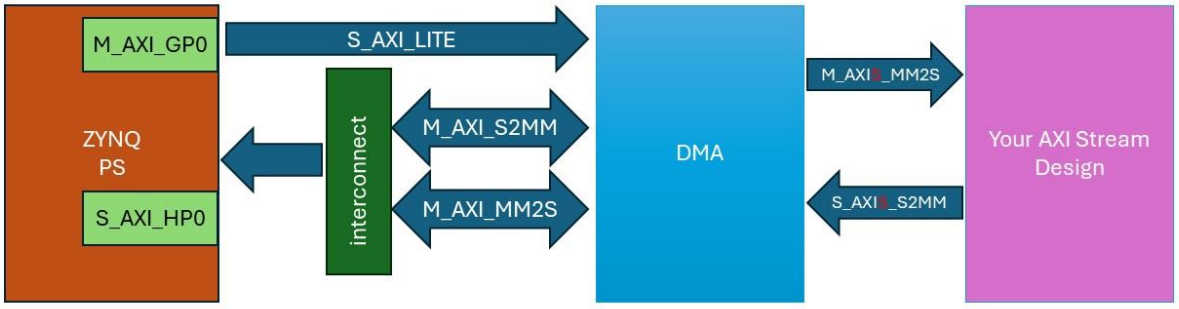

Figure (13): High level View of Hardware Accelerator Architecture.

Must remember:

I assume that you have finished watching the playlist by now, so here is a recap of most important things you must not forget as a beginner:

- Always check which module is set as top in Vivado.
- After performing any change, an "updating" message and icon will be shown on top of the sources tab, wait till it is done before performing your next step.
- When using ILA to debug your signals, load the overlay in python first, then arm your trigger and finally execute the rest of the code. If you arm the trigger before loading the overlay, your trigger will disarm without capturing any data in the ILA.
- Always save and run "validate design" before attempting to generate bitstream or any other flow (simulation, Synthesis & Implementation)

Very Important Point:

TLAST signal associated with your design output is very important. DMA only assumes transaction is done when ***TLAST*** is asserted, and if not, even if your transaction is done and ***TVALID*** has de-asserted, DMA will assume transaction is still happening and hang. Another important signal is ***TREADY***. This signal is driven by the receive end of DMA to your design output. ***TREADY*** states when asserted that the receiver is ready to receive and vice versa. Your design needs to include logic to **halt transmission once DMA de-asserts its *TREADY*,** otherwise the data will be lost. One result to that is a scenario where lost data contain the last fragment of the output (***TLAST*** signal is asserted), hence your DMA will keep hanging. One way to solve this is to introduce a FIFO between your design and DMA with enough depth so it can hold the data generated by your design while DMA is not ready (AXI-4 DATA Stream FIFO IP in Vivado also handles ***TREADY*** from DMA).

Additional Resources:

The following video is a tutorial on how to create a hardware adder circuit and use it as a function in Python code. The tutorial is based on a tutorial in PYNQ documentation. It should be noted that in these two tutorials, the hardware function is created from a C code with special settings (Pragmas). This operation is called High Level Synthesis (HLS) and it converts C code to HDL code, while the pragmas serve as the settings to be used when doing such conversion. HLS is not our target since our goal is to develop hardware circuits that are optimized with careful study. But there is no harm in learning it to develop experience.

Important Note: At some step along the tutorial, you will be copying files created by Vivado to your FPGA board. When copying the ".bit" file to your board, you need to copy a file with extension ".hwh" with it as well. The video doesn't do this step but if you don't do it you will face "object not found" error. The ".hwh" file is located in your Vivado project directory in one of the subfolders. To find it, search the project directory for ".hwh" (Figure 11).

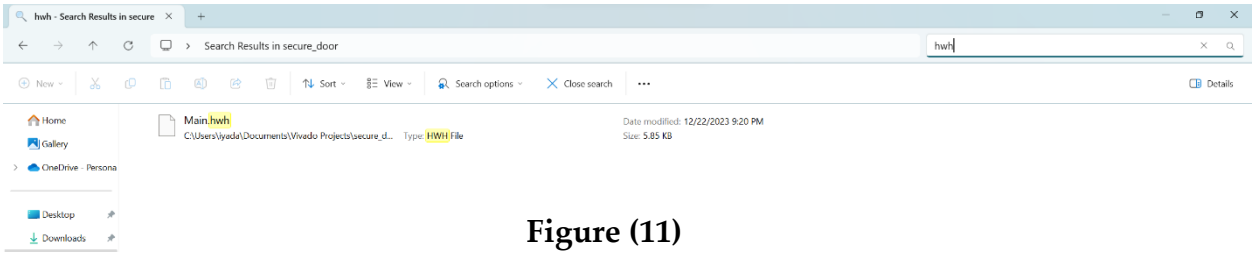

Figure (11)

Following are links for both tutorials:

Creating a Custom PYNQ Overlay - FPGA Developer

Overlay Tutorial - PYNQ Documentation

After you are done with the previous set of tutorials, I urge you to do the next one which is to accelerate a FIR filter. In this tutorial you won't be doing any hardware coding since there is a FIR filter IP available to use for free. The tutorial serves as extra training to get used to the Vivado IDE and Jupyter Notebook. It also guides you to implement the FIR both in software & hardware then compares the performance. This is where things become interesting you start seeing the beauty of FPGA Acceleration.

Note: Sadly, Although the author mentions that the notebook is available for download, I found that the download link in his website is broken so you need to pay a close look for the code when shown in video.

Below is the link to the tutorial:

Accelerating a function with PYNQ - FPGA Developer

Finally, The most time consuming but **beneficial** resource! This YouTube list is made for an advanced embedded course, what is new here is the fact that the professor uses **SDK** software to develop PS software code that drives and communicates with the PL, He accelerates a FFT module, but you can follow him for any design you made, he also teaches how to use **ILA to debug your signal**. But why live the trouble of using SDK instead of PYNQ python environment? There are two main reasons. First, at some point you will need to use a different board that may not have access to similar python environment like the PYNQ environment. Second, PYNQ environment Does not support all functions you may need to use such as scatter-gather DMAs; thus, you will need to switch to C code development on SDK tool. Videos that you will need to refine the specific set of skills we are trying to develop in this tutorial are 1-24, rest of the videos are great for becoming more advanced.

Here is the link to the Course YouTube list:

IIIT Delhi ECE573: Advanced Embedded Logic Design Lab

Happy Learning