

Vivado flow to create an AXI4-Lite IP and using it in a project

Author: Shubhayu Das for VL505 at IIITB

Date: 4th November 2022

This document is going to guide you through the steps involved in designing your own IP. This IP is going to utilize the AXI4-Lite interface available in Xilinx's Zynq 7000 series FPGAs (such as the Zedboard, Pynq, and Zybo boards). The AXI4-Lite interface is utilized to perform PS-PL communication.

PS: The "Processing System", present inside the FPGA chip. It consists of one or more ARM cores.

PL: The "Programmable Logic", which is the normal FPGA fabric that you have been dealing with so far in the course.

We have the program the PS and the PL separately. We will use Vivado to generate a bitstream, which will be used to configure the PL. We will then use Xilinx SDK/Vitis (depending on your version of Vivado) to program the PS side (this will be done in C).

This flow has been tested to work in Vivado versions 2018.3 to 2021.2.

In this tutorial, we are going to focus on using the *Zedboard*. This only matters in **Step 1**, after which all the other steps are the same for the Pynq board.

Step 0

As we are going to use a new board in Vivado, we need to add some configurations into Vivado, so that it recognizes the Zynq7000 boards.

First, ensure that you had added support for Zynq 7000 series while installing Vivado. If you hadn't done this step, please go ahead and install support for Zynq 7000 SoCs.

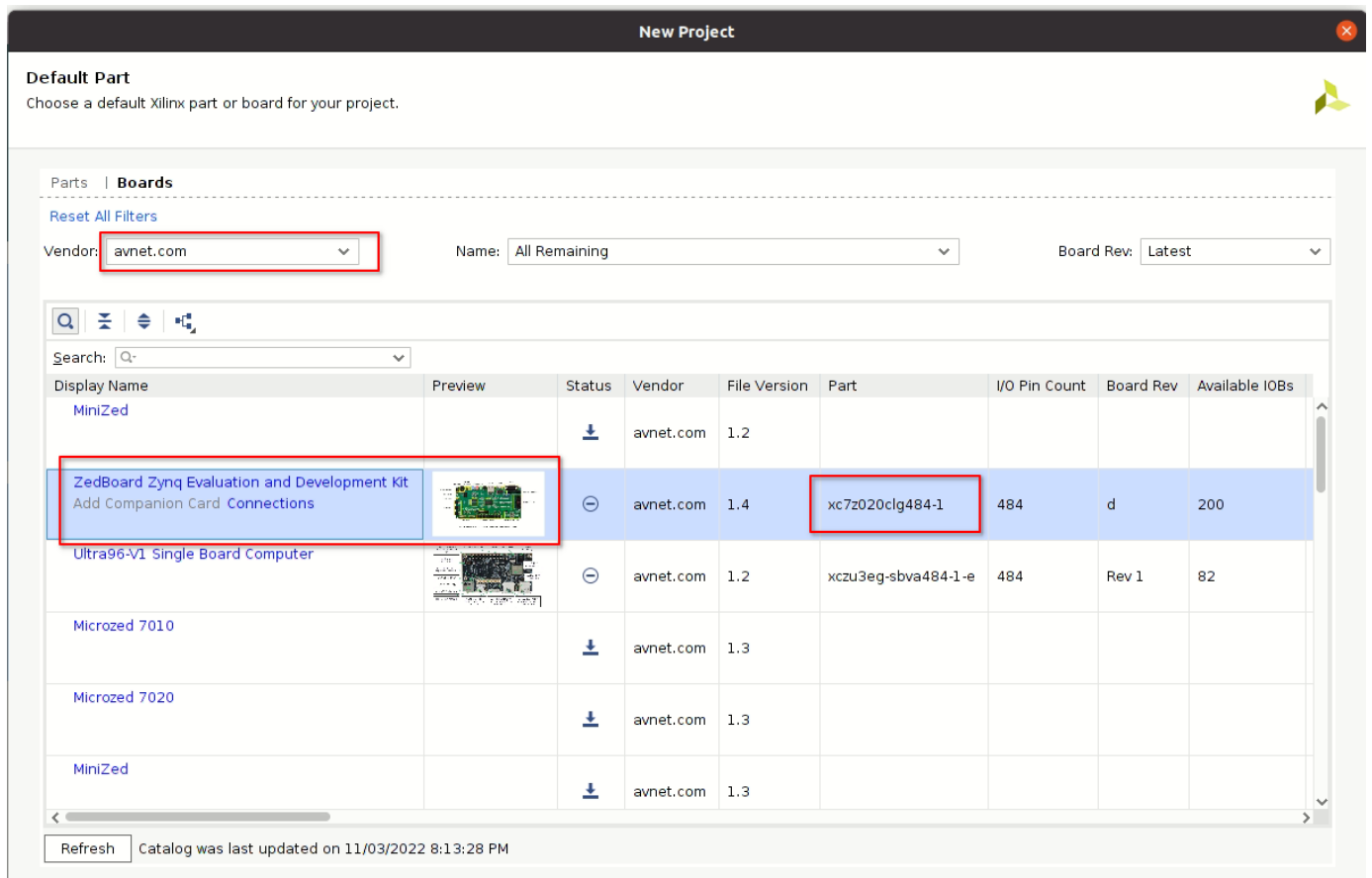
Steps:

1. Open your version of Vivado. If you are in Linux, open Vivado using sudo: `sudo vivado`.
2. In the top bar, click on Help --> Add Design Tools or Devices. 3. Log into your account, and check if Zynq 7000 series have been installed.

Once you are sure that you have support for Zynq 7000 FPGAs, go to [Avnet's board packages GitHub repository](#). **Follow the steps mentioned in the README**. Make sure that you copy the folders with the FPGA boards's name inside the `board_files` folder. [Additional reference: [UG895 - Board Files](#)]

Step 1 - creating a project

Once you've completed the previous step, **re-open Vivado**, and start creating a new project. Add a new design source called `mac.v`. You don't need to add any constraints right now. Next choose the **Zedboard** as the part:



Step 2 - coding your IP

Now that you've created a project, we will write the Verilog code for the IP. In this IP, we are going to perform a simple multiply-add (MAC) operation. Go ahead and add the following code to the Verilog file:

```
module mac(  
    input [15:0] i_a,  
             i_b,  
    input [31:0] i_c,  
  
    output [31:0] o_result  
);  
  
    assign o_results = i_a * i_b + i_c;  
  
endmodule
```

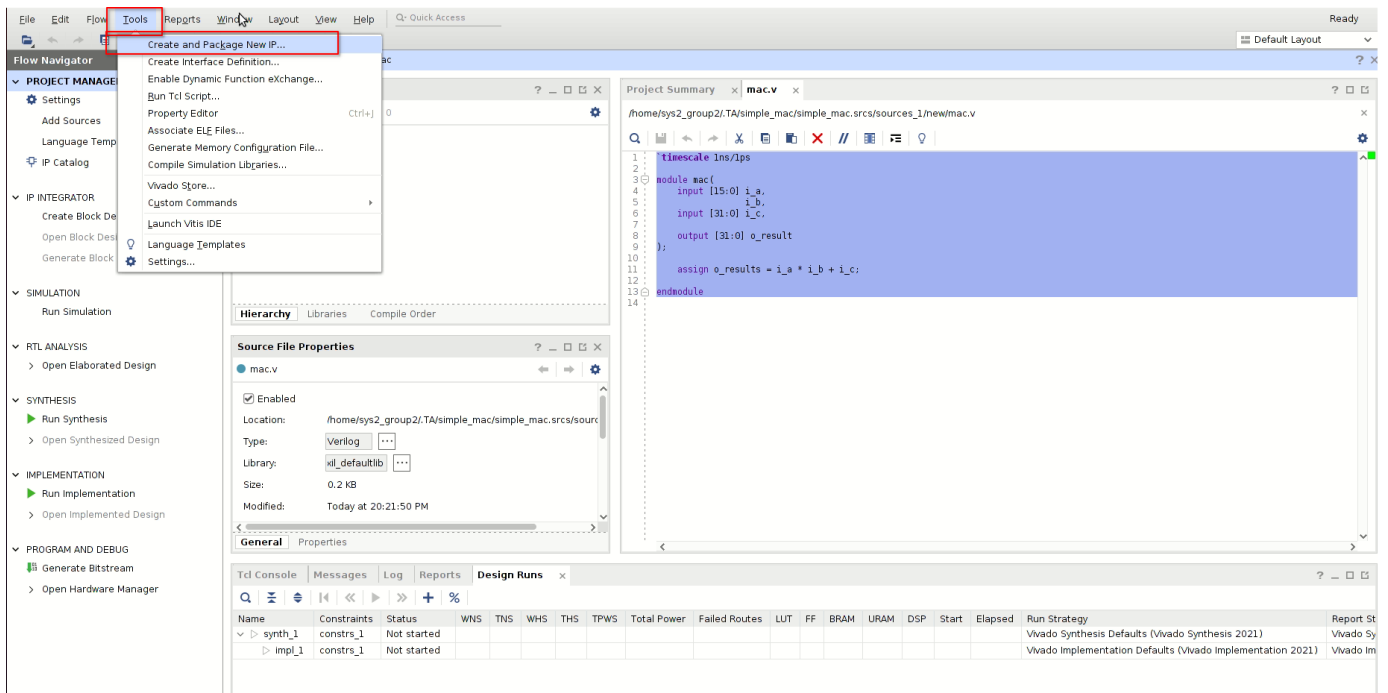
You'll notice that this module is purely combinational in nature. This is just a simple observation, it has no consequences later on.

Save the file. In a more realistic project, you'll design your IP, and then write a testbench to verify that the IP is working as per expectations. We are not going to do that here for such a simple module.

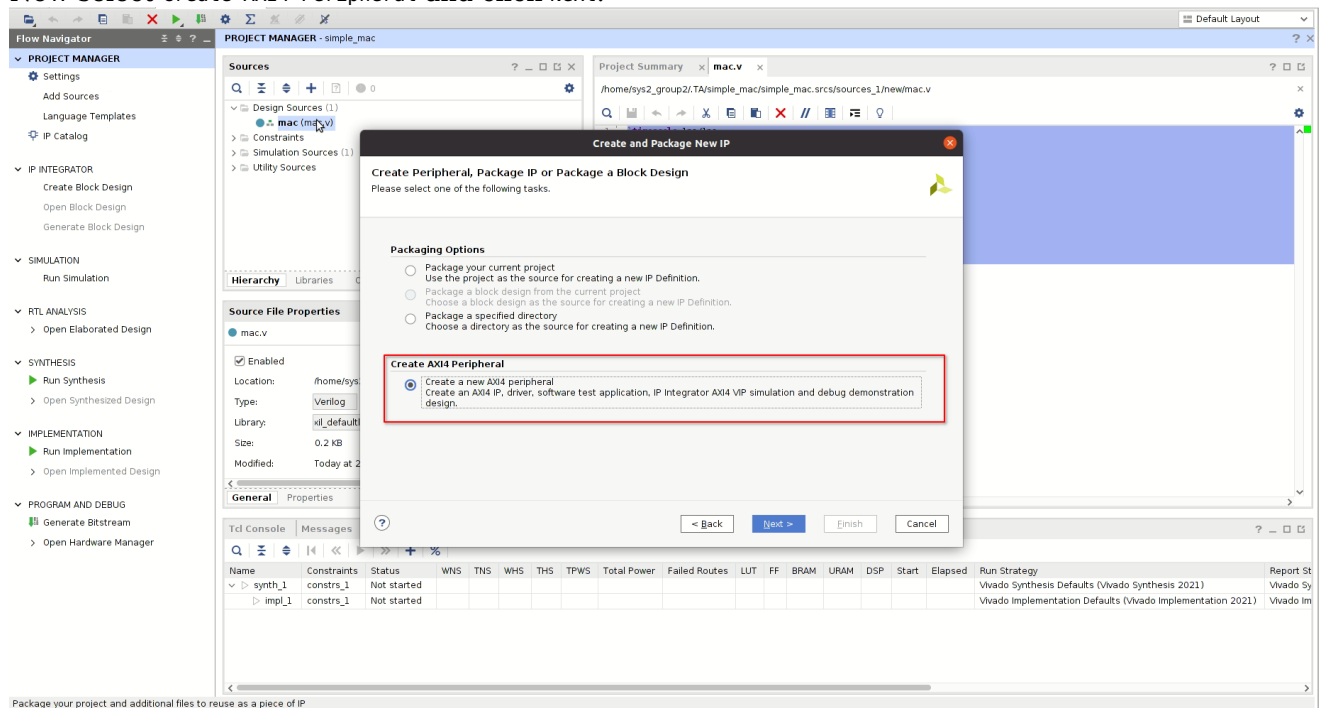
Step 3 - packaging your IP

Now, that the Verilog code has been written (and verified), we will package this project into an IP, along with the AXI4-Lite port mentioned earlier.

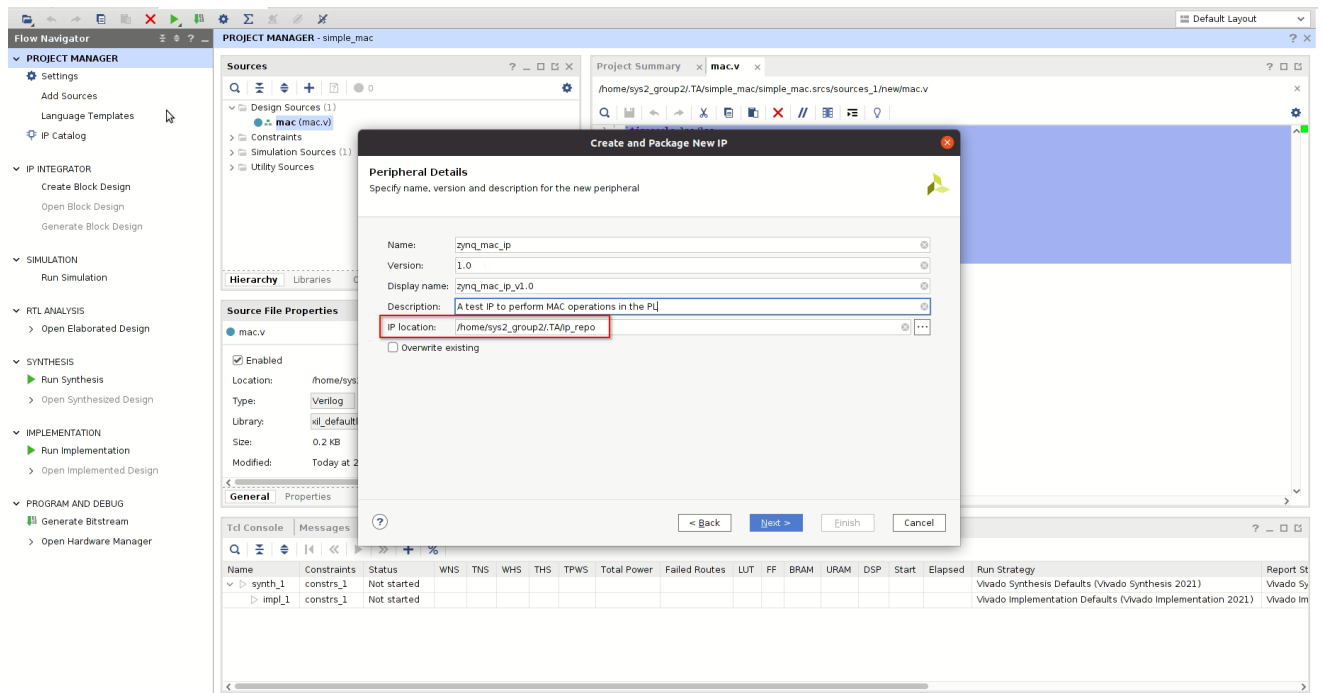
To do this, click on the `Tools` menu at the top:



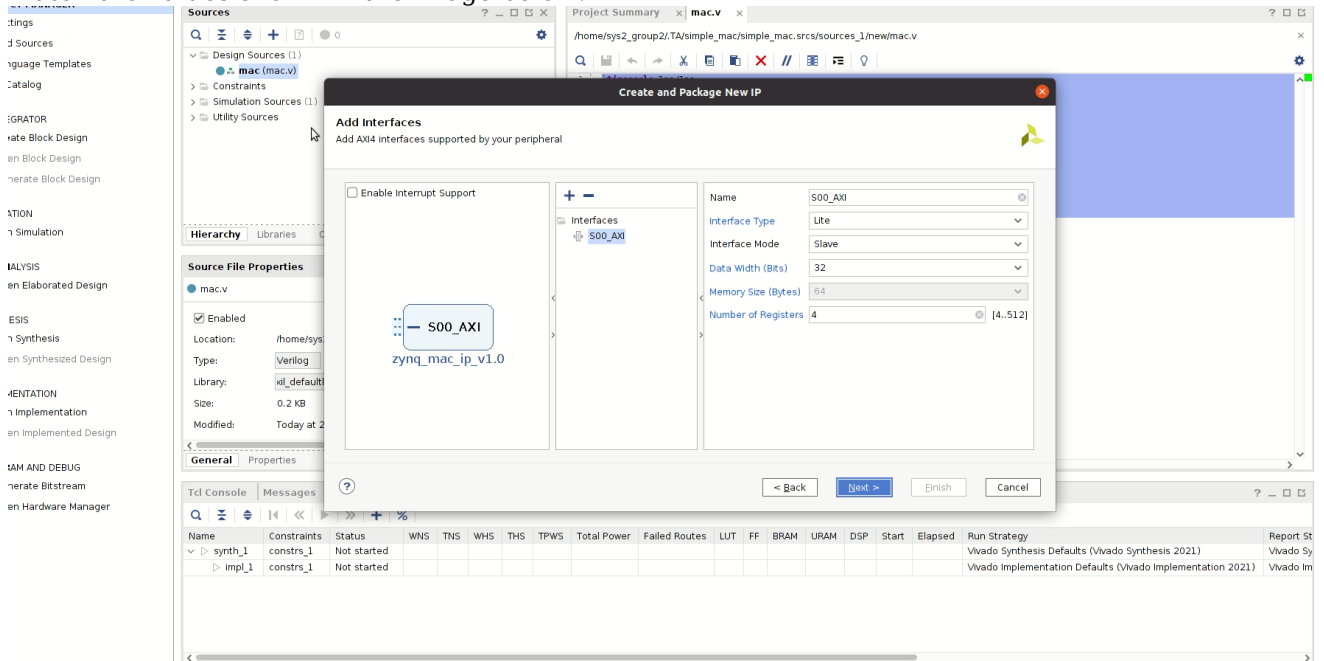
- Click on Create and Package new IP...
- Click on Next in the Window that opens
- Now select Create AXI4 Peripheral and click Next:



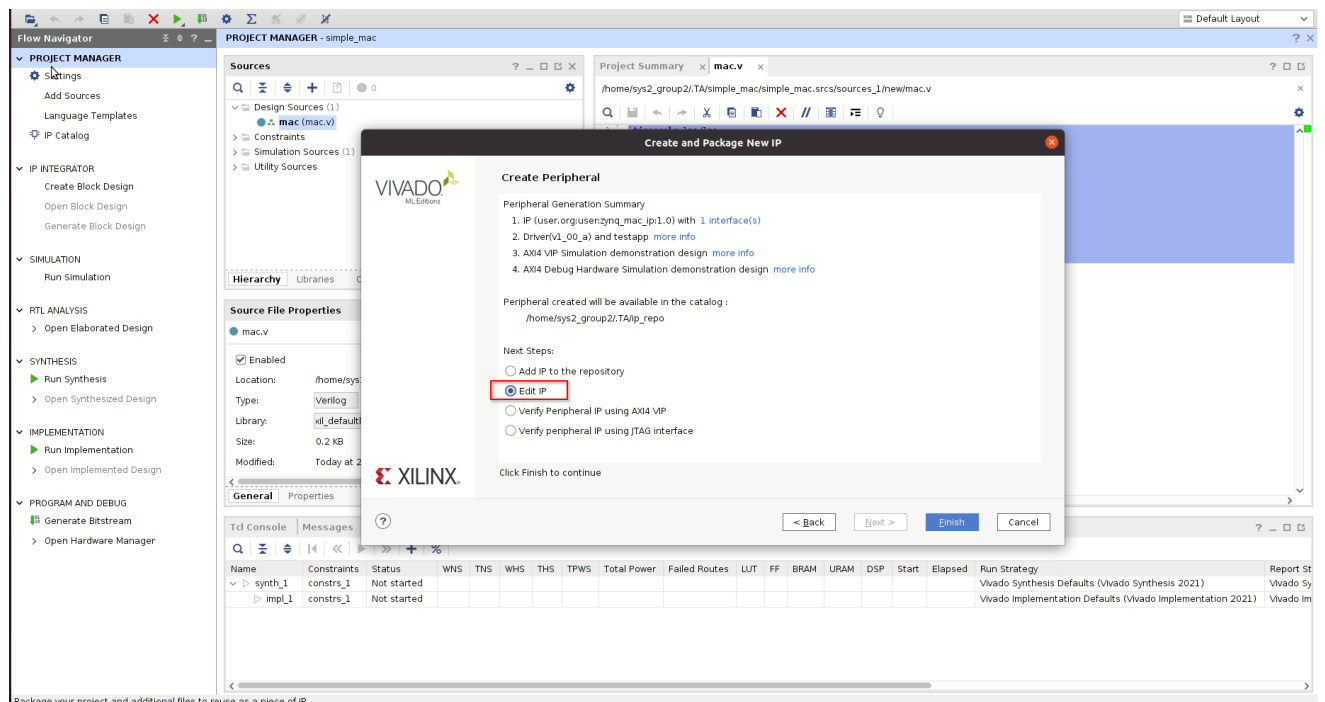
- Fill in the relevant details, give a proper name to your IP and **remember the IP location**. Click Next:



- In the next window, you'll see the configurations for the AXI4-Lite interface. Make sure that they match the values shown in the image below:



- In the last window, click on Edit IP:



The last step will open another Vivado window.

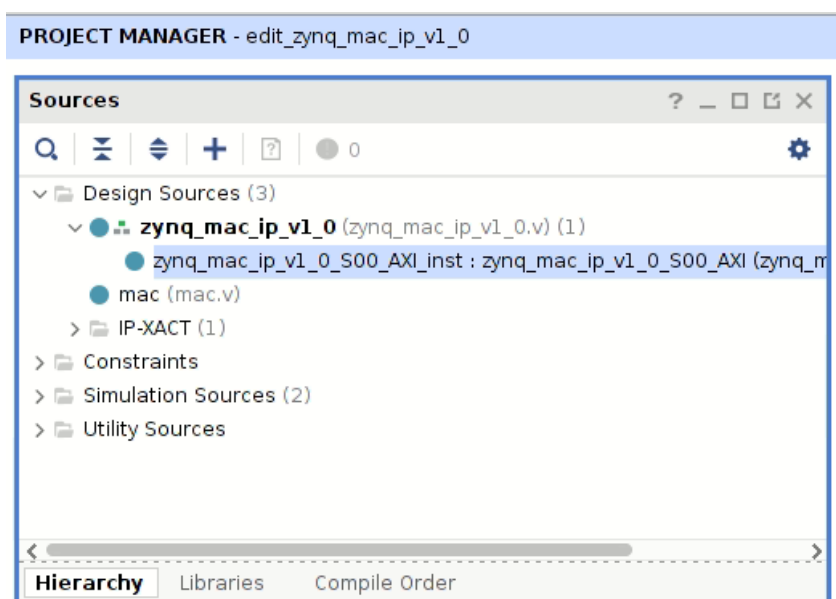
Step 4 - connecting your code to the AXI4-Lite interface

The first thing that you need to understand is that AXI4 is a very sophisticated and complicated protocol of transferring data between 2 endpoints. It is highly used in very high performance ARM cores (among many other places). AXI4-Lite is a simpler version of AXI4 (Full), but even then, it is very complicated. We won't be diving into any details in this document. We can write entire books on how AXI4 works and not do complete justice to it.

In the sources tab of the new Vivado window, you'll notice 2 Verilog files, which don't contain the mac module anywhere!

So the first thing you need to do is create a new file in this new project, and copy the mac module from the other(original) Vivado project into this new project.

The sources tab will look like this(with different file names):



You'll notice that the mac.v file is added to the project, but it is not being used anywhere. Now, we need to connect our code with the *v1_0_S00_AXI.v file. In my case, this file is called zynq_mac_ip_v1_0_S00_AXI.v. Open this file, then make the following changes:

```

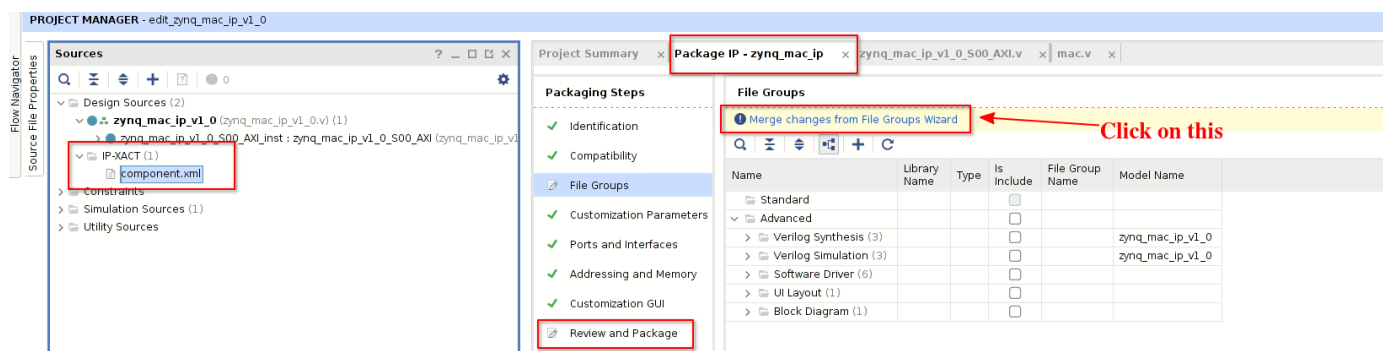
/home/sys2_group2/TA/ip_repo/zynq_mac_ip_1.0/hdl/zynq_mac_ip_v1_0_S00_AXI.v
364 :
365 // Implement memory mapped register select and read logic generation
366 // Slave register read enable is asserted when valid address is available
367 // and the slave is ready to accept the read address.
368 assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
369 always @(*)
370 begin
371     // Address decoding for reading registers
372     case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
373         2'h0 : reg_data_out <= slv_reg0;
374         2'h1 : reg_data_out <= slv_reg1;
375         2'h2 : reg_data_out <= mac_output;
376         2'h3 : reg_data_out <= slv_reg3;
377         default : reg_data_out <= 0;
378     endcase
379 end
380
381 // Output register or memory read data
382 always @( posedge S_AXI_ACLK )
383 begin
384     if ( S_AXI_ARESETN == 1'b0 )
385     begin
386         axi_rdata <= 0;
387     end
388     else
389     begin
390         // When there is a valid read address (S_AXI_ARVALID) with
391         // acceptance of read address by the slave (axi_arready),
392         // output the read data
393         if (slv_reg_rden)
394         begin
395             axi_rdata <= reg_data_out;    // register read data
396         end
397     end
398 end
399
400 // Add user logic here
401 wire [31:0] mac_output;
402
403 mac compute(
404     .i_a(slv_reg0[31:16]),
405     .i_b(slv_reg0[15:0]),
406     .i_c(slv_reg1),
407
408     .o_result(mac_output)
409 );
410 // User logic ends
411
412 endmodule
413

```

The changes are highlighted. Please note that the line numbers may vary, depending on the exact version of Vivado being used. Note the general locations and the blocks where the changes are made. Be very careful with this step. Save the file after making all the changes.

Step 5 - package the IP

After we have made changes to the AXI4-Lite wrapper, we need to save the Vivado project as an IP. To do this, we need to merge the changes that we just made. You will have a Package IP - <package name> tab open in Vivado. If it is *not* open, then double click on component.xml under IP-XACT in the Sources tab:



Click on Merge changes from File Groups Wizard link, highlighted at the top. After this, switch to the Review and Package tab, and click on Re-Package IP.

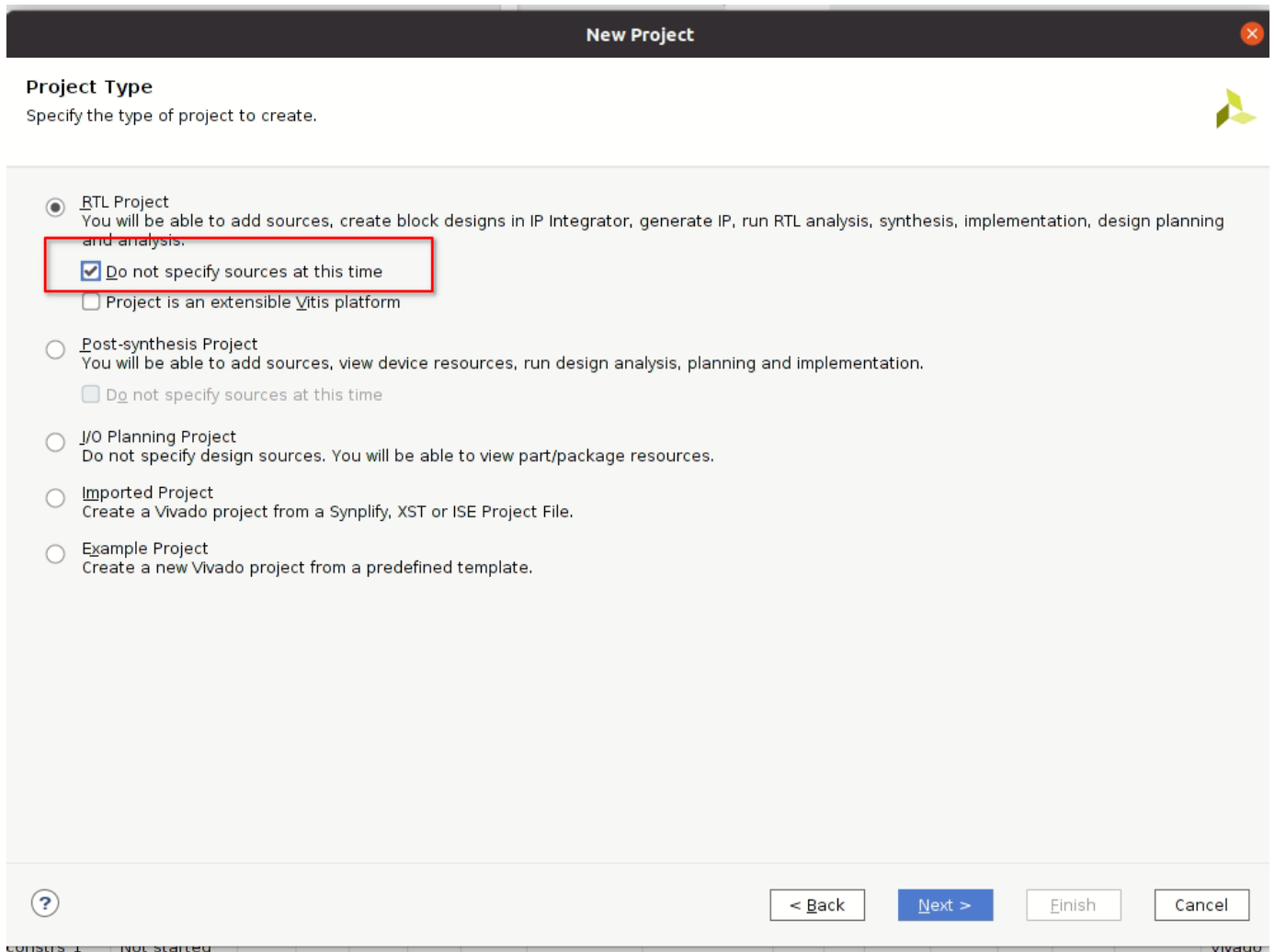
With this step, you have successfully created a new IP!

Using the IP in a Vivado project.

Step 1 - creating a new project

So far, you've created one Vivado project in which you verified the functionality of your code(the first Vivado project in this case). Then you opened another *temporary* Vivado project, where you edited the AXI4-Lite wrapper code and added your own logic. This temporary project generated the IP, which is stored in a specific **IP Location**(which you have noted in step 3).

Now, you need to create *another* Vivado project. In this project, we will use the IP that you just created. While creating the project, make sure to select the Do not specify sources at this time checkbox in the Project Type tab:



The screenshot shows the 'New Project' dialog box in Vivado. The title bar says 'New Project'. Below the title bar, it says 'Project Type' and 'Specify the type of project to create.' There are five radio button options for project types: 'RTL Project', 'Post-synthesis Project', 'I/O Planning Project', 'Imported Project', and 'Example Project'. Under the 'RTL Project' option, there is a checkbox labeled 'Do not specify sources at this time' which is checked and highlighted with a red rectangle. Other options include 'Project is an extensible Vitis platform' (unchecked), 'Do not specify sources at this time' (unchecked) under Post-synthesis Project, and 'Do not specify design sources. You will be able to view part/package resources.' (unchecked) under I/O Planning Project. At the bottom, there are buttons for '< Back', 'Next >', 'Finish', and 'Cancel'.

After this, make sure to select the correct board(Zedboard).

This will open a new Vivado project, with no source Verilog files. In this project, we will do things a little differently by using a graphical interface.

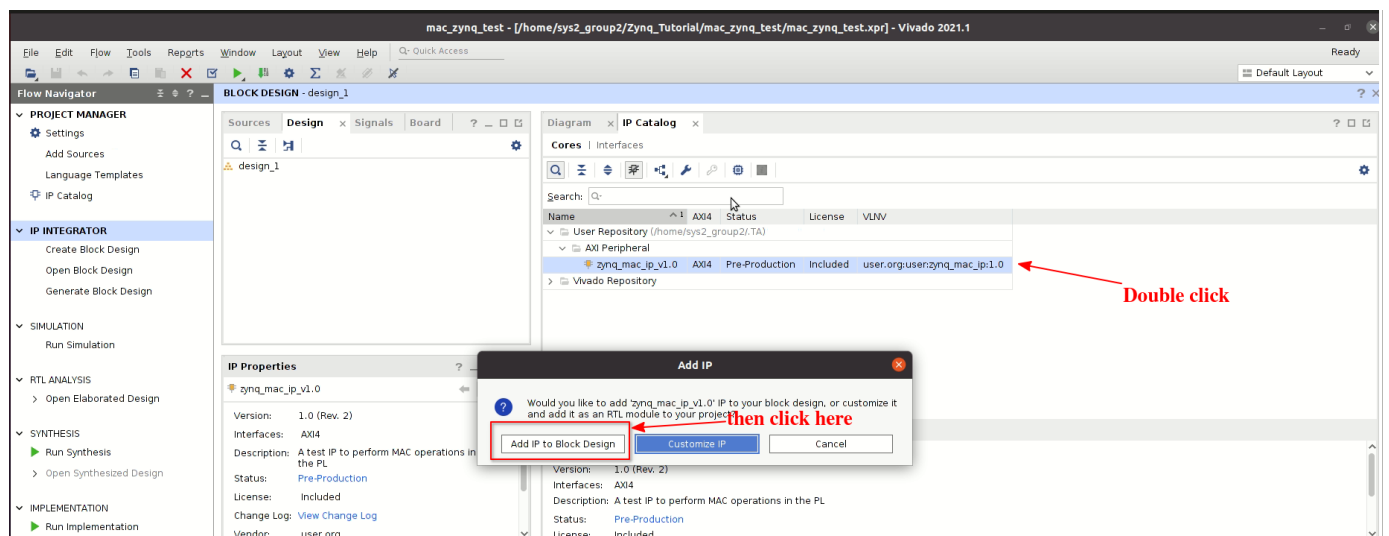
Step 2 - Creating a block diagram

In the Flow Navigator on the left, click on Create Block Diagram under IP Integrator(you can leave the name as design_1):

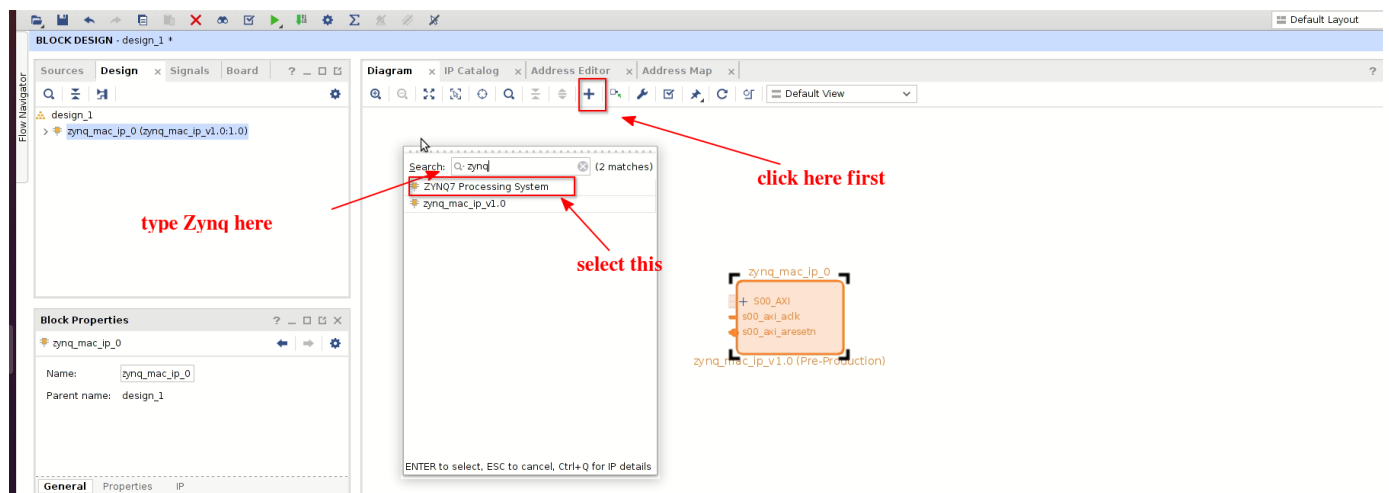
You'll notice that a large Diagram empty area opens up. We will add all the necessary IPs here. First step is

to add the IP that you just created. From your exercises with HLS, you know how to import a new IP into Vivado. I'm not explicitly showing this step, but you need to Add your IP folder into the IP catalog.

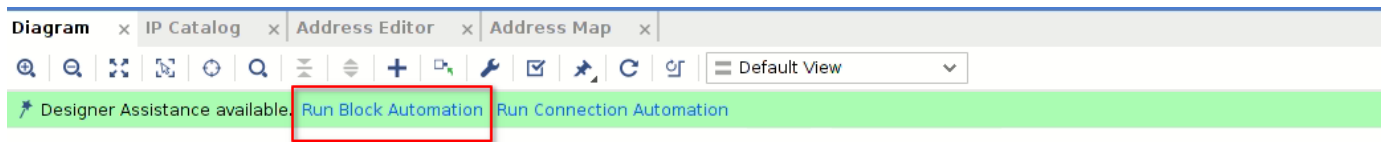
After this, double click on your IP in the IP Catalog, then click on Add IP to Block Design:



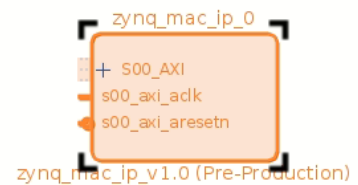
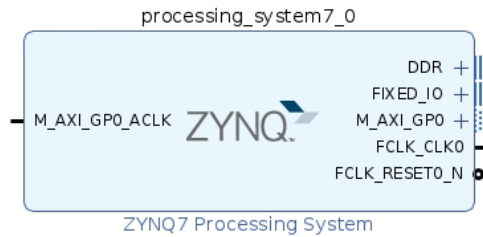
You'll notice that the IP has been added in the Diagram tab. Now, click on the + symbol in the Diagram tab, and add the Zynq7 Processing System IP as shown:



After adding the PS, it will look like this:

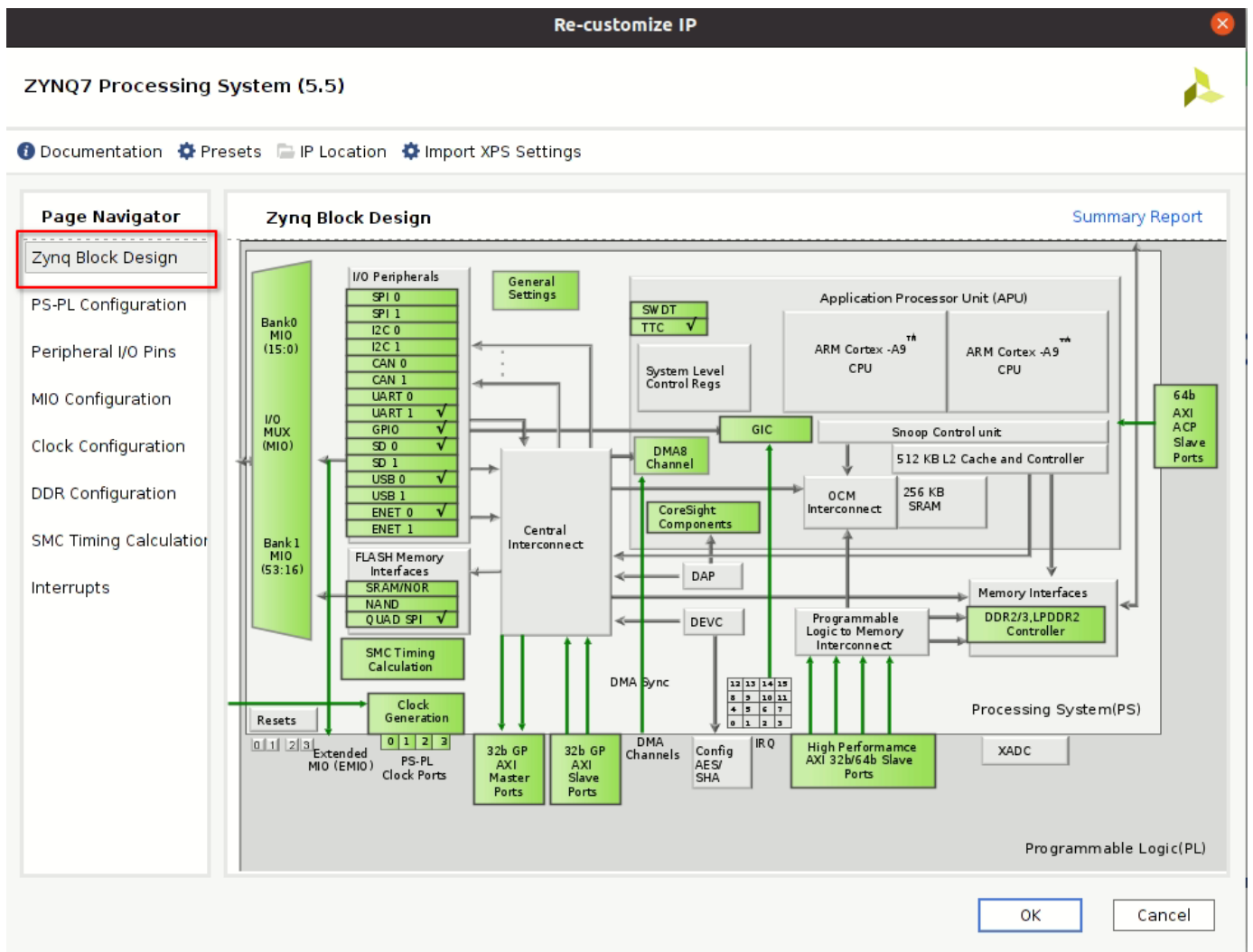


click on this



Now, click on Run Block Automation and click ok in the dialog box that opens. This will automatically configure the PS into a certain default configuration.

Optional step: When you run block automation, it enables a lot of additional things that are not necessary and can make things a little complex later. So we will customize the Zynq PS and disable things. To do this, double click on the processing_system7_0 block. A new pop-up will open, which looks like:

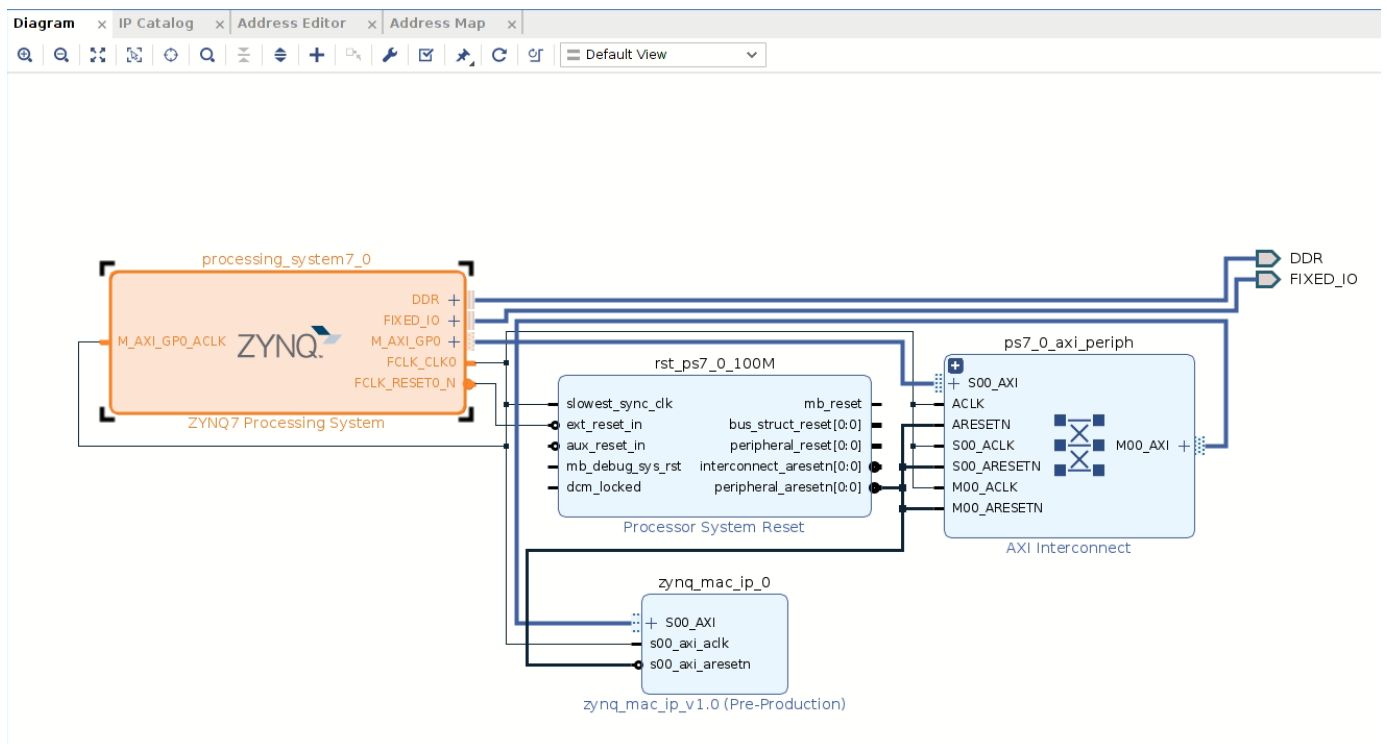


Except UART1, click on everything that has a tick beside it(one-by-one), and disable the component by clicking on the check box that it leads to. Note that the block design is in the Zynq Block Design tab. Once you are done, click on ok.

Now, we have added our IPs and placed them in the Block Diagram(BD). We need to connect the IPs together. This can be tedious, because there are a lot of signals. To make things simple, Vivado can automatically connect the IPs. To do this, click on Run Connection Automation at the top of the diagram, and click in ok. Do **NOT** change any of the configurations right now.

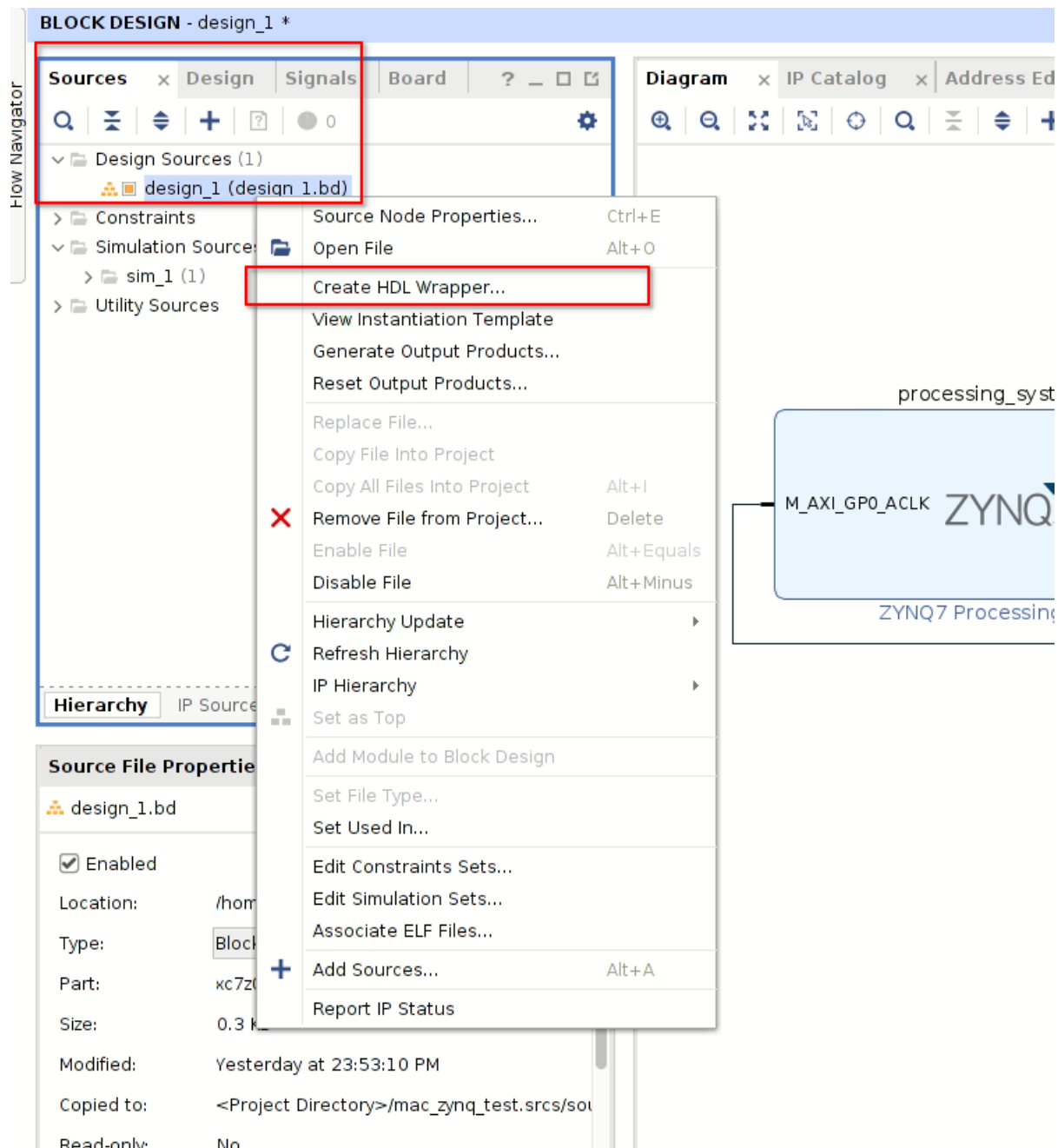
When you have more complicated designs, you might have to play with the automation settings to get things working.

After running connection automation, your diagram will look something like this:



You'll notice that additional IPs have been added automatically to the BD. These are necessary for the entire design to work, so Vivado has added them automatically.

Now press Ctrl+S to save the block design. After that, click on the Sources tab on the left, and then right click on the `design.bd` file in the Design Sources. Click on the Create HDL wrapper:



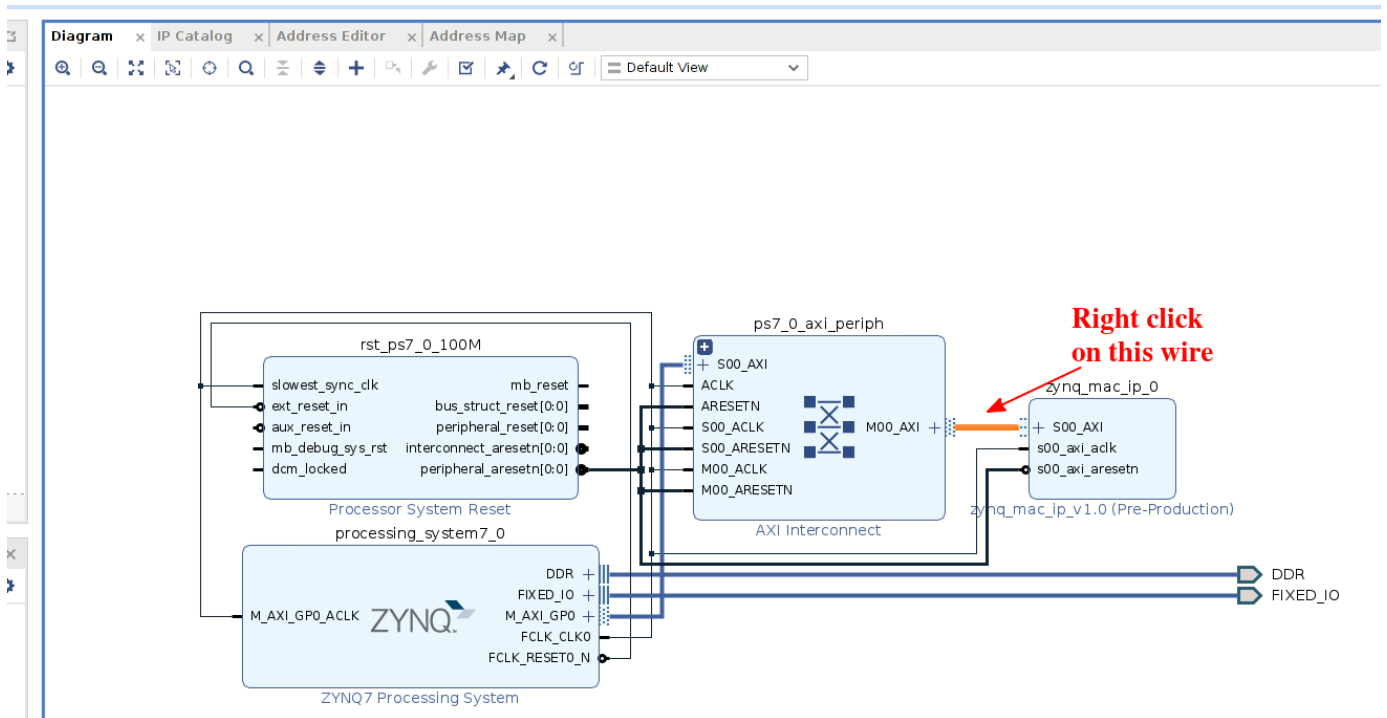
Select Let Vivado manage wrapper and auto-update and click ok. This step creates a Verilog file, which contains the Verilog form of the design you just made in the block diagram.

With this step, you are technically done creating the design!

Optional Step : adding an ILA

This is a recommended step, so that we can view the AXI4-Lite data movement later on.

To do this, right click on the wire connected to `s00_AXI` on your IP (highlighted in the image below):



Then select the Debug option. This will add a bug-like symbol on the wire, which indicates that you want to debug the signals on that wire. Also note that the Run Connection Automation option has re-appeared at the top. Click on it, and click on ok in the pop-up. **Do not change anything.**

You'll notice that a "System" ILA has been added to your design. This is functionally the same as a normal ILA, it has some extra intelligence to view the AXI4-Lite signals.

Step 4 : Generate bitstream and exporting the design

After you have saved the block diagram (after adding the System ILA), generate the bitstream. Be careful to not have any other applications open, because this is a RAM heavy step. Your laptop may crash if you have ≤ 8 GB RAM and have a browser open.

Once the bitstream is generated:

This step is different for those using Vivado versions before and after 2019.2.

A. Vivado versions ≤ 2019.2 :

- Click on File
- Click on Export
- Click on Export Hardware
- Select the Export Bitstream checkbox and click on Okay

After this step:

- Click on File
- Click on Launch SDK
- Give some time for SDK to open.
- The project's configuration will be loaded into SDK automatically.

Steps to do in the SDK will be added later, it's a lot of effort to type such a long document.

=====

B. Vivado version ≥ 2020.10

In these versions, Xilinx SDK is replaced by Vitis. The steps are slightly different for Vitis:

- Click on File

- Click on Export
- Click on Export Hardware
- A new pop-up window open, click on Next
- Select Include bitstream and click on Next
- Choose a path to export to, and remember the path. You can change the name of the XSA file too, but remember the name.
- Click Next and the Finish.
- Click on the Tools Menu at the top, and then click on Launch Vitis IDE
- You'll be asked about a workspace, just click Launch, it doesn't really matter
- A blank Vitis window will open, which does *not* have the project configurations
- Click on Create Application Project and click Next in the pop-up window
- In the next tab, select Create a new new platform from hardware (XSA). In the XSA file, select the path where you exported your project. There will be a XSA file in the folder select that:

New Application Project

Platform

Note: A platform project will be generated automatically in workspace for the selected XSA. It can be customized later.

Select a platform from repository | **Create a new platform from hardware (XSA)**

Hardware Specification

XSA File: /home/sys2_group2/Zynq_Tutorial/zynq_mac_test/design_1_wrapper.xsa Browse...

Boot Components

☒ Generate boot components

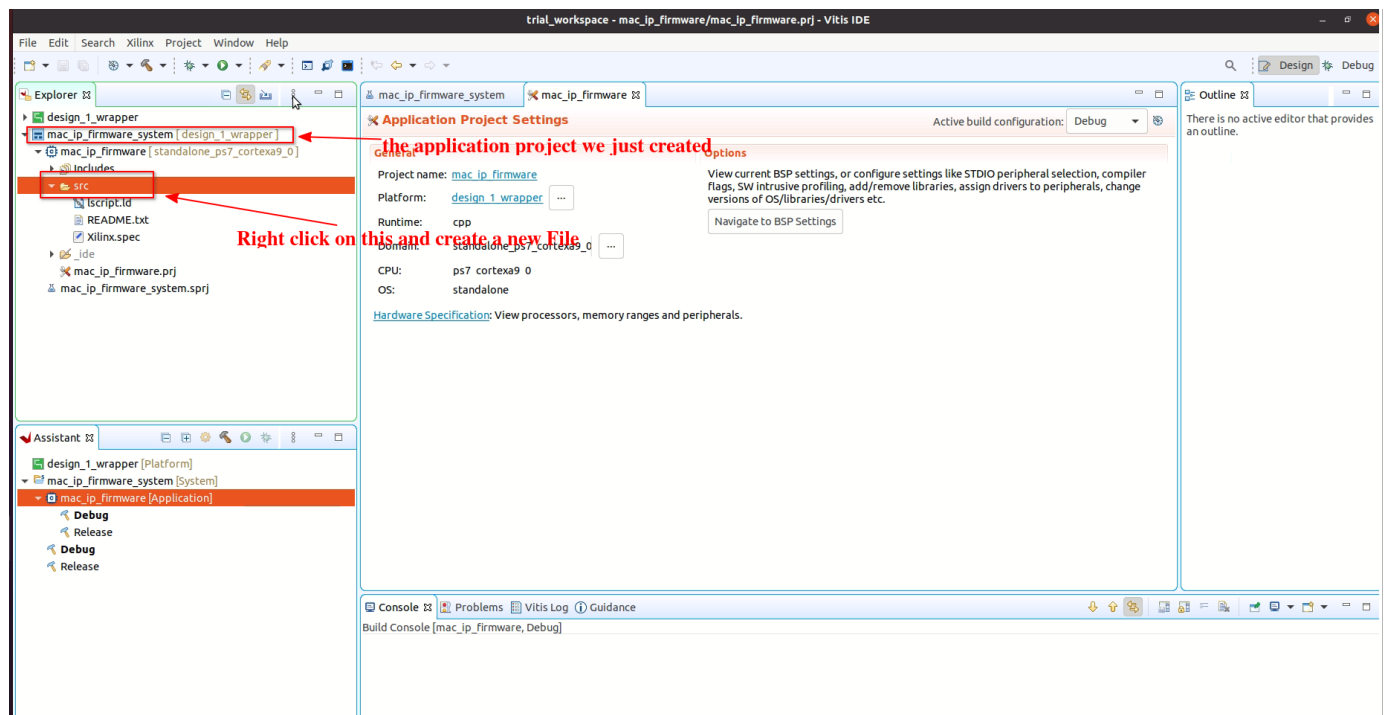
Platform name: design_1_wrapper

< Back **Next >** Cancel Finish

- Click on Next
- In the Application Project Details tab, give some name to the application, and then click Next.
- Click Next twice, till you reach the Templates tab. Select Empty Application(C) and click on Finish.

After alllll these steps, you've finally created a Vitis project, which will be used to program the PS. It will

look like this (with different names possibly):



- Right click on the src folder --> New --> File.
- Name the file main.c and open it.
- Paste this program in it (**Notice that some of the macros might be named different based on the name of your IP**):

```
#include "xil_printf.h"
#include "xil_types.h"
#include "xparameters.h"
#include "xil_io.h"
#include "zynq_mac_ip.h"

// This might be different in your case
// The names depend on what you called your IP
#define BASEADDR XPAR_ZYNQ_MAC_IP_0_S00_AXI_BASEADDR
#define REG0_OFFSET ZYNQ_MAC_IP_S00_AXI_SLV_REG0_OFFSET
#define REG1_OFFSET ZYNQ_MAC_IP_S00_AXI_SLV_REG1_OFFSET
#define REG2_OFFSET ZYNQ_MAC_IP_S00_AXI_SLV_REG2_OFFSET

// This is all going to be the same
int main(){
    s16 mul_operand1, mul_operand2;
    s32 add_operand;

    mul_operand1 = 11;
    mul_operand2 = 23;
    add_operand = 10;

    s32 axi_reg0_data = (mul_operand1 << 16) | mul_operand2;
    s32 axi_reg1_data = add_operand;

    s32 calculated_result;

    xil_printf("Sending the input operands\n\r");
    ZYNQ_MAC_IP_mWriteReg(BASEADDR, REG0_OFFSET, axi_reg0_data);
    xil_printf("Sent data to the AXI-Lite Reg0\n\r");

    Xil_Out32(BASEADDR + REG1_OFFSET, axi_reg1_data);
    xil_printf("Sent data to the AXI-Lite Reg1\n\r");

    calculated_result = Xil_In32(BASEADDR + REG2_OFFSET);
    xil_printf("Read AxiLite Reg2: %dx%ld+%ld=%ld\n\r", mul_operand1, mul_operand2, add_operand, calculated_result);
    return 0;
}
```

The steps after this will be discussed in class, it takes a lot of time to type all this out