Praktikum/Forschungpraktikum Embedded System Design

Chair for Processor Design, cfaed, TUD

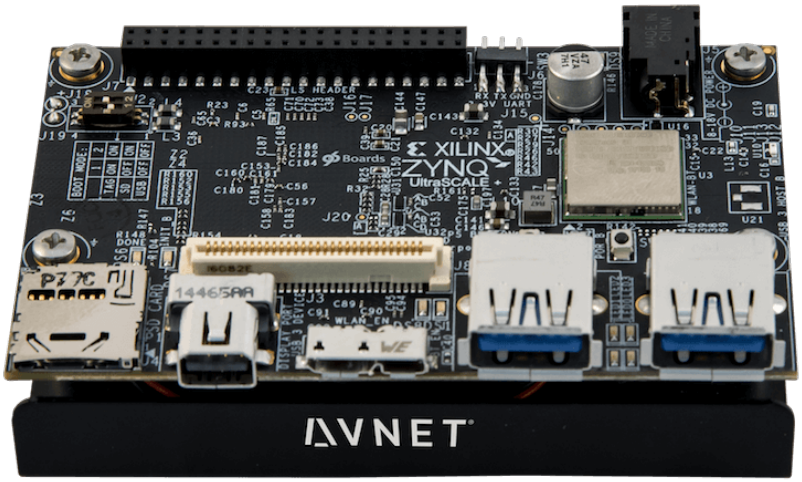
Winter Semester, 2019-2020

Lab 1 – Introduction to Working with Zynq-based System within Xilinx Vivado

In this lab, the students will be guided through two tutorials to know how to instantiate the Zyng Processing Subsystem (PS) within the Xilinx Zynq-based FPGA. After that, the students will write a simple C/C++ code to run on ARM inside the PS to interact with the outside world through the UART Serial interface. Next, the students will implement a simple hardware on the Programmable Logic region of the FPGA. The ARM interacts with this hardware through the AXI-Lite interface perform a simple addition operation. The purpose of this lab is to show the students the basic structures of a computer architecture how the software (running on ARM) can communicate with the hardware (running on the FPGA). There are two homework assignments after the lab which focus more on the hardware development with Hardware Development Language. The students can choose Verilog, VHDL or any other language for development which is supported by Xilinx Vivado.

Introduction

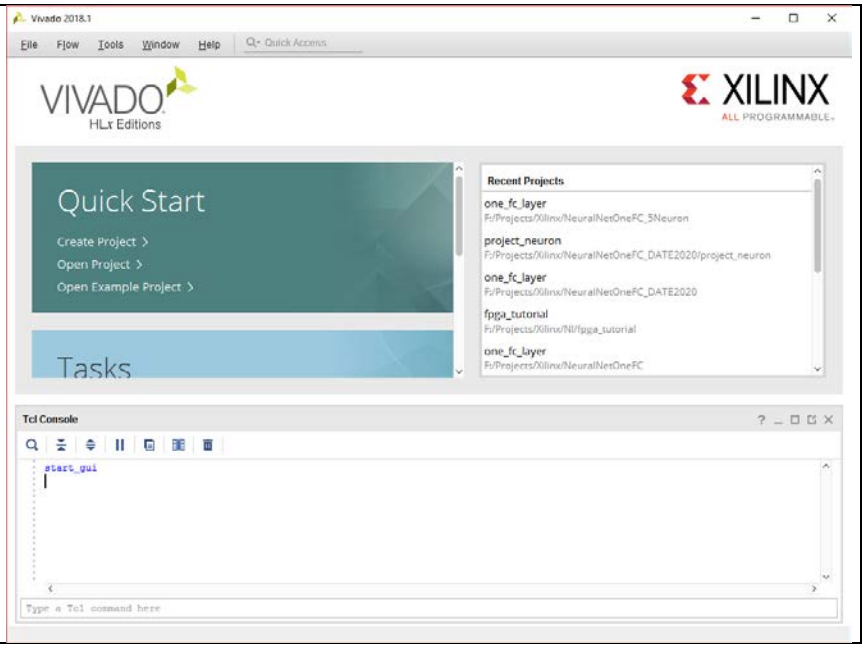
The development board that we are going to use throughout the module is Avnet Ultra96 v1 as shown below. It is equipped with the latest Xilinx Zynq UltraScale+ ™ MPSoC and other peripherals such as Wifi, Bluetooth, USB-3 ports, etc. More information of the board can be found here.

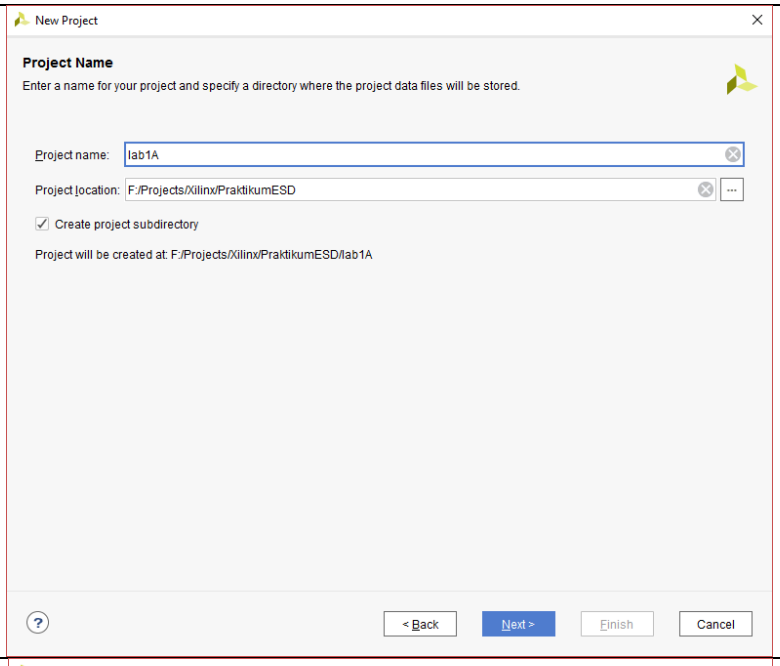
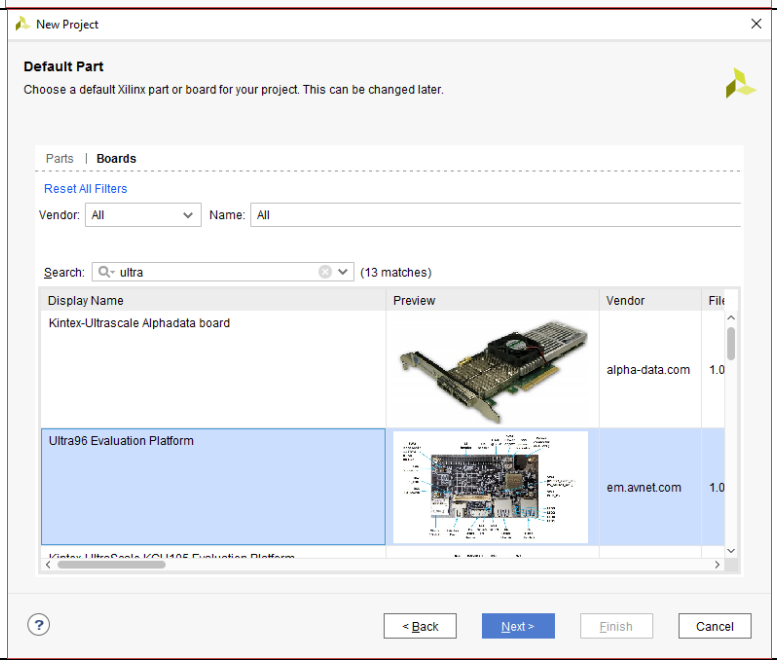


Besides, in order to work and debug with the board, an extra circuit is required to enable JTAG. It allows loading bitstream and ARM executable code from the PC to the board. It is also possible to use Vivado Chipscope to analyze the signals of the hardware accelerators within the FPGA via JTAG. That circuit is Ultra96 USB-to-JTAG/UART Pod. It is attached to the board as follows. More information about it can be found here.

The design tool used in this module is Xilinx Vivado Design Suite - HLx Editions version 2018.1. It is free to download and there is a 30-day evaluation period with full features. However, if the students are accessing our lab server, they can have access to our licenses to use Vivado. The students can download the tool here. Please note that the students need to register for an account on Xilinx to get the tool.

Lab 1 – A

Open Vivado, click "Create Project" to create a new project.	

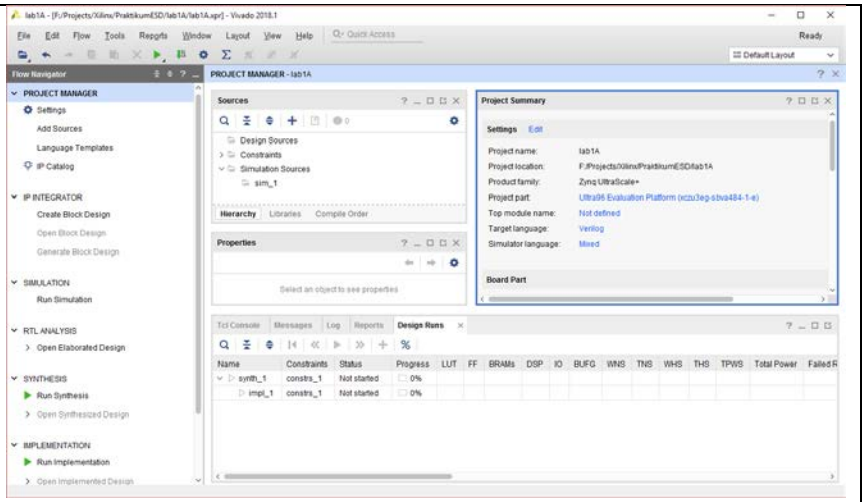
It asks for the name and location of the project. Please note that, there should not be any space character in the path as well as the name of the project.	
Click Next (keep all options default) until the tool asks to choose the Parts/Boards. Search for Ultra96. If it is not listed there, please follow the instructions HERE to add it to Vivado. Click Next, then Finish. It takes a while for Vivado to initialize the project.	

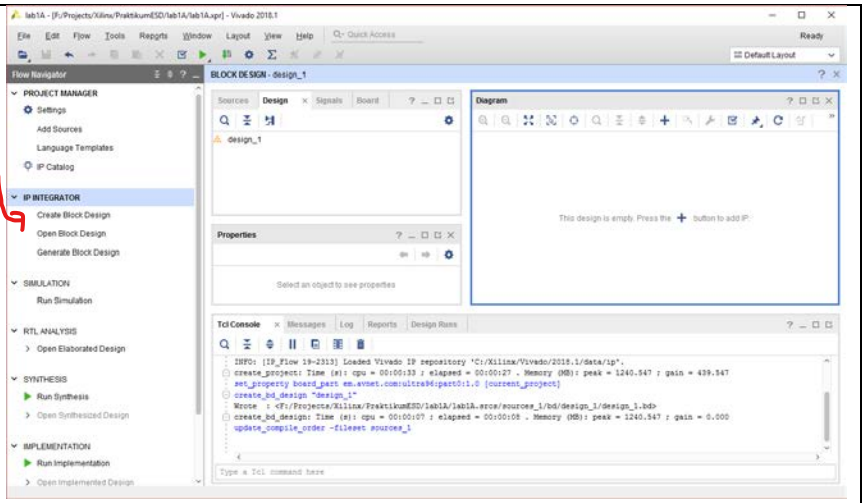
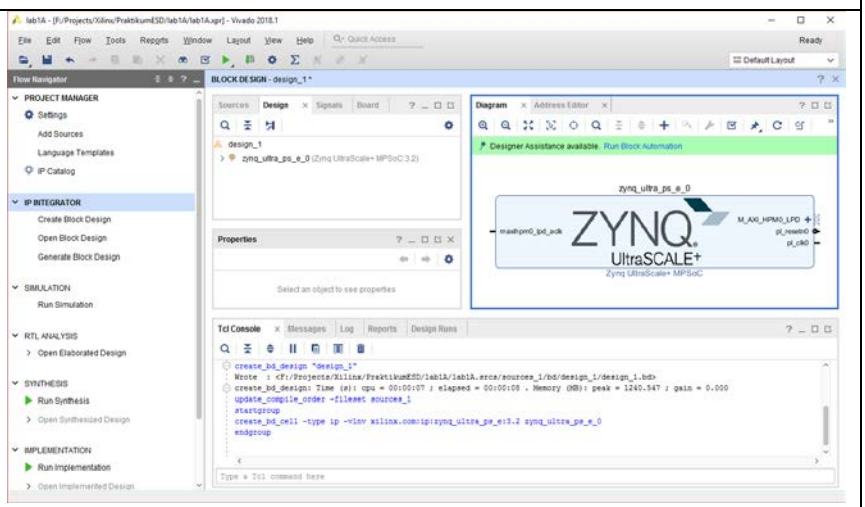
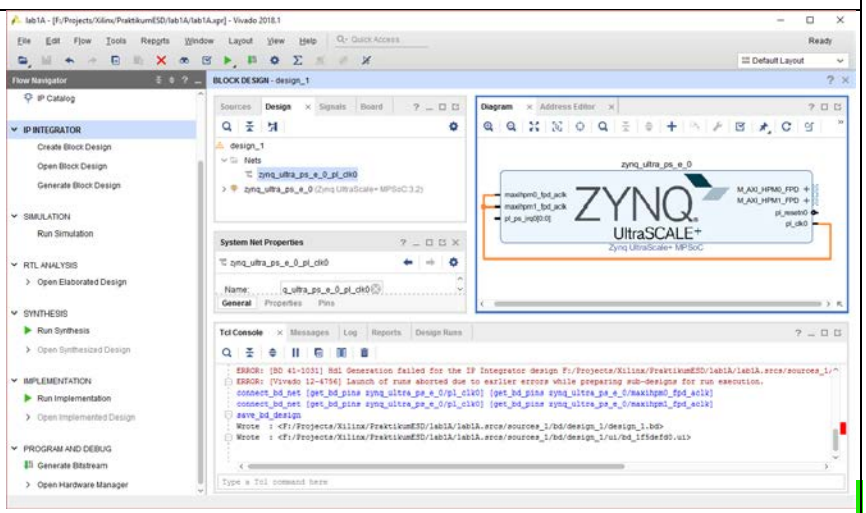
This is the main GUI of Vivado where most of the works are done. The Flow Navigator pane on the left shows most of the basic steps throughout the design flow.

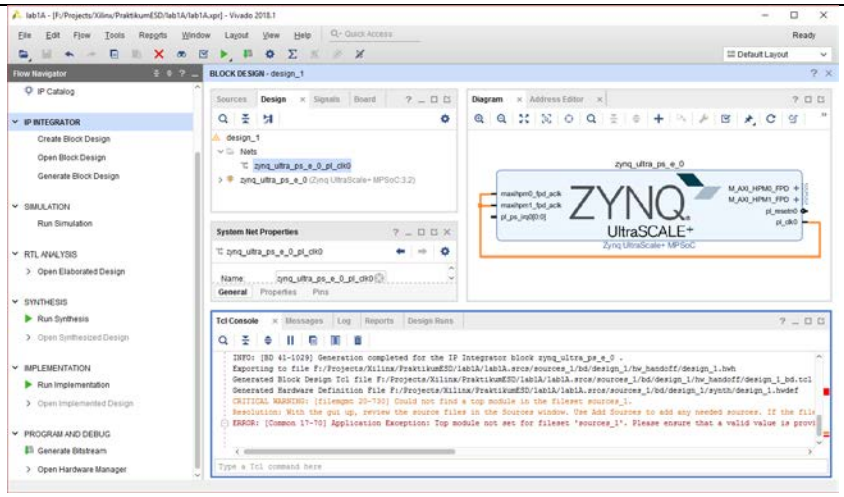
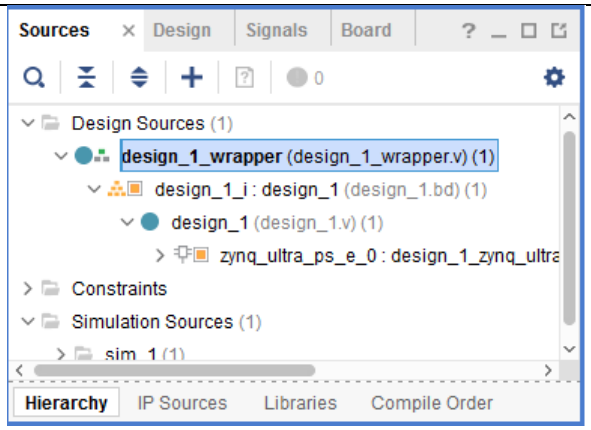
The Sources panel lists all of the hardware design, simulation and constraint files used in the project.

In the lower right panel, there are multiple tabs.

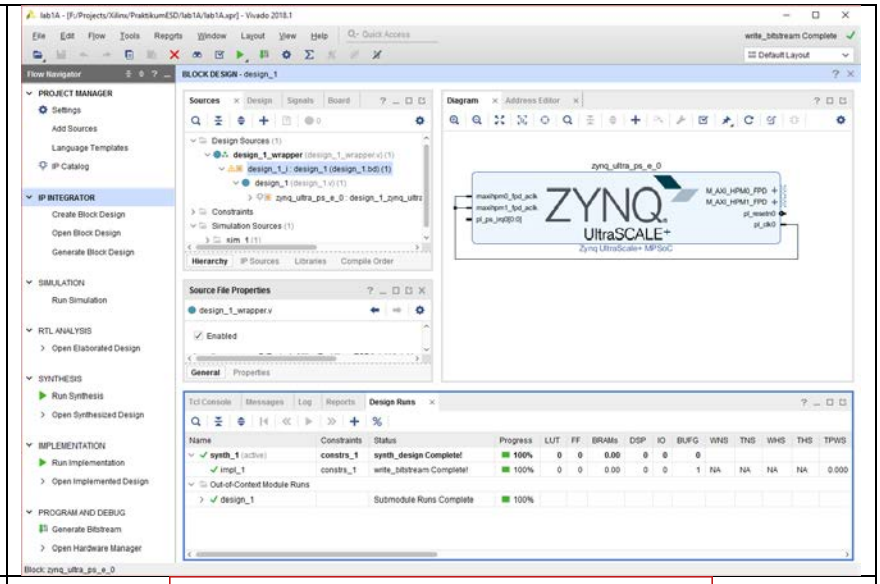
- The Tcl Console shows all of the TCL commands used by Vivado to interact with the project. It means that everything can be done by a TCL script without opening the GUI.
- The Messages tab summarizes all of the messages written in the Log tab.
- The reports for each development step are shown in the Reports tab.
- In the Design Runs tab, you can control the synthesis/implementation processes with different constraints and parameters to synthesize the design as well generating the bitstream to load into the FPGA.



In the Flow Navigator pane, under the IP INTEGRATOR, click “Create Block Design”, choose the “Design Name” if needed, then click OK. The block design environment will be opened with the Diagram panel. This most useful feature provided by Vivado is to enable drag-and-drop mechanism in designing with FPGA by wrapping the IPs (hardware libraries) into blocks. The blocks can be easily connected together either automatically if they are known by Vivado, or manually by using the mouse.	
Click the blue plus button to add IP. Search for the Zynq UltraScale+ MPSoC. Double-click the entry to add the Zynq block into the design. This step is necessary to enable the ARM cores within the PS. It is also required to connect the hardware accelerators to the PS. Click “Run Block Automation” to initialize the block based on the board configuration.	
Press “Ctrl + S” to save the design. Click “Generate Bitstream” in the Flow Navigator. If there is no issue with the design, Vivado will generate the bitstream. However, in this case, after checking the design, it detects that some clock pins are not properly connected. Those pins are for the AXI interfaces which are enabled by default. You can either double click on the Zynq block to try to disable them, or you can connect the	

"pl_clk0" pin of the Zynq block to those AXI clock pins.	
This time, Vivado complains that the Top module is not set. At this stage, only the high-level block design is made. Vivado, which is made to work with HDL-based designs, doesn't know which one is the top module of the design to start implementing from that. A block design can either be a top module of used inside another HDL module. In the latter case, an IP can be easily created from other IPs by using block design and then used as a big IP in other module. It is one of the biggest features of Vivado to make it easier to work with FPGA.	
Now, go to the Sources tab, right click on the block design that is just created (design_1.bd). Click "Create HDL Wrapper", choose "Let Vivado manage wrapper and auto-update". If this option is selected, from now on, every change is made for the block design will be updated automatically by Vivado. It is not needed to create a new wrapper anymore. After creating the wrapper, Vivado automatically chooses it as the top module (notice the little icon with three squares in which the top one is green).	

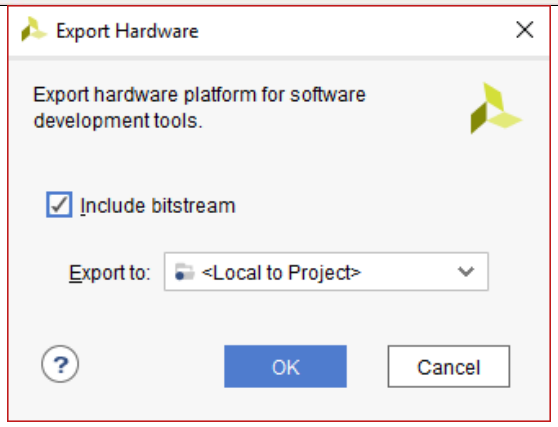
There should be no issue in generating the bitstream. The progress can be monitored in the “Design Runs” tab.



When the bitstream generation is finished, we need to export the hardware information to Xilinx Software Development Kit (SDK) to start working with ARM.

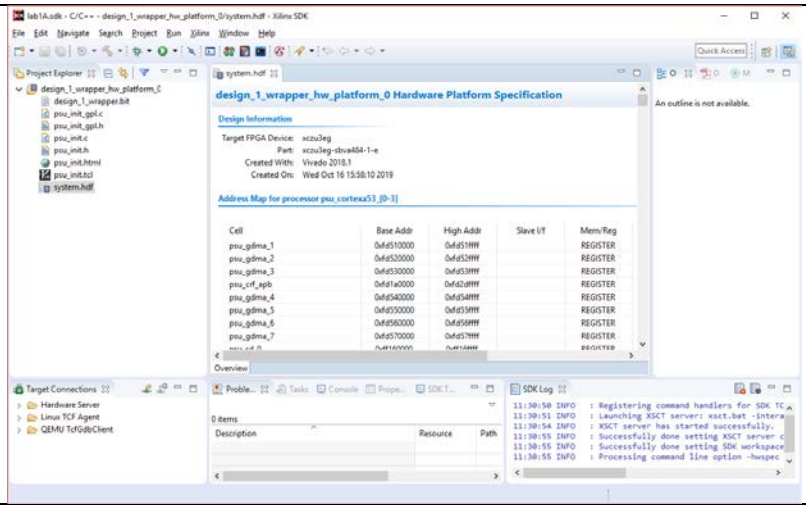
Click “File → Export → Export Hardware”, select “Include Bitstream” then “OK”.

After that, click “File → Launch SDK” then “OK” to open Xilinx SDK



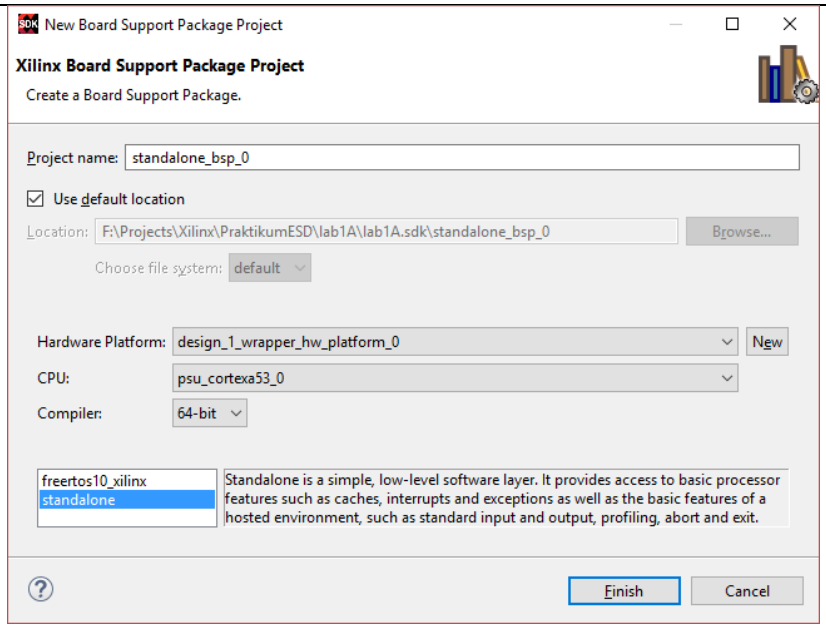
Xilinx SDK will be opened shortly after that. It is built based on Eclipse as shown in the screenshot.

The file system.hdf is the Hardware Description File generated by Vivado. It contains all information of the system including the list of components, their accessible addresses as well as their IP versions.



The next step is to create a Board Support Package to the system with the necessary drivers to control the IP.

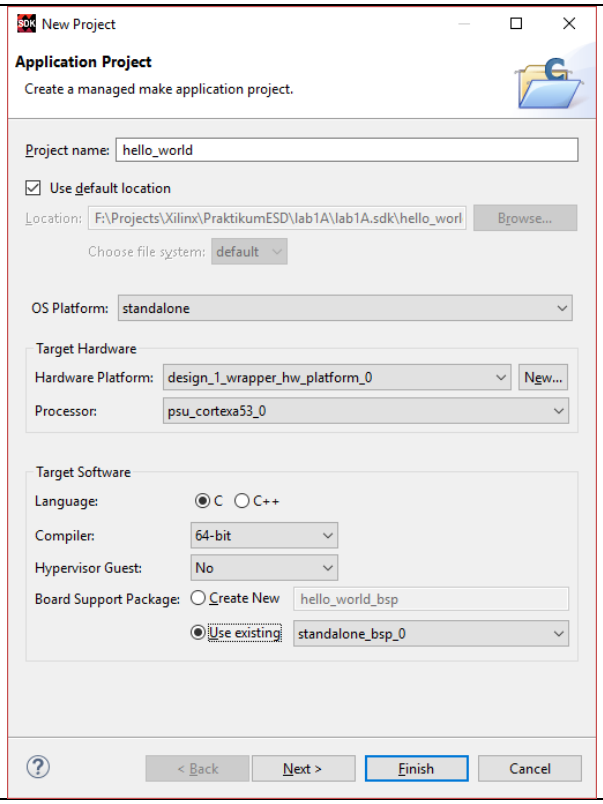
Click “File → New → Board Support Package”. Use the default configurations as shown in the figure. Then click “Finish”. Another prompt will be opened to configure the BSP or select the drivers to be included. Click OK to generate the BSP with default settings. The code will be generated and compiled automatically.



Now we need to create an application based on that BSP. Click “File → New Application”. Make sure that you select the BSP that was just created as a reference one. The processor must also match with the BSP.

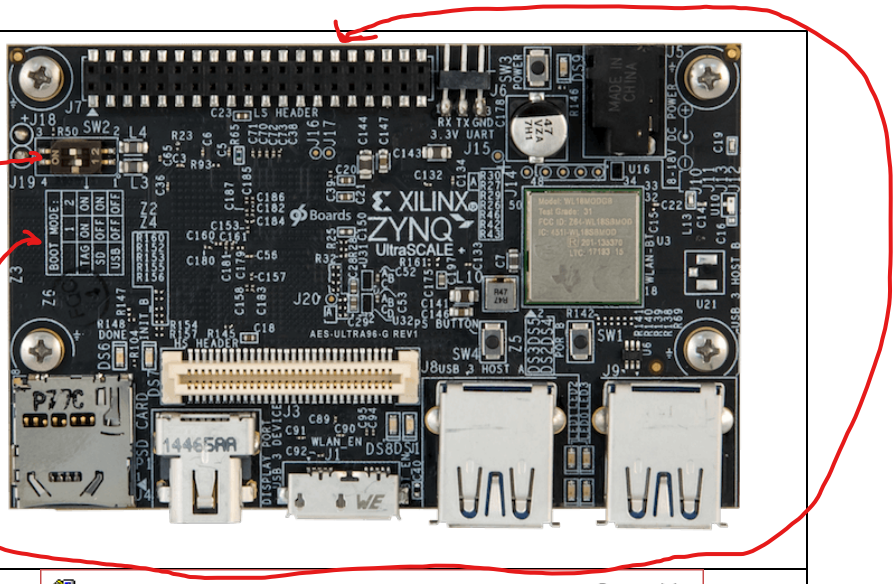
Click “Next” to choose the “Hello World” template, then “Finish”.

The hello world application will be compiled. Now it’s ready to be downloaded into the board.

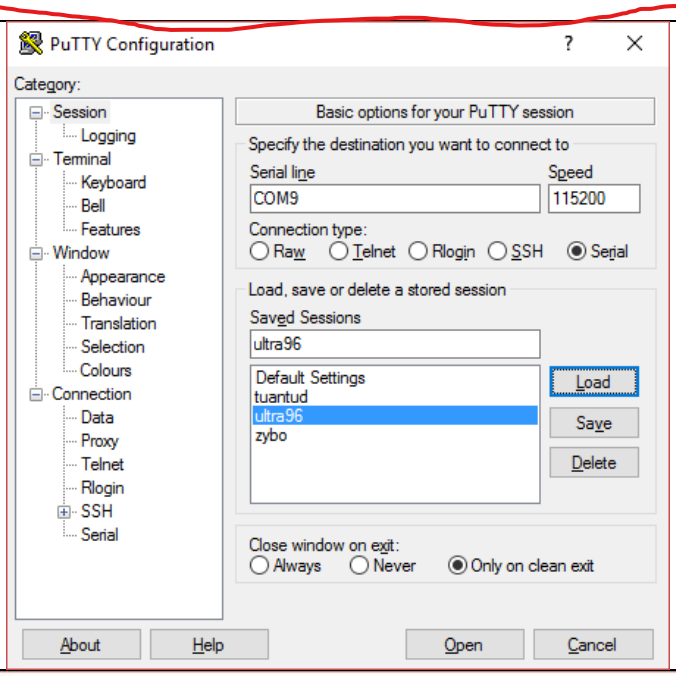


Before starting the board, make sure that the correct BOOT MODE is selected. We are going to work with the board via JTAG, please go ahead and look for the place where it can be configured. Hint: they are the physical switches on the board.

Once it is done. Plug in the JTAG pod, the micro-USB cable to that pod, power cable to the board then power it on.



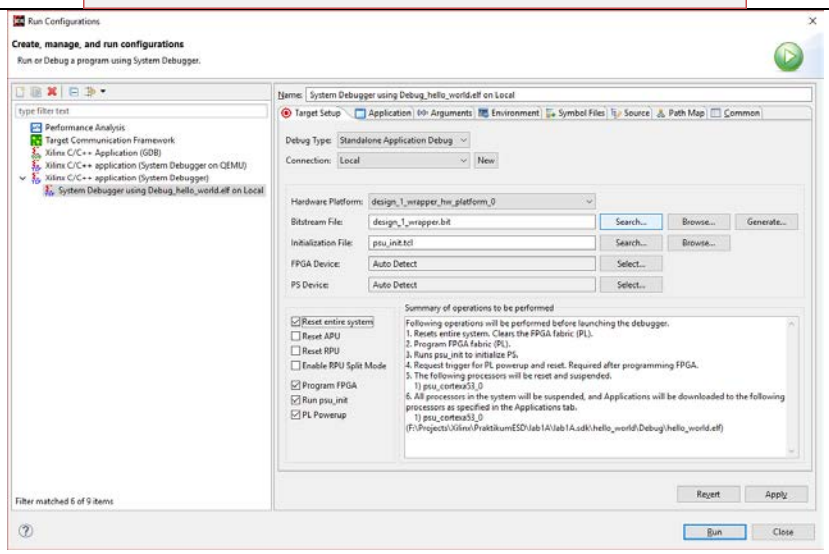
A serial console is needed to communicate with the board. The Port on your system may be different than this example. However, the speed must be set as shown.

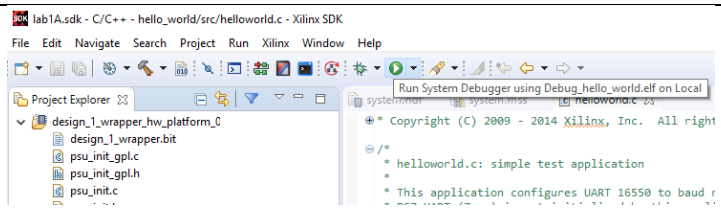
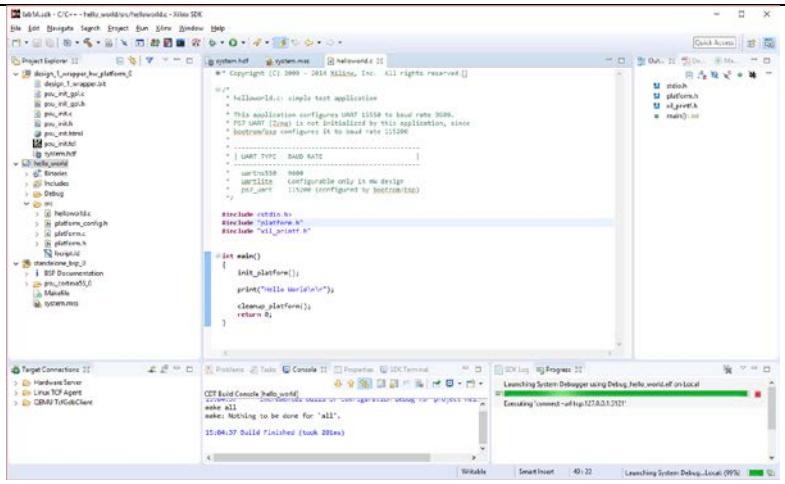
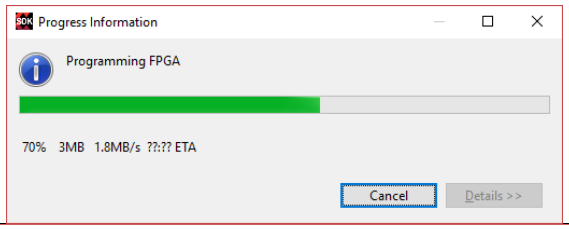


Now, go back to Xilinx SDK. Right click on the "hello_world" application → Run as → Run Configurations

Double-click on the "Xilinx C/C++ application (System Debugger)" to create a new configuration to run the application.

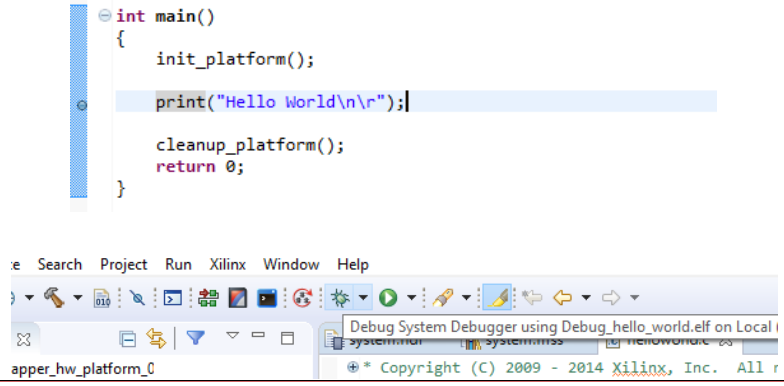
Make sure that the Bitstream File is correct and the "Reset entire system" + "Program FPGA" options are checked.



Click “Apply” to save the configuration. You can click “Run” to start downloading the bitstream as well as the executable file for the ARM. Alternatively, the green start button on the main GUI page can be used for the same purpose after making sure that the “hello_world” entry in the “Project Explorer” pane is highlighted.	
After running the application, the progress can be seen within the Progress Tab in the lower right corner. The bitstream will be downloaded to the FPGA as well.	 
Nevertheless, there is nothing printed on the console! It means that either the configurations are wrong or the application is not executed correctly.	

Let's try to debug the application on the board. Set the break point at the print statement. Then click the "Debug" button.

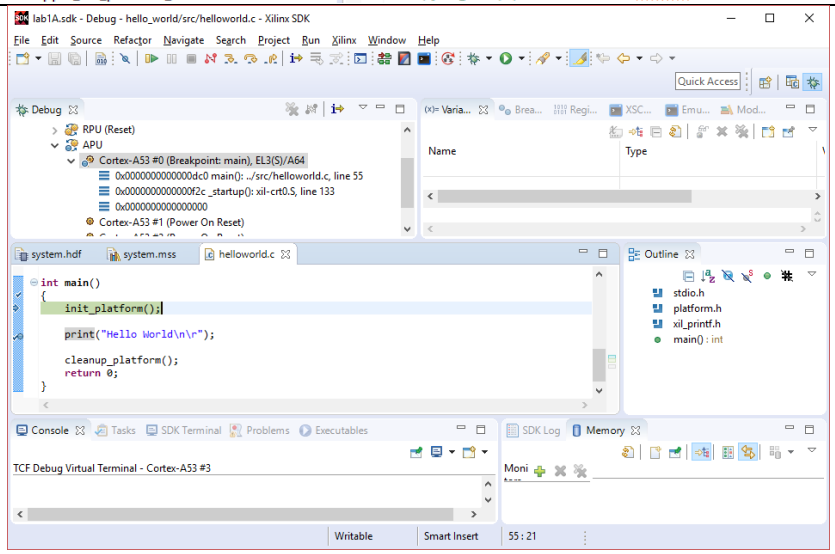
You will see that the executable file for ARM and the bitstream for FPGA will be downloaded.



When the Debug perspective is switched to and the application is successfully downloaded to the ARM, it will stop at the beginning of the "main" function. Press F6 to step over the function "init_platform" and the "print" statement.

It seems that the application is executed correctly. In this case, the possible reason is that the output from the ARM is not correctly routed through the JTAG to the PC. The BSP configuration should be checked. Please look for the solution by yourselves.

Hint: look for the tutorial on Avnet. You may need an account there to download.

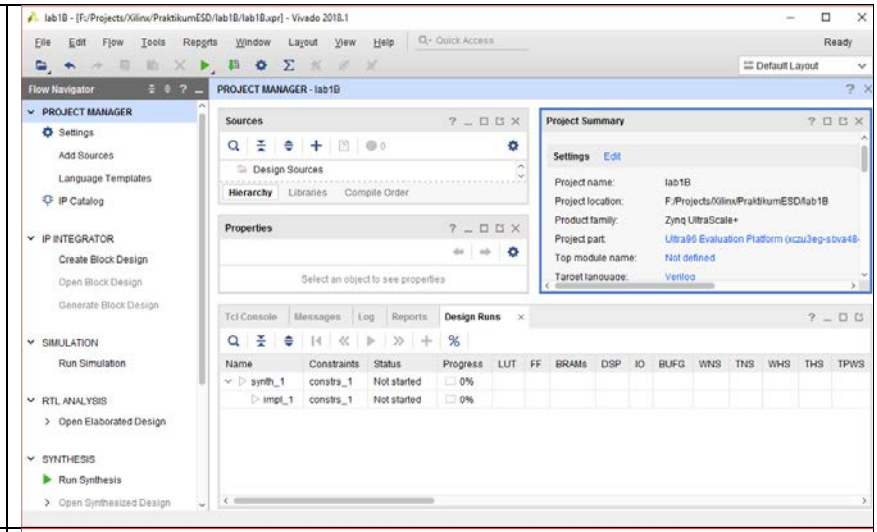


Once you have fixed the BSP configuration. The console should be able to capture the data sent from the board.

Now you can write a more sophisticated application to take inputs from the console and react to that as well.

Lab 1 – B

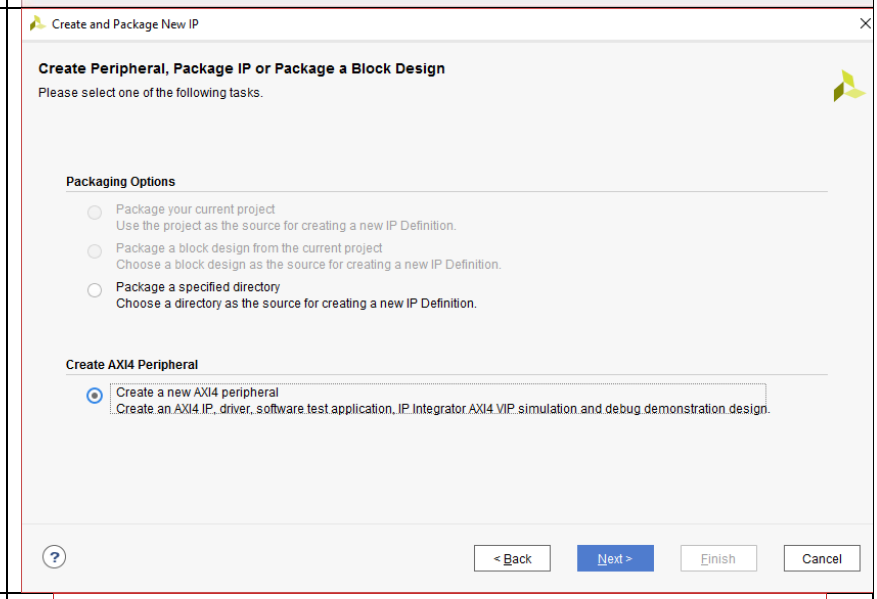
Create a new project in Vivado.



In this lab, a simple AXI-Lite co-processor (or hardware or peripheral or IP) will be created and connected to the ARM to do a simple addition operation.

Click "Tools → Create and Package New IP".

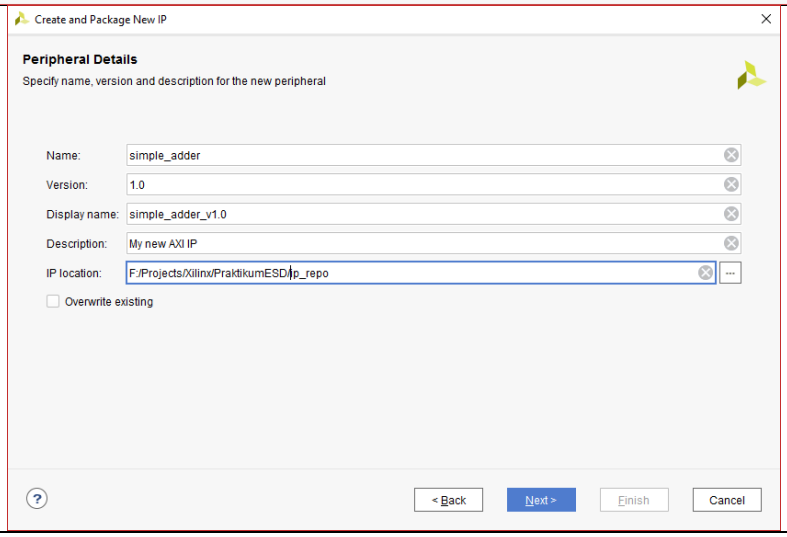
Click Next, select the option "Create a new AXI4 peripheral"

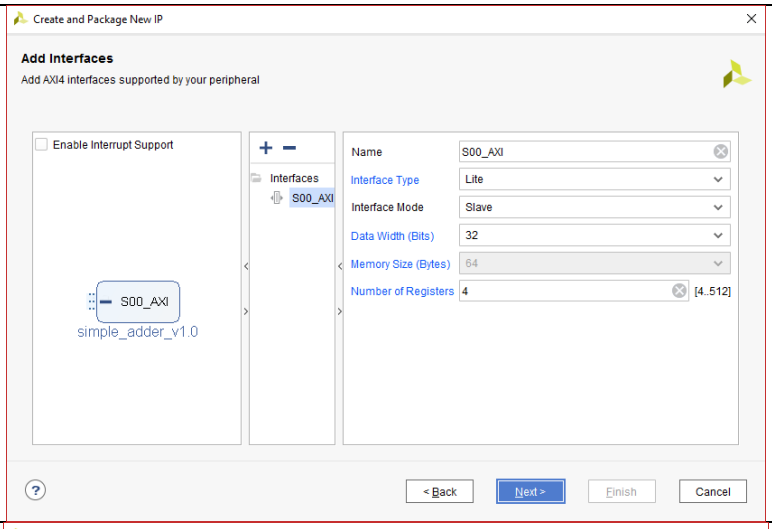
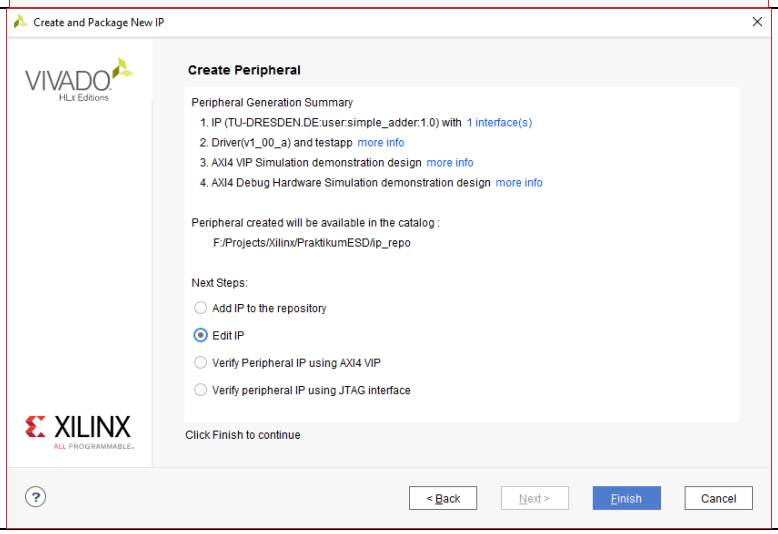


Choose the name as you like and the location where you want to store this IP.

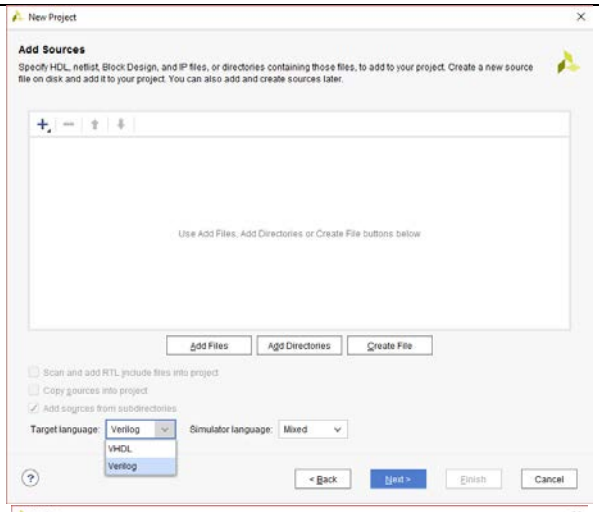
Since you may need to reuse the IP in the future, keep it outside of the project folder. It is easier to import the IPs inside the ip_repo folder into other projects.

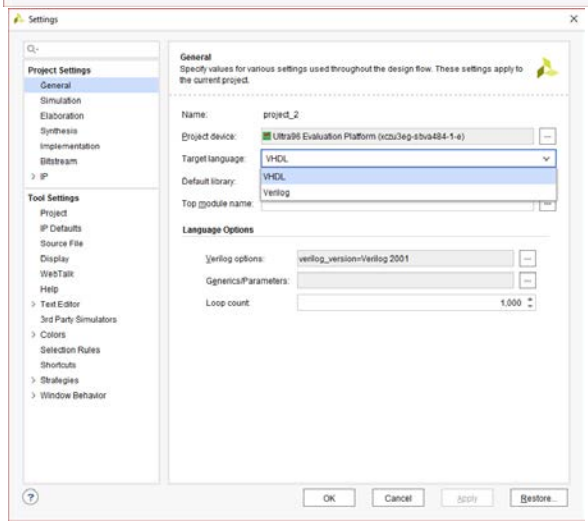
Click Next



Make sure that the interface configuration for the IP is similar to the figure. Question: What does “Interface mode” mean? What is “Slave” and “Master”? Click Next	
Select “Edit IP” as the next step because we want to edit the IP to take the command from the ARM and perform the addition.	

The code template generated by Vivado will be in VHDL or Verilog depending on the initial project configuration for the target language. It can also be changed in the Project Settings if it's already created. The settings can be found under the Project Manager inside the Flow Navigator.





In case of Verilog, the file simple_adder_v1_0_S00_AXI.v implements the actual logic used to process the write and read command from the ARM. Currently, it supports 4 registers that can be accessed by the ARM.

For AXI communication protocol, there are 2 basic channels: READ and WRITE. For each channel, there are two sub-channels: request and response. For the READ request, the response is the value of the indicated register. For the WRITE request, the response is the acknowledgement that the

```
289        if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready && S_AXI_WVALID)
290          begin
291            // indicates a valid write response is available
292            axi_bvalid <= 1'b1;
293            axi_bresp  <= 2'b0; // 'OKAY' response
294          end                   // work error responses in future
295        else
296          begin
297            if (S_AXI_BREADY && axi_bvalid)
298              //check if bready is asserted while bvalid is high)
299              //(there is a possibility that bready is always asserted high)
300              begin
301                axi_bvalid <= 1'b0;
302              end
303          end
```

request is processed. In the current implementation, the response is asserted immediately 1 clock cycle when the request is received. For more information about the AXI protocol, please visit [HERE](#).

The registers are accessed by ARM via the Address signal within each channel depending on the type of the request (read or write). Therefore, for AXI4-(Lite), it is called **Memory-mapped interface** to differentiate from the AXI-S which is a **Streaming** interface.

```
345     always @( posedge S_AXI_ACLK )
346     begin
347       if ( S_AXI_ARESETN == 1'b0 )
348         begin
349           axi_rvalid <= 0;
350           axi_rresp  <= 0;
351         end
352       else
353         begin
354           if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
355             begin
356               // Valid read data is available at the read data bus
357               axi_rvalid <= 1'b1;
358               axi_rresp  <= 2'b0; // 'OKAY' response
359             end
360           else if (axi_rvalid && S_AXI_RREADY)
361             begin
362               // Read data is accepted by the master
363               axi_rvalid <= 1'b0;
364             end
365         end
366     end
```

For example, for the write request, this block of code decodes the Address signal (axi_awaddr) then write the data to the corresponding register.

```
222     always @( posedge S_AXI_ACLK )
223     begin
224       if ( S_AXI_ARESETN == 1'b0 )
225         begin
226           slv_reg0 <= 0;
227           slv_reg1 <= 0;
228           slv_reg2 <= 0;
229           slv_reg3 <= 0;
230         end
231       else begin
232         if (slv_reg_wren)
233           begin
234             case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
235               2'h0:
236                 for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
237                   if ( S_AXI_WSTRB[byte_index] == 1 ) begin
238                     // Respective byte enables are asserted as per write strobes
239                     // Slave register 0
240                     slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
241                   end
242               2'h1:
243                 for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
```

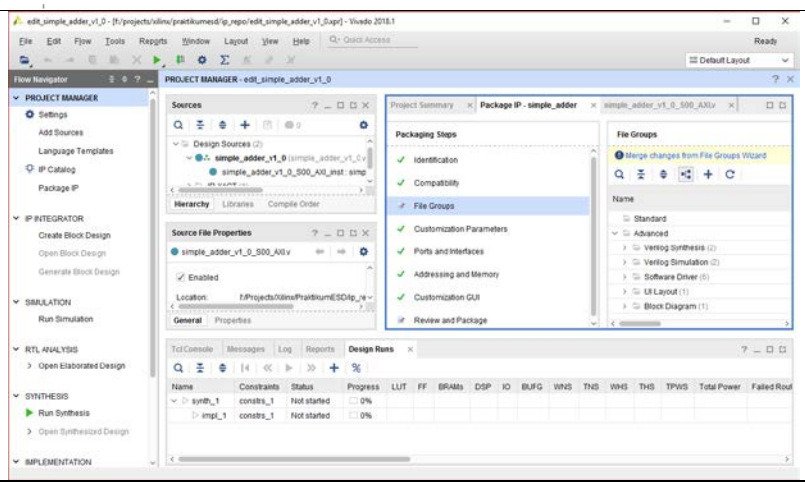
In this example, the adder only takes 2 inputs. Therefore, we only need to provide write access to 2 registers. Modify the file simple_adder_v1_0_S00_AXI.v as shown.

```
219     always @( posedge S_AXI_ACLK )
220     begin
221       if ( S_AXI_ARESETN == 1'b0 )
222         begin
223           slv_reg0 <= 0;
224           slv_reg1 <= 0;
225 //        slv_reg2 <= 0;
226 //        slv_reg3 <= 0;
227         end
228       else begin
229         if (slv_reg_wren)
230           begin
231             case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
232               2'h0:
233                 for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
234                   if ( S_AXI_WSTRB[byte_index] == 1 ) begin
235                     // Respective byte enables are asserted as per write strobes
236                     // Slave register 0
237                     slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
238                   end
239               2'h1:
240                 for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
241                   if ( S_AXI_WSTRB[byte_index] == 1 ) begin
242                     // Respective byte enables are asserted as per write strobes
243                     // Slave register 1
244                     slv_reg1[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
245                   end
246 //            2'h2:
247 //              for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
248 //                if ( S_AXI_WSTRB[byte_index] == 1 ) begin
249 //                  // Respective byte enables are asserted as per write strobes
250 //                  // Slave register 2
251 //                  slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
252 //                end
253 //            2'h3:
254 //              for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
255 //                if ( S_AXI_WSTRB[byte_index] == 1 ) begin
256 //                  // Respective byte enables are asserted as per write strobes
257 //                  // Slave register 3
258 //                  slv_reg3[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
259 //                end
260               default : begin
261                           slv_reg0 <= slv_reg0;
262                           slv_reg1 <= slv_reg1;
263 //                        slv_reg2 <= slv_reg2;
264 //                        slv_reg3 <= slv_reg3;
265                         end
266             endcase
267           end
268       end
```

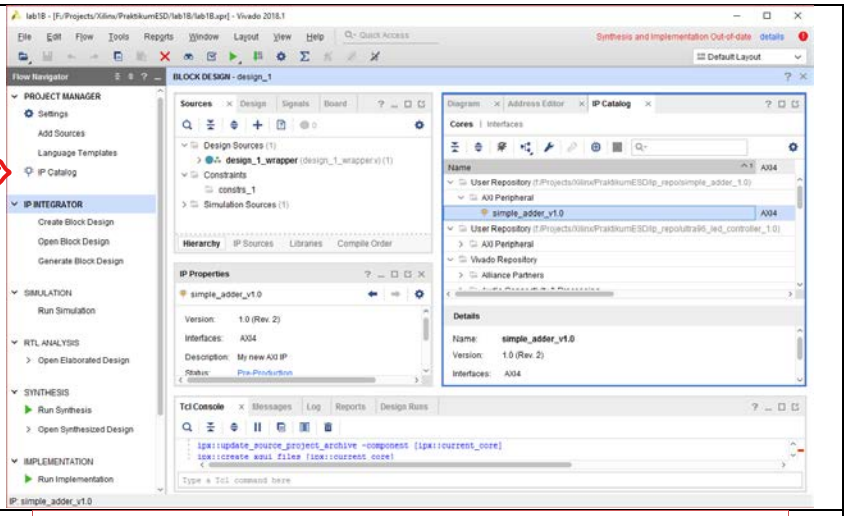
At the end of the file, add our own logic there to perform the addition.

```
401     // Add user logic here
402     //implement the Addition functionality on two registers, the result
403     //is written to the third register
404     always @(*) begin
405         slv_reg2 = $signed(slv_reg0) + $signed(slv_reg1);
406     end
407     // User logic ends
```

After that, click the "Package IP" under the Project Manager inside the Flow Navigator to open the Package IP tab. The changes you made to the files will be detected here and it asks for you to merge it into the IP description. Go into each Packaging Steps to review and merge the changes.

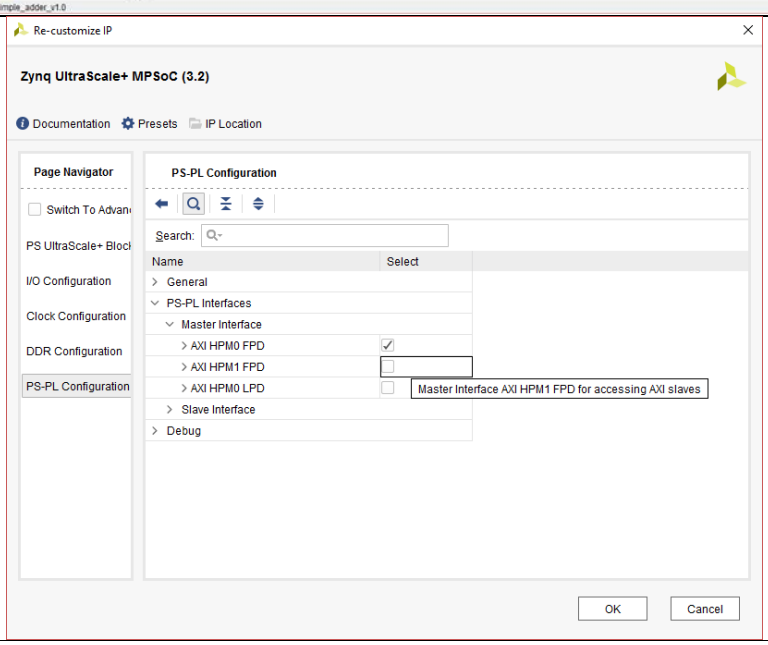


Get back to the original project.
Click the IP Catalog under the Project Manager to make sure that the user-defined IP repository is already added to the Project Repo.



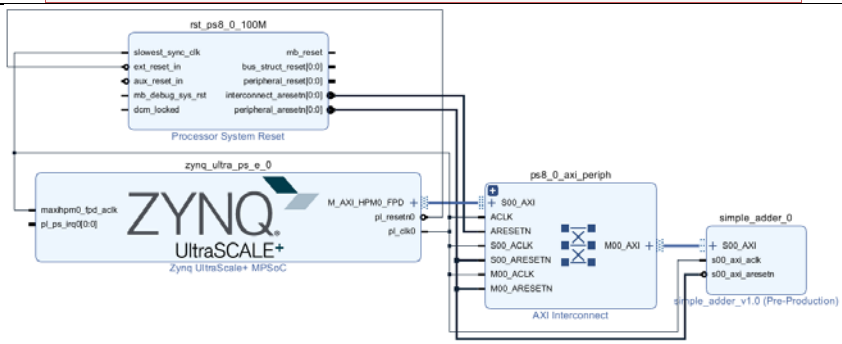
Now create a block design to add the Zynq Subsystem similarly to Lab 1.A.

After that, double click on the Zynq UltraScale block to configure it. We only need one AXI master port from the Zynq PS. Uncheck the AXI HPM1 FPD port. Click OK.

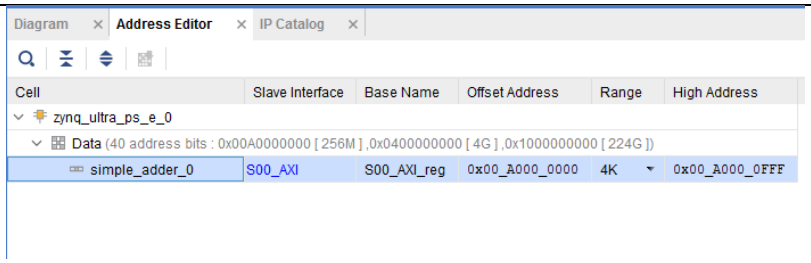


Now back to the Diagram tab, add the simple_adder IP, then "Run Connection Automation" to connect the S00_AXI interface of the IP to the Zynq. You should be able to see the resulting blocks and connections like this.

Question: what are the new blocks that are added by Vivado? What do they do?



Open the Address Editor tab, it can be seen that the IP is assigned an address that can be accessed by the ARM. We don't need to remember this address because it will be stored as a macro when a BSP is created within the Xilinx SDK.

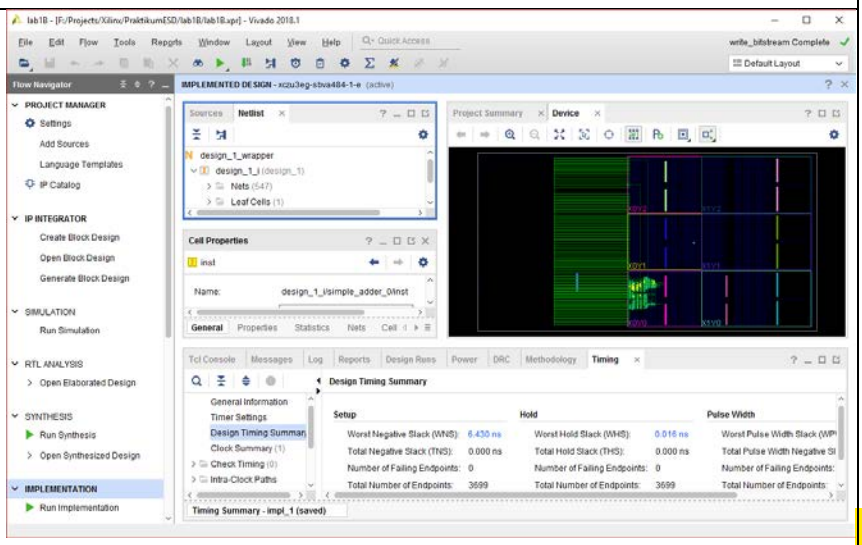


Click Generate Bitstream. Once it's finished, it asks to open the Implemented Design, Click OK to open it.

You can see where these components are placed on the FPGA inside the Device tab.

Inside the Netlist tab, right click on any component → Highlight Leaf Cells → choose color. Then the respective sub-components will be highlighted.

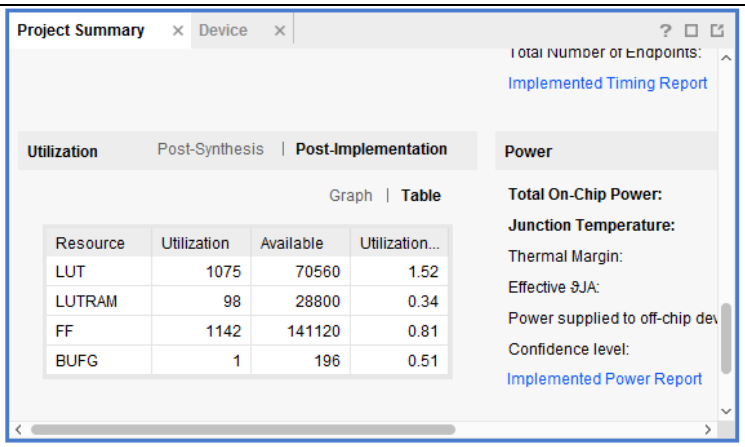
Question: In the Timing tab, you can see the timing-related reports. What do they mean?

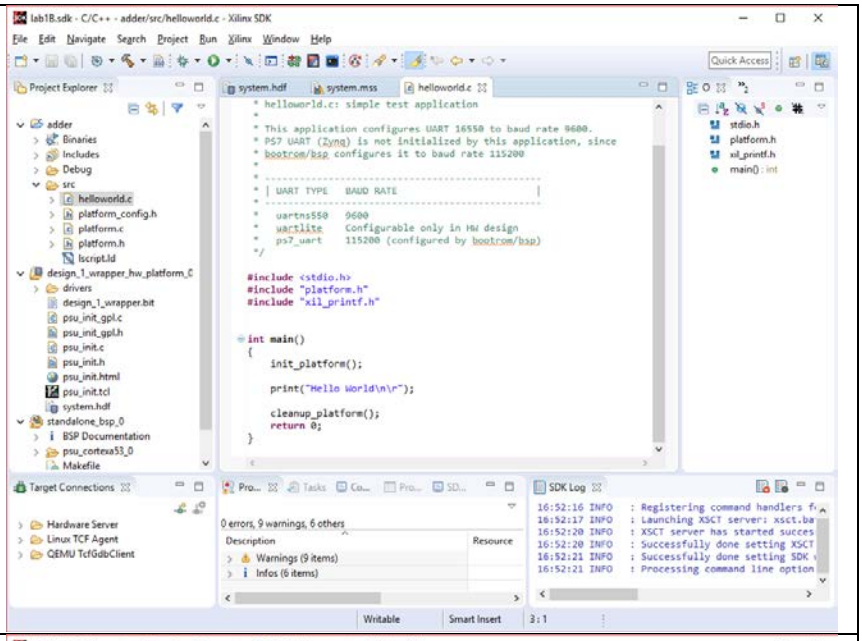
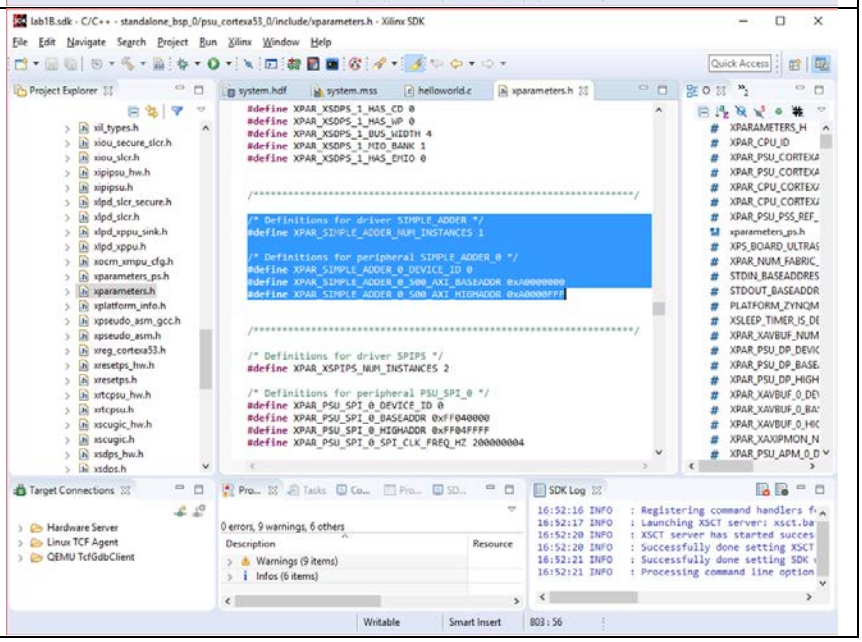


不一样

In the Project Summary tab, if you click on the Table option within the Post-Implementation Utilization, the overall resources utilization of the design will be displayed.

Question: what are "post-synthesis" and "post-implementation"?
Question: how do you find the resource utilization of every individual module?



Now, export the hardware + bitstream then launch the SDK. Create the BSP and an application based on the Hello World template.	
In the Project Explorer, expand to the folder psu_cortexa53_0/include of the BSP that you have just created. Look for the file xparameters.h You can see in the file that there are several macros defined for the ADDER hardware.	

Modify the file helloworld.c as shown then run the application to see the results.

Question: why can we access the registers by using the pointers like that?

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

#define HW_ADDER_REG_A XPAR_SIMPLE_ADDER_0_S00_AXI_BASEADDR
#define HW_ADDER_REG_B XPAR_SIMPLE_ADDER_0_S00_AXI_BASEADDR + 4
#define HW_ADDER_REG_C XPAR_SIMPLE_ADDER_0_S00_AXI_BASEADDR + 8


int main()
{
    int *a;
    int *b;
    int *c;
    a = (UINTPTR) HW_ADDER_REG_A;
    b = (UINTPTR) HW_ADDER_REG_B;
    c = (UINTPTR) HW_ADDER_REG_C;

    int i;
    init_platform();

    print("This program test the simple AXI-Lite adder hardware implementation on the FPGA\n\r");

    for (i = -5; i < 5; i++){
        *a = i;
        *b = 2 * i;
        printf("a = %d, b = %d, c = %d\n\r", *a, *b, *c);
    }

    cleanup_platform();
    return 0;
}
```

The results should look like this

```
COM9 - PuTTY
This program test the simple AXI-Lite adder hardware implementation on the FPGA
a = -5, b = -10, c = -15
a = -4, b = -8, c = -12
a = -3, b = -6, c = -9
a = -2, b = -4, c = -6
a = -1, b = -2, c = -3
a = 0, b = 0, c = 0
a = 1, b = 2, c = 3
a = 2, b = 4, c = 6
a = 3, b = 6, c = 9
a = 4, b = 8, c = 12
```

Lab 1 – Homework 1

Extend the above HW adder as an ALU to support addition, subtraction, multiplication and power functions. You should have one Status/Command register with at least 2 fields:

- 1 read/write field called OP used to set the desired operation,
- 1 read-only field called DONE to check the status of the operation (the power function may take several clock cycles to complete depending on how you implement it)

The other two read/write registers are used as inputs. The last one, read-only, is used as output result.

The ARM first set the desired operation, write the inputs, wait until DONE is 1, then read the result.

Lab 1 – Homework 2

Extend the Homework 1 to support vector operations. There will be two input vectors. Each vector has one corresponding register. The elements of each vector are written one by one to its register.

For example, REGISTER_1 is used for VECTOR_A, REGISTER_2 is used for VECTOR_B.

The elements of VECTOR_A are transferred to the hardware from ARM as such:

For (I = 0; I < size of vector; i++):

Write VECTOR_A[I] to REGISTER_1

In this case, when designing the hardware, consider REGISTER_1 as a point of entry. You should maintain your own counter for VECTOR_A to count the number of elements and to address into the memory on the FPGA to store the element. For example, you should do something like this in your hardware:

If write to REGISTER_1:

VECTOR_A_mem[counter_A] = REGISTER_1

counter_A = counter_A + 1

The Status/Command register is used by ARM to set the number of elements, the desired operation and to check the DONE status. Whenever there is a write to this register, you reset the counter_A to 0 to wait for the data for VECTOR_A. Similarly for VECTOR_B. When both counters reach the desired number of elements set by the ARM initially, start the computation.

?

You should write a separate FSM to do the computation.

You should use a dual port BRAM on the FPGA to store the data for each input vector. One port is used to write the data from ARM. The other one is used to read to compute. Similarly for the result vector, one port is used to write the result to, the other one is used to read to return to ARM.