# System Design Flow on Zynq using Vivado Workshop
# ZYBO

## COURSE DESCRIPTION

This course provides professors necessary skills to design and debug a system using Vivado IP Integrator, hardware analyzer, and Vivado HLS.

### 1. Install Xilinx software

Professors may submit the online donation request form at http://www.xilinx.com/member/xup/donation/request.htm to obtain the latest Xilinx software. The workshop was tested on a PC running Microsoft Windows 7 professional edition.

- Vivado 2013.4 System Edition

### 2. Setup hardware

Connect ZYBO

a. Set the power supply jumper to USB so the board can be powered up and laboratory assignments can be carried out using single micro-usb cable
b. Connect micro USB cable between PROG UART port of ZYBO and PC

### 3. Install distribution

Extract the labsource.zip file in c:\xup\sys_design directory. This will generate **sources** and **labs** folders. The labdocs.zip file consists of lab documents in the PDF format. Extract this zip file in c:\xup\sys_design\ directory or any directory of your choice.

### 4. For Professors only

Download the labsolution.zip and docs_source.zip files using your membership account. Do not distribute them to students or post them on a web site. The docs_source.zip file contains lab documents in Microsoft Word and presentations in PowerPoint format for you to use in your classroom.

### 5. Get Started

Review the presentation slides (see course agenda) and step through the lab exercises (see lab descriptions) to complete the labs.

**COURSE AGENDA**

| Day 1 Agenda | Day 1 Materials |
|---|---|
| Class Intro | 01_class_intro.pptx |
| 7 Series Architecture Overview | 11_7_Series_Architecture_Overview.ppt x |
| Vivado Design Flow | 12_Vivado Design_Flow.pptx |
| Lab 1: Synthesizing a RTL Design | 11a_lab1_intro.pptx<br>Lab01.docx |
| Xilinx Design Constraints | 13_Xilinx_Design_Constraints.pptx |
| Lab 2: Xilinx Design Constraints | 13a_lab2_intro.pptx<br>Lab02.docx |
| IP Integrator and Embedded System Design Flow | 14_IPI_And_Embedded_System_Design.pptx |
| Lab 3: Create a Processor System using IP Integrator | 14a_lab3_intro.pptx<br>Lab03.docx |
| **Day 2 Agenda** | **Day 2 Materials** |
| Creating and Adding Custom IP | 21_Creating_and_Adding_Custom_IP.pptx |
| Lab 4: Creating and Adding Custom IP in PL | 21a_lab4_intro.pptx<br>Lab04.docx |
| System Debugging | 22_System_Debugging.pptx |
| Lab 5: System Debugging using Vivado Logic Analyzer and SDK | 22a_lab5_into.pptx<br>Lab05.docx |
| Profiling and Performance Improvement | 23_Profiling_and_Performance_Improvement.pptx |
| Introduction to High-Level Synthesis with Vivado HLS | 24_Vivado_HLS_Intro.pptx |
| Improving Performance and Resource Utilization | 25_Improving_Performance_and_Resource_Utiliza tion |
| Creating an Accelerator | 26_Creating_an_accelerator.pptx |
| Lab 6: Creating a Processor System | 26a_lab6_into.pptx<br>Lab06.docx |

**LAB DESCRIPTIONS**

Lab 1 - Use Vivado IDE to create a simple HDL design. Simulate the design using the XSim HDL simulator available in Vivado design suite. Generate the bitstream and verify in hardware.
Lab 2 - Create a project with I/O Planning type, enter pin locations, and export it to the RTL. Then create the timing constraints and perform the timing analysis.
Lab 3 – Create a simple ARM Cortex-A9 based processor design targeting the ZYBO using IP Integrator.
Lab 4 - Use the Manage IP feature of Vivado to create a custom IP and extend the system with the custom peripheral. Write a basic C application to access the peripherals.
Lab 5 - Insert various Vivado Logic Analyzer cores to debug/analyze system behavior.
Lab 6 - Profile an application performing a function both in software and hardware. Create an accelerator in Vivado HLS. Use the generated accelerator to build a complete system.

## 6. Contact XUP

Send an email to xup@xilinx.com for questions or comments

# Synthesizing a RTL Design

## Introduction

This lab shows you the synthesis process and the effect of changing of the synthesis settings. You will analyze the design and the generated reports.

## Objectives

After completing this lab, you will be able to:

- Create a Vivado project sourcing HDL model(s) and targeting the ZYBO
- Use the provided Xilinx Design Constraint (XDC) file to constrain the pin locations
- Simulate the design using the Vivado simulator
- Synthesize and implement the design
- Generate the bitstream
- Configure the FPGA using the generated bitstream and verify the functionality

## Procedure

This lab is broken into steps that consist of general overview statements providing information on the detailed instructions that follow.

## Design Description

The design consists of a uart receiver receiving the input typed on a keyboard and displaying the least significant 4 bits of binary equivalent of the typed character on the 4 LEDs. When a push button is pressed, the upper bits of the binary equivalent are displayed. The block diagram is as shown in **Figure 1**.

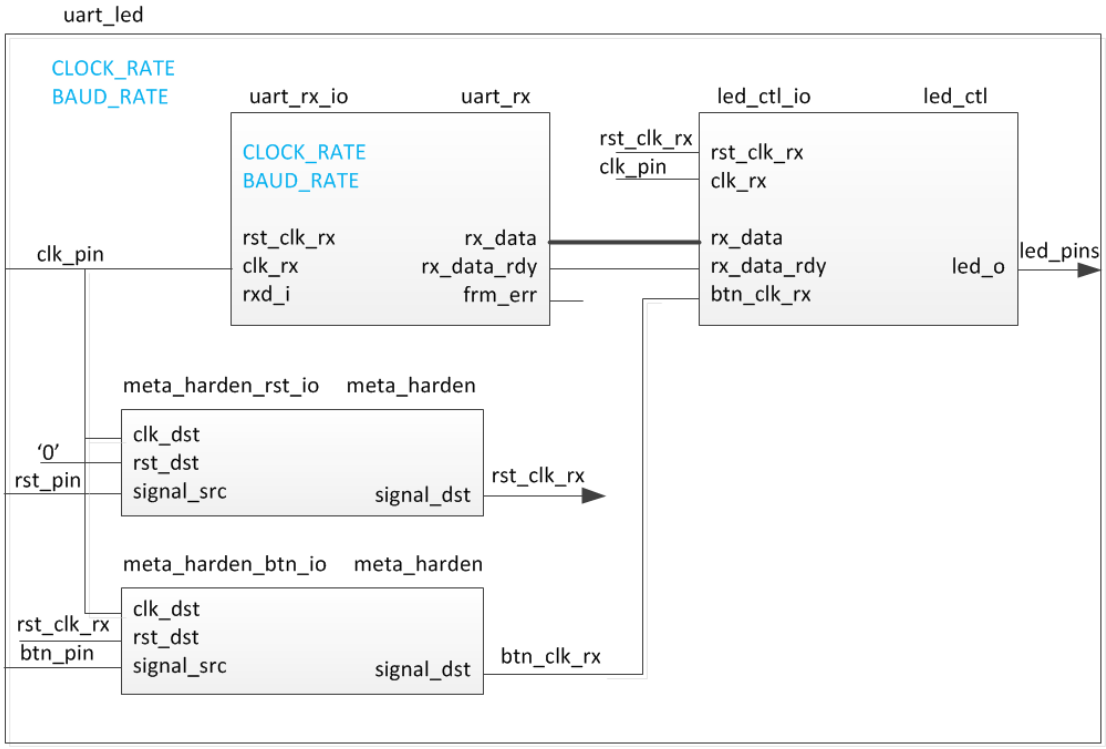
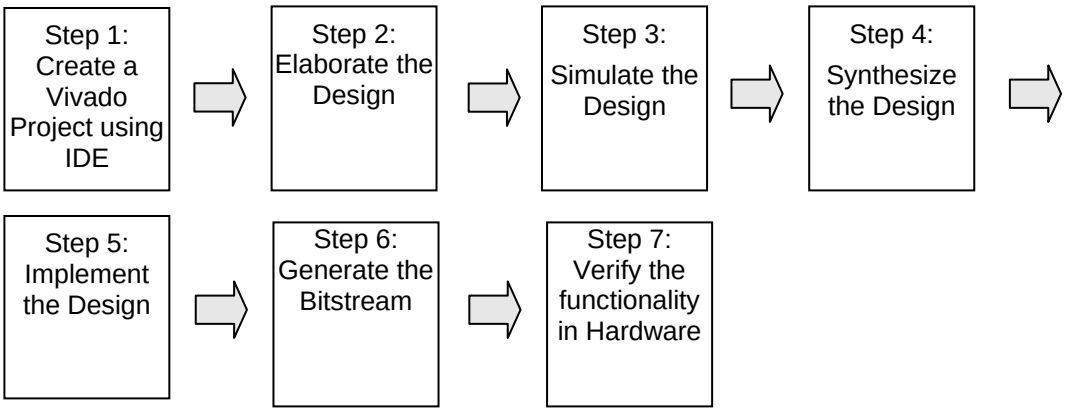



**Figure 1. The Completed Design**

## General Flow




## Create a Vivado Project using IDE — Step 1

### 1-1. Launch Vivado and create a project targeting the XC7Z010CLG400-1 device and using the Verilog HDL. Use the provided Verilog source files and uart_led_timing.xdc file from the *sources\lab1* directory.

**1-1-1.** Open Vivado by selecting **Start > All Programs > Xilinx Design Tools > Vivado 2013.4 > Vivado 2013.4**

**1-1-2.** Click **Create New Project** to start the wizard. You will see *Create A New Vivado Project* dialog box. Click **Next**.

**1-1-3.** Click the Browse button of the *Project location* field of the **New Project** form, browse to **c:\xup\sys_design\labs**, and click **Select**.

**1-1-4.** Enter **lab1** in the *Project name* field. Make sure that the *Create Project Subdirectory* box is checked. Click **Next**.

**1-1-5.** Select **RTL Project** option in the *Project Type* form, and click **Next**.

**1-1-6.** Select **Verilog** as the *Target Language* and the *Simulator language* in the *Add Sources* form.

**1-1-7.** Click on the **Add Files…** button, browse to the **c:\xup\sys_design\sources\lab1** directory, select *led_ctl.v, meta_harden.v, uart_baud_gen.v, uart_led.v, uart_rx.v,* and *uart_rx_ctl.v* files (do not include Verilog files whose name start with *test_* or *tb_),* click **OK**, and then click **Next**.

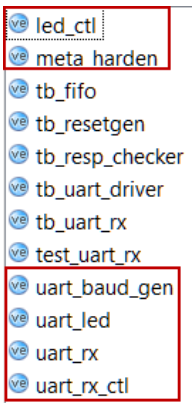



**Figure 2. Source files to be added**

**1-1-8.** Click **Next** to get to the *Add Cons*traints form.

**1-1-9.** You will see uart_led_timing.xdc and uart_led_pins.xdc have automatically been included. Remove the uart_led_pins.xdc, for now, by selecting the entry and clicking on the X on the side. (If you don't see uart_led_timing.xdc entry then browse to the **c:\xup\sys_design\sources\lab1** directory, select *uart_led_timing.xdc* and click **Open**.

**1-1-10.** Click **Next.**

This Xilinx Design Constraints file assigns the basic timing constraints (period, input delay, and output delay) to the design.

**1-1-11.** In the *Default Part* form, using the **Parts** option and various drop-down fields of the **Filter** section, select the *XC7Z010CLG400-1* part. Click **Next**.

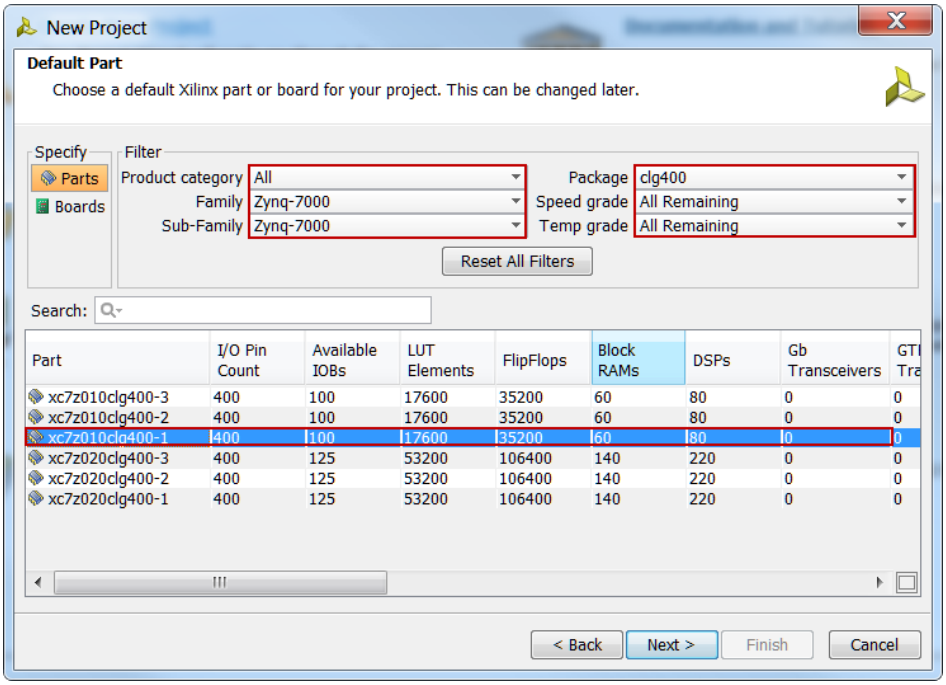



**Figure 3. Selecting the target board device**

**1-1-12.** Click **Finish** to create the Vivado project.

## 1-2. Analyze the design source files hierarchy.

**1-2-1.** In the *Sources* pane, expand the **uart_led** entry and notice hierarchy of the lower-level modules.

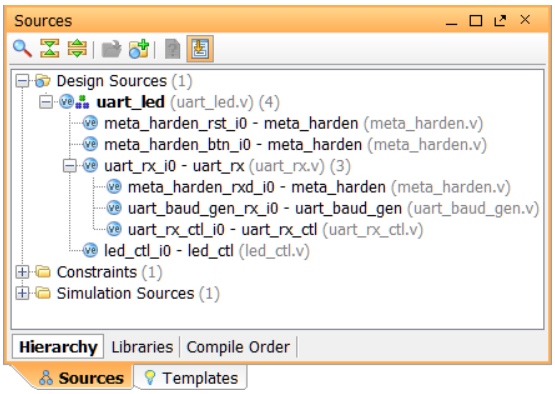



**Figure 4. Opening the source file**

**1-2-2.** Double-click on the **uart_led** entry to view its content.

Notice in the Verilog code, the BAUD_RATE and CLOCK_RATE parameters are defined to be 115200 and 125 MHz respectively as shown in the design diagram (Figure 1). Also notice that the lower level modules are instantiated. The meta_harden modules are used to synchronize the asynchronous reset and push-button inputs.

**1-2-3.** Expand **uart_rx_i0** instance to see its hierarchy.

It uses the baud rate generator and the finite state machine. The rxd_pin is sampled at x16 the baud rate.

## 1-3. Open the uart_led_timing.xdc source and analyze the content.

**1-3-1.** In the *Sources* pane, expand the *Constraints* folder and double-click the **uart_led_timing.xdc** entry to open the file in text mode.

```
1 # ZYBO xdc
2 # define clock and period
3 create_clock -period 8.000 -name clk_pin -waveform {0.000 4.000} [get_ports clk_pin]
4
5 # input delay
6 set_input_delay -clock clk_pin 0 [get_ports rxd_pin]
7 set_input_delay -clock clk_pin -min -0.5 [get_ports rxd_pin]
8
9 set_input_delay -clock clk_pin -max 0.0 [get_ports btn_pin]
10 set_input_delay -clock clk_pin -min -0.5 [get_ports btn_pin]
11
12 #output delay
13 set_output_delay -clock clk_pin 0 [get_ports led_pins[*]]
14
15 set_property IOB TRUE [all_fanin -only_cells -startpoints_only -flat [all_outputs]]
```

**Figure 5. Timing constraints**

Line 3 creates the period constraint of 8 ns with a duty cycle of 50%. The rxd_pin (lines 6 and 7), the btn_pin (lines 9, 10), and the led_pins (line 13) are constrained with respect to the design clock. Line 15 instructs the implementation tool to place the input/output registers in IOB wherever possible.

# Elaborate the Design Step 2

## 2-1. Elaborate and perform the RTL analysis on the source file.

**2-1-1.** Expand the *Open Elaborated Design* entry under the *RTL Analysis* tasks of the *Flow Navigator* pane and click on **Schematic**.

The model (design) will be elaborated and a logic view of the design is displayed.

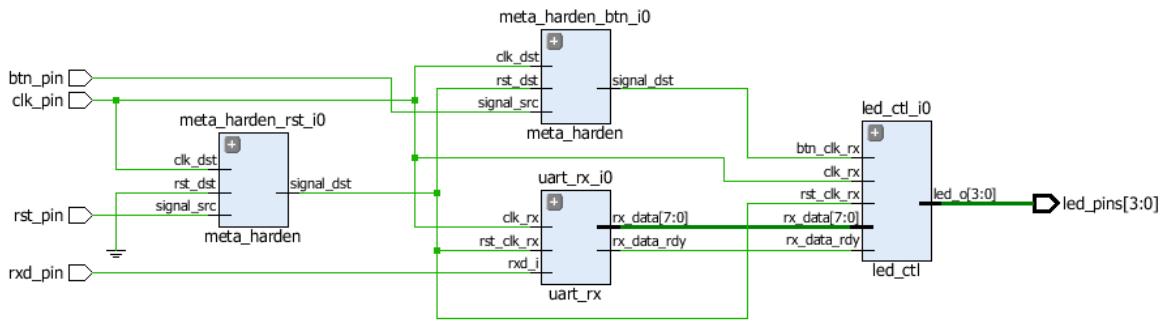



**Figure 6. A logic view of the design**

You will see four components at the top-level, 2 instances of meta_harden, one instance of uart_rx, and one instance of led_ctl.

**2-1-2.** To see where the uart_rx_i0 gets generated, right-click on the uart_rx_i0 instance and select **Go To Source** and see that line 84 in the source code is generating it.

**2-1-3.** Double-click on the uart_rx_i0 instance in the schematic diagram to see the underlying components.

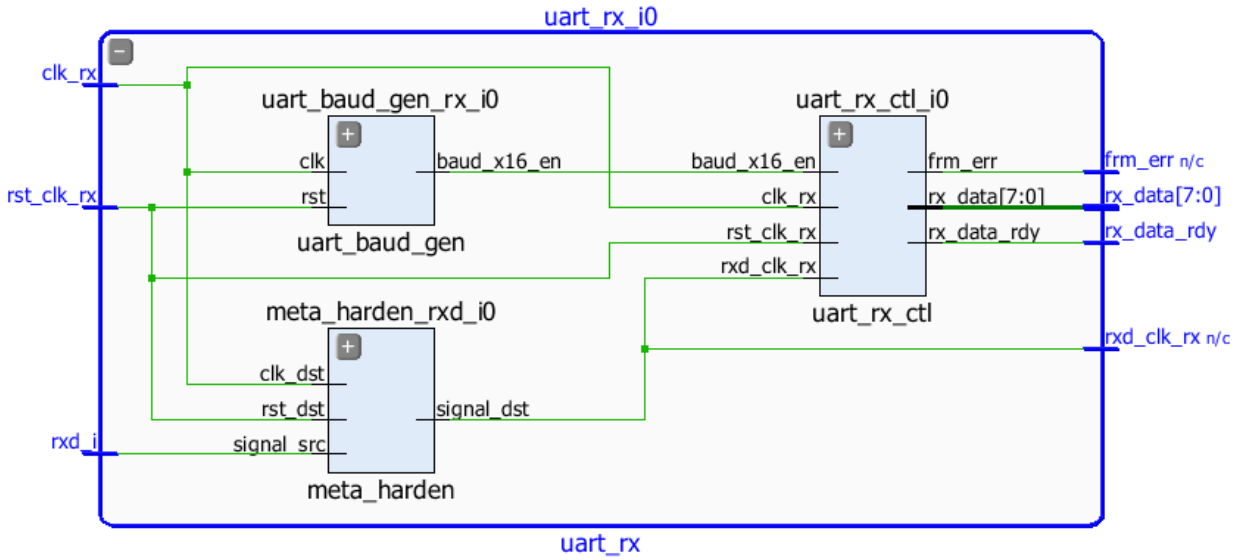



**Figure 7. Lower level components of the uart_rx_i0 module**

**2-1-4.** Click on **Report Noise** under the *Open Elaborated Design* entry of the *RTL Analysis* tasks of the *Flow Navigator* pane.

**2-1-5.** Click **OK** to generate the report named **ssn_1**.

**2-1-6.** View the ssn_1 report and observe the unplaced ports, Summary, and I/O Bank Details are highlighted in red because the pin assignments were not done. Note that only output pins are reported as the noise analysis is done on the output pins.

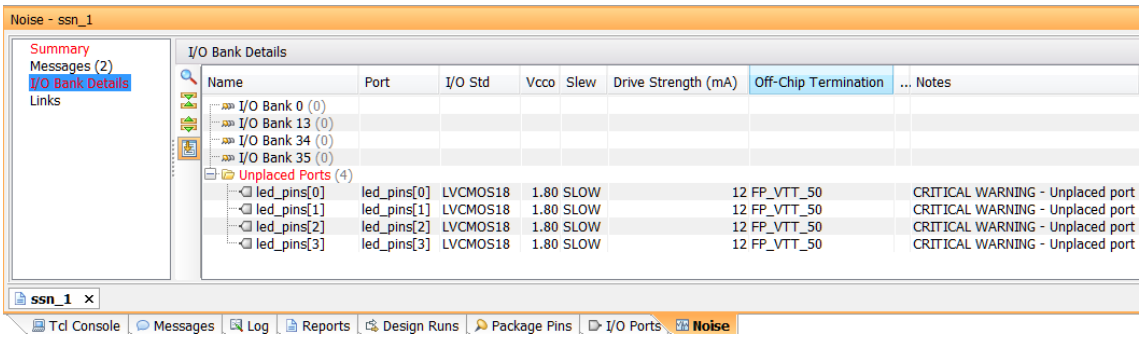
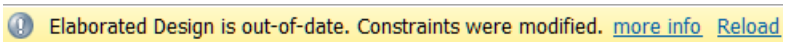



**Figure 8. Noise report**

**2-1-7.** Click on **Add Sources** under the *Project Manager*, select the *Add or Create Constraints* option, and click **Next**.

**2-1-8.** Click on the **Add Files…** button, browse to the **c:\xup\sys_design\sources\lab1** directory, select **uart_led_pins.xdc** file, click **OK**, and then click **Finish** to add the pins location constraints.

Notice that the sources are modified and the tools detected it, showing a warning status bar to re-load the design.




**2-1-9.** Click on the **Reload** link.

## Simulate the Design using the Vivado XSim Simulator Step 3

### 3-1. Add the provided testbench files from the c:\xup\sys_design\sources\lab1 directory.

**3-1-1.** Click **Add Sources** under the *Project Manager* tasks of the *Flow Navigator* pane.

**3-1-2.** Select the *Add or Create Simulation Sources* option and click **Next**.

**3-1-3.** In the *Add Sources Files* form, click the **Add Files…** button.

**3-1-4.** Browse to the **c:\xup\fpga_flow\labs\sources\lab1** folder and select five files whose name starts with *tb_* and another file whose name starts with *test_*, and click **OK**.

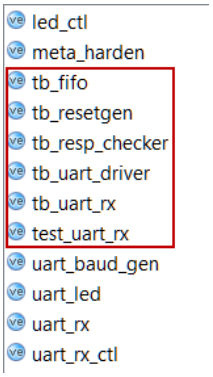



**Figure 9. Testbench files being selected and added**

**3-1-5.** Click **Finish**.

**3-1-6.** Select the *Sources* tab and expand the *Simulation Sources* group.

Notice the hierarchy created by the added testbenches.

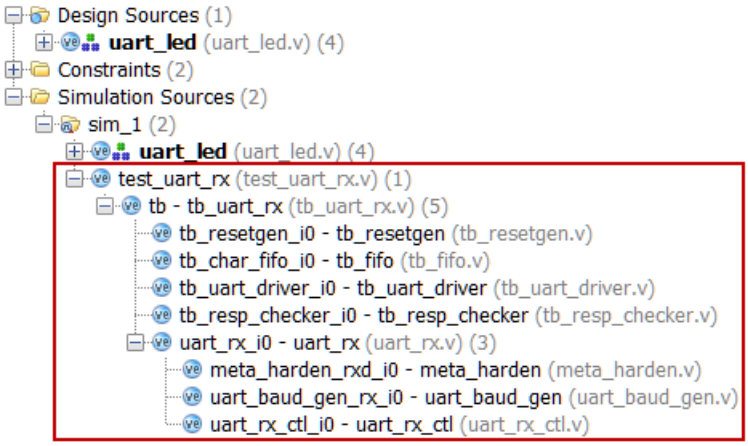



**Figure 10. Simulation Sources hierarchy**

**3-1-7.** Select the *test_uart_rx* entry, right-click, and select **Set as Top**.

## 3-2. Simulate the design using the Vivado XSim simulator.

**3-2-1.** Click on **Run Simulation > Run Behavioral Simulation** under the *Project Manager* tasks of the *Flow Navigator* pane.

The testbench and source files will be compiled and the Vivado simulator will be run (assuming no errors). By default, an Untitled Waveform viewer will appear displaying only the signals at the top level of the testbench. You will see a simulator output similar to the one shown below.

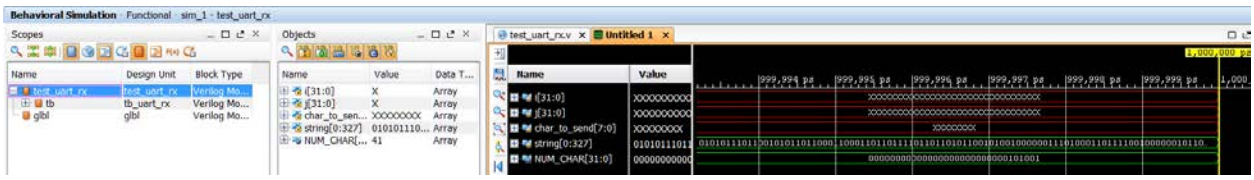


**Figure 11. Simulator output**

You will see four main views: (i) *Scopes,* where the testbench hierarchy as well as glbl instances are displayed, (ii) *Objects,* where top-level signals are displayed, (iii) the waveform window, and (iv) *Tcl Console* where the simulation activities are displayed.

You will see several buttons next to the waveform window which can be used for the specific purpose as listed in the table below.

**Table 1. Various buttons available to view the waveform**

| | |
|---|---|
|  | Waveform options<br>Save the waveform<br>Zoom In<br>Zoom Out<br>Zoom Fit<br>Zoom to cursor<br>Go to Time 0<br>Go to Last Time<br>Previous Transition<br>Next Transition<br>Add Marker<br>Previous Marker<br>Next Marker<br>Swap Cursors<br>Snap to Transition |

**3-2-2.** Click on the *Zoom Fit* button (  ) to see the entire waveform.

## 3-3. Change display format if desired. Add more signals to monitor the lower-level signals.

**3-3-1.** Select **char_to_send** in the waveform window, right-click, select *Radix*, and then select *ASCII* to view it in the *ASCII* form. Similarly, change the radix of **strings** to *ASCII*.

**3-3-2.** Expand the **tb** and **uart_rx_io** instances, if necessary, in the *Scopes* window and select the **uart_rx_io** instance. The corresponding signals will be displayed in the *Objects* window.

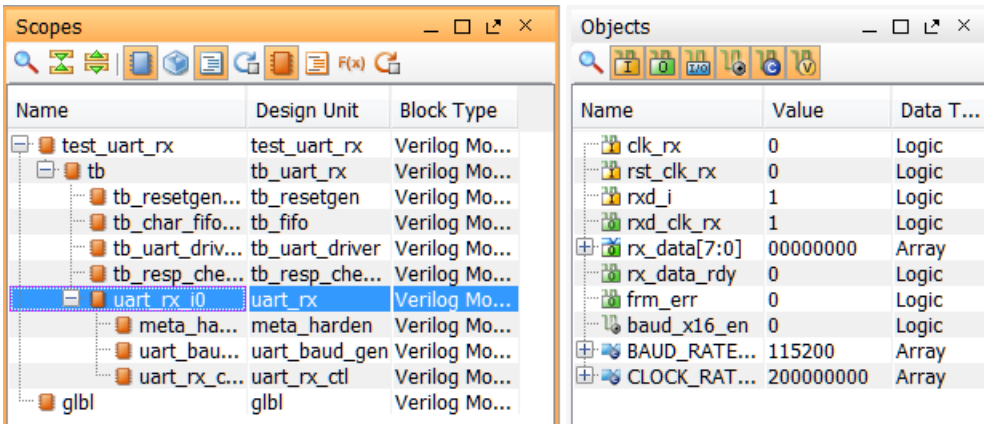



**Figure 12. Selecting lower-level signals**

**3-3-3.** In the *Objects* window, select **clk_rx**, and while pressing the **<Shift>** key, click **baud_x16_en** to select all of the signals at this level of the hierarchy, right-click, and select **Add to Wave Window**.

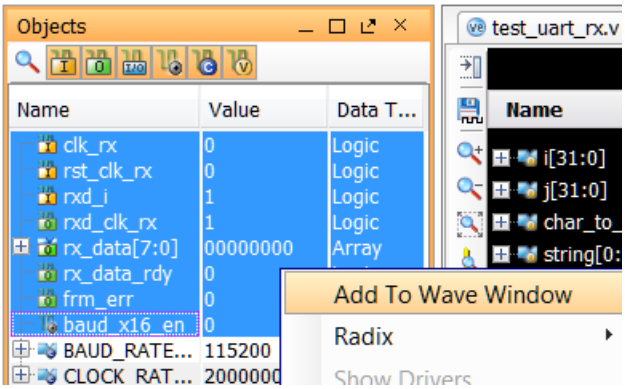



**Figure 13. Adding additional signals to the wave window**

The signals are added to the bottom of the waveform window, but no values are displayed because the simulator only saves signal values for signals that are displayed when the simulation is run. The simulation must be restarted for the values on the new signals to be seen.

**3-3-4.** In the waveform window, right-click **rx_data[7:0]** and select **Radix** > **ASCII.**

**3-3-5.** Click the **Restart** toolbar button ( ).

**3-3-6.** Click the **Run All** toolbar button ( ) to rerun the simulation.

## 3-4. Analyze the simulator output.

**3-4-1.** Scroll through the Tcl Console to ensure that the sending character and character received are same.

```
run all
      10.00 ns       Starting simulation
      10.00 ns       Asserting reset for         20 clocks
     107.50 ns       Deasserting reset
  500107.50 ns       Sending character 57 (W)
  574538.50 ns       Character received 57 (W)
  606917.50 ns       Sending character 65 (e)
  681358.50 ns       Character received 65 (e)
  713727.50 ns       Sending character 6c (l)
  788178.50 ns       Character received 6c (l)
  820537.50 ns       Sending character 63 (c)
  894998.50 ns       Character received 63 (c)
  927347.50 ns       Sending character 6f (o)
```

**Figure 14. Tcl console output**

**3-4-2.** Zoom in to the waveform window to visually inspect the received characters.

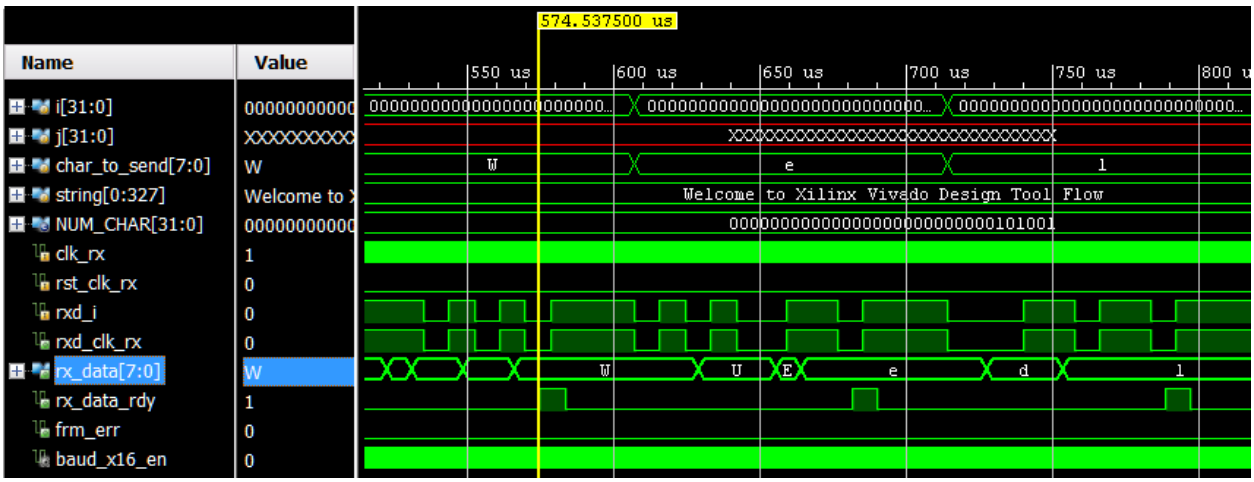



**Figure 15. Waveform output showing characters sent and received**

**3-4-3.** Close the simulator by selecting **File > Close Simulation**.

**3-4-4.** Click **OK** and then click **No** to close it without saving the waveform.

## Synthesize the Design — Step 4

### 4-1. Synthesize the design with the Vivado synthesis tool and analyze the Project Summary output.

**4-1-1.** Click on **Run Synthesis** under the *Synthesis* tasks of the *Flow Navigator* pane.

The synthesis process will be run on the uart_led.v and all its hierarchical files. When the process is completed a *Synthesis Completed* dialog box with three options will be displayed.

**4-1-2.** Select the *Open Synthesized Design* option and click **OK** as we want to look at the synthesis output.

Click **Yes** to close the elaborated design if the dialog box is displayed.

**4-1-3.** Select the **Project Summary** tab

If you don't see the Project Summary tab then select **Layout > Default Layout, or** click the **Project Summary** icon  .

**4-1-4.** Click on the **Table** tab in the **Project Summary** tab and fill out the following information.

### Question 1

Look through the table and find the number used of each of the following:

FF: _____
LUT: _____
I/O: _____
BUFG: _____

**4-1-5.** Click on **Schematic** under the *Synthesized Design* tasks of *Synthesis* tasks of the *Flow Navigator* pane to view the synthesized design in a schematic view.

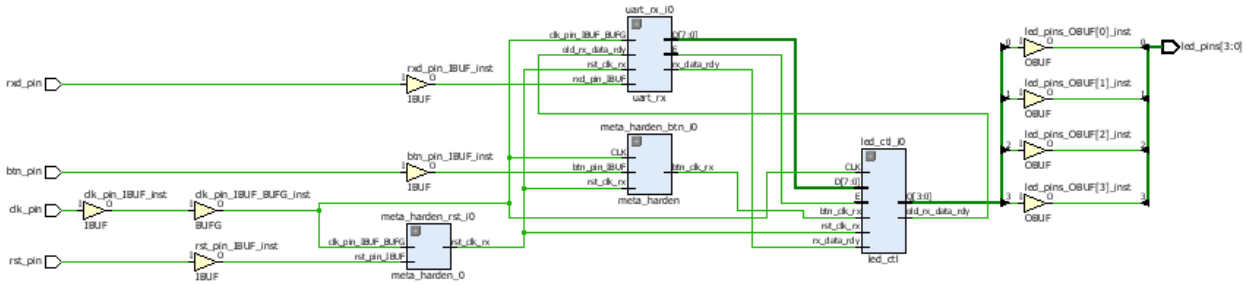



**Figure 16. Synthesized design's schematic view**

Notice that IBUF and OBUF are automatically instantiated (added) to the design as the input and output are buffered. There are still four lower level modules instantiated.

**4-1-6.** Double-click on the **uart_rx_i0** instance in the schematic view to see the underlying instances.

**4-1-7.** Select the **uart_baud_gen_rx_i0** instance, right-click, and select *Go To Source*.

Notice that line 92 is highlighted. Also notice that the CLOCK_RATE and BAUD_RATE parameters are passed to the module being called.

**4-1-8.** Double-click on the **meta_harden_rxd_io** instance to see how the synchronization circuit is being implemented using two FFs. This synchronization is necessary to reduce the likelihood of meta-stability.

**4-1-9.** Click on the ( ) in the schematic view to go back to its parent block.

## Implement the Design — Step 5

### 5-1. Run the implementation.

**5-1-1.** Click on the **Run Implementation** in the *Flow Navigator* pane.

**5-1-2.** Click **Yes** to close the synthesized design and run the implementation process.

When the implementation is completed, a dialog box will appear with three options.

**5-1-3.** Select the *Open Implemented Design* option and click **OK**.

### 5-2. View the amount of FPGA resources consumed by the design using Report Utilization.

**5-2-1.** In the *Flow Navigator* pane, select **Implemented Design** > **Report Utilization**.

The Report Utilization dialog box opens.

**5-2-2.** Click **OK**.

The utilization report is displayed at the bottom of the Vivado IDE. You can select any of the resources on the left to view its corresponding utilization.

**5-2-3.** Click Chart (rather than table) and select one of the resources (e.g. Slice Logic) to view how much and which module consumes the resource.

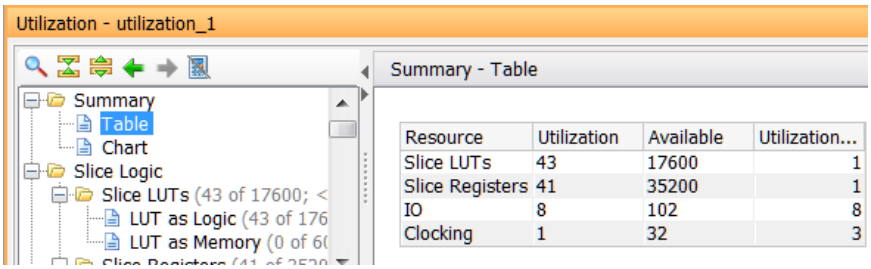



**Figure 17. Resource utilization**

**5-2-4.** Select **Implemented Design > Report Clock Networks**.

**5-2-5.** Click **OK**.The Clock Networks report will be displayed in the Console pane showing two clock net entries.

**5-2-6.** Select *clk_pin* entry and observe the selected nets in the Device view. The clock nets are located in the X1Y0 and X1Y1 clock regions.

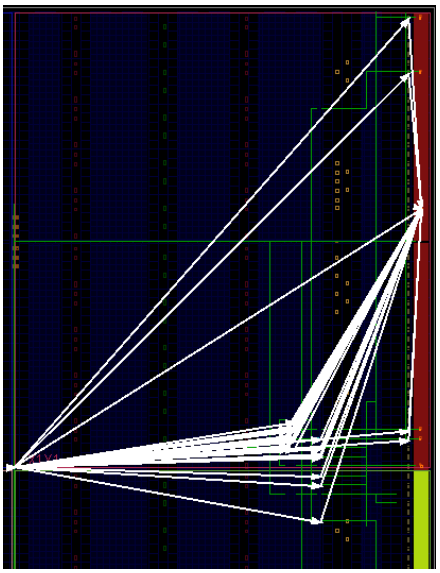


**Figure 18. Clock nets**

## Generate the Bitstream — Step 6

### 6-1. Generate the bitstream.

**6-1-1.** In the *Flow Navigator* pane, under *Program and Debug*, click **Generate Bitstream.**

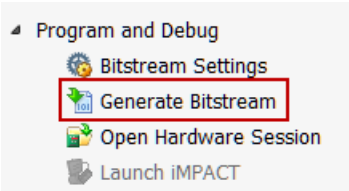



**Figure 19, Generating the bitstream**

**6-1-2.** Click **Cancel** when the bitstream generation is completed.

## Verify the Functionality Step 7

### 7-1. Connect a micro-usb cable between the PmodUSBUart module and insert the module into the top-row of the JE PMOD. Connect the board with another micro-usb cable and power it ON. Open a hardware session, and program the FPGA.

**7-1-1.** Connect a micro-USB cable between the PmodUSBUart module and the host PC USB port.

**7-1-2.** Plug-in the PmodUSBUart module into the top-row of the JE PMOD connector (below the slide-switch 4).

**7-1-3.** Make sure that another micro-USB cable is connected to the JTAG PROG connector (next to the power supply connector). Make sure that the JP7 is set to select USB power. Turn ON the power.

**7-1-4.** Select the *Open Hardware Manager* option and click **OK**.

The Hardware Manager window will open indicating “unconnected” status.

**7-1-5.** Click on the **Open a new hardware target** link.

You can also click on the **Open recent target** link if the board was already targeted before.


**Figure 18. Opening new hardware target**

**7-1-6.** Click **Next** to see the Vivado CSE Server Name form.

**7-1-7.** Click **Next** with the localhost port selected.

The JTAG cable which uses the Xilinx_tcf should be detected and identified as a hardware target. It will also show the hardware devices detected in the chain.

**7-1-8.** Click **Next** twice and **Finish**.

**7-1-9.** The Hardware Session status changes from Unconnected to the server name and the device is highlighted. Also notice that the Status indicates that it is not programmed.

**7-1-10.** Select the device in the *Hardware Device Properties,* and verify that the **uart_led.bit** is selected as the programming file in the General tab.

### 7-2. Start a terminal emulator program such as TeraTerm or HyperTerminal. Select an appropriate COM port (you can find the correct COM number using the Control Panel). Set the COM port for 115200 baud rate communication. Program the FPGA and verify the functionality.

**7-2-1.** Start a terminal emulator program such as TeraTerm or HyperTerminal.

**7-2-2.** Select an appropriate COM port (you can find the correct COM number using the Control Panel).

**7-2-3.** Set the COM port for 115200 baud rate communication.

**7-2-4.** Right-click on the FPGA entry in the Hardware window and select Programming Device…

**7-2-5.** Click on the Program button.

The programming bit file will be downloaded and the DONE light will be turned ON when the FPGA has been programmed.

**7-2-6.** Type in some characters in the terminal emulator window and see the corresponding ASCII equivalent bit pattern (least significant 4-bits) displayed on the LEDs.

**7-2-7.** Press and hold BTN0 and see the the upper four bits of the character.

**7-2-8.** When satisfied, close the terminal emulator program and power OFF the board.

**7-2-9.** Select **File > Close Hardware Manager**. Click **OK**.

**7-2-10.** Close the **Vivado** program by selecting **File > Exit** and click **OK**.

## Conclusion

The Vivado IDE tool can be used to perform a complete HDL design flow. The project was created using the supplied source files (HDL model and user constraint files). A behavioral simulation was done using the provided testbench files to verify the model functionality. The model was then synthesized, implemented, and a bitstream was generated. The functionality was verified in hardware using the generated bitstream.

## Answers

1. Look through the table and find the number used of each of the following:

| | |
|---|---|
| FF: | 45 |
| LUT: | 43 |
| I/O: | 8 |
| BUFG: | 1 |

# Xilinx Design Constraints

## Introduction

In this lab you will use the uart_led design that was introduced in the previous lab. You will start the project with I/O Planning type, enter pin locations, and export it to the rtl. You will then create the timing constraints and perform the timing analysis.

## Objectives

After completing this lab, you will be able to:

- Create a I/O Planning project
- Enter the pin locations and IO standards via Device view, Package Pins tab, and Tcl commands
- Create Period, Input Setup, and Output Setup delays
- Perform timing analysis

## Procedure

This lab is broken into steps that consist of general overview statements providing information on the detailed instructions that follow. Follow these detailed instructions to progress through the lab.

## Design Description

The design consists of a uart receiver receiving the input typed on a keyboard and displaying the least significant 4 bits of binary equivalent of the typed character on the 4 LEDs. When a push button is pressed, the upper bits of the binary equivalent are displayed. The block diagram is as shown in **Figure 1**.

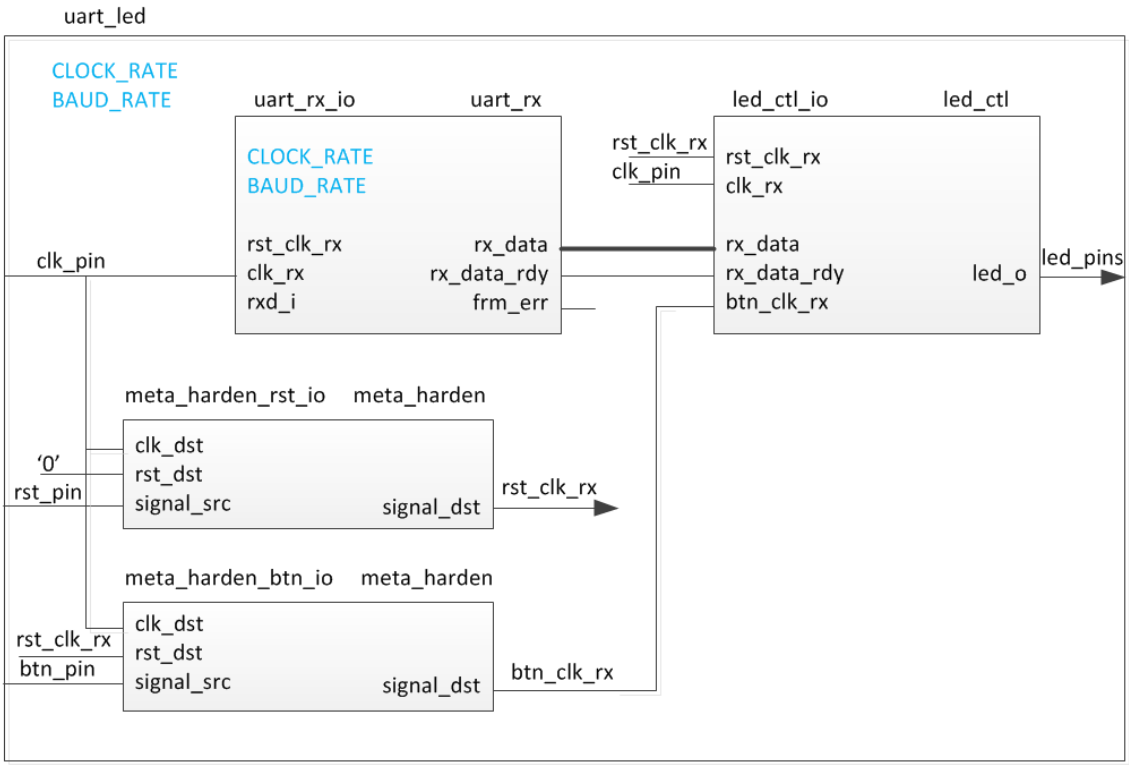
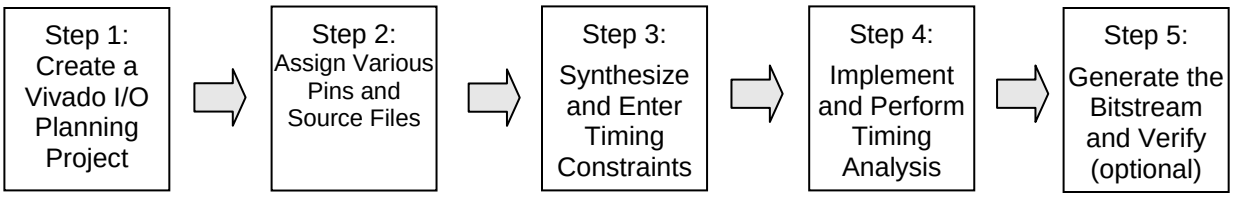



**Figure 1. The design**

## General Flow




## Create a Vivado I/O Planning Project — Step 1

### 1-1. Launch Vivado and create a I/O Planning project targeting the ZYBO.

**1-1-1.** Open Vivado by selecting **Start > All Programs > Xilinx Design Tools > Vivado 2013.4 > Vivado 2013.4**

**1-1-2.** Click **Create New Project** to start the wizard. You will see *Create A New Vivado Project* dialog box. Click **Next**.

**1-1-3.** Click the Browse button of the *Project location* field of the **New Project** form, browse to **c:\xup\sys_design\labs**, and click **Select**.

**1-1-4.** Enter **lab2** in the *Project name* field. Make sure that the *Create Project Subdirectory* box is checked. Click **Next**.

**1-1-5.** Select **I/O Planning Project** option in the *Project Type* form, and click **Next**.

**1-1-6.** Select **Do not import I/O ports at this time**, and click **Next**.

**1-1-7.** In the *Default Part* form, using the **Parts** option and various drop-down fields of the **Filter** section, select the *XC7Z010CLG400-1* part. Click **Next**.

**1-1-8.** Click **Finish** to create the Vivado project.

The device view window and package pins tab will be displayed.

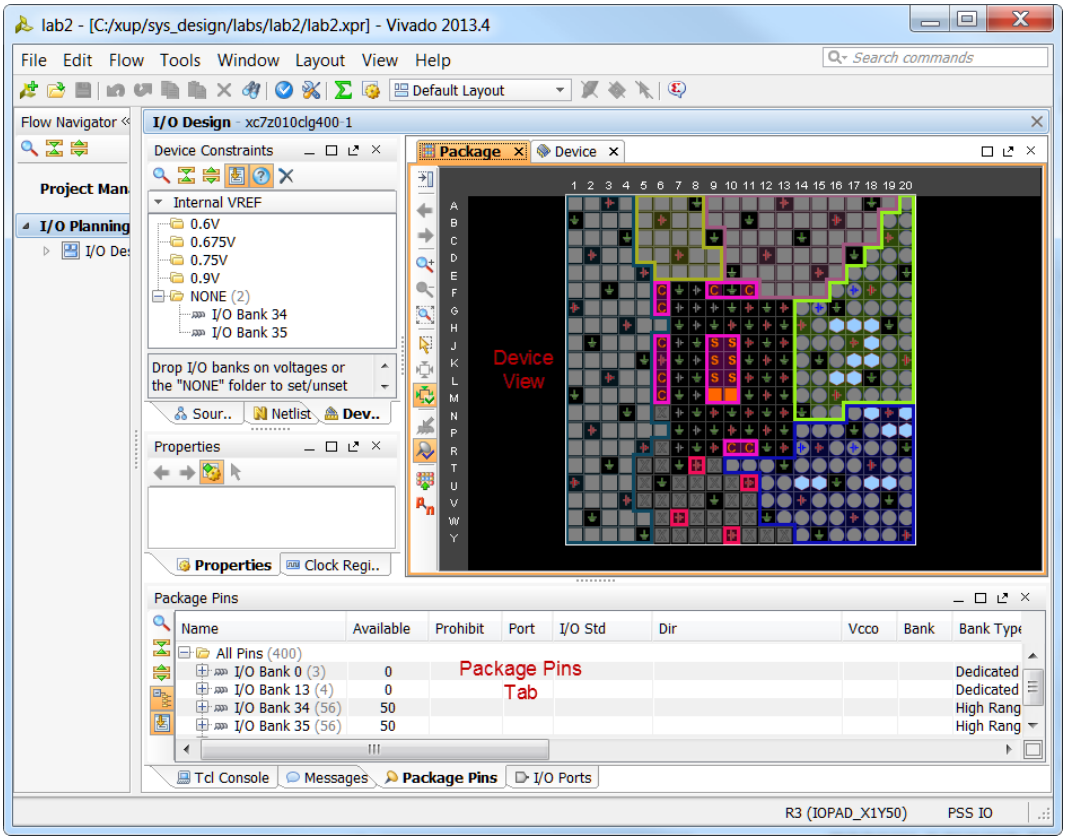



**Figure 2. I/O Planning project's default windows and views**

## Assign Various Pins and Add Source Files — Step 2

### 2-1. Assign input pins clk_pin, btn_pin, rxd_pin, and rst_pin to L16, R18, J15, and P16 locations using the Device view and package pins.

**2-1-1.** Move mouse over the Device view window and hover over it on the **L16** location.

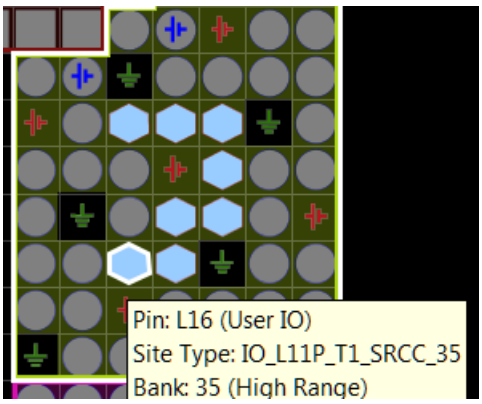



**Figure 3. Locating L16 pin in the Device view**

**2-1-2.** When located, click on it.

The L16 entry will be displayed in the Package Pins tab.

**2-1-3.** In the *Package* Pins pane, click in the Ports column of **L16** pin's row, enter **clk_pin.**

**2-1-4.** Click in the next column, *I/O Std*.

Notice that LVCMOS18 is displayed in Red under the IO Standard column. It is shown in Red because of mismatch IO standard.

**2-1-5.** Click on the *LVCMOS18* entry to see a pop-up window. Scroll down and select **LVCMOS33** and then hit **Enter**. The LVCMOS33 is displayed in black. Keep the direction as input since clk_pin is an input port.

**2-1-6.** Similarly, add the **btn_pin** input port at **R18** with *LVCMOS33* standard.

**2-1-7.** Select **Edit > Find** or Ctrl-F to open the Find form. Select **Package Pins** in the *Find* drop-down field, type ***J15** in the match criteria field, and click on **OK**.

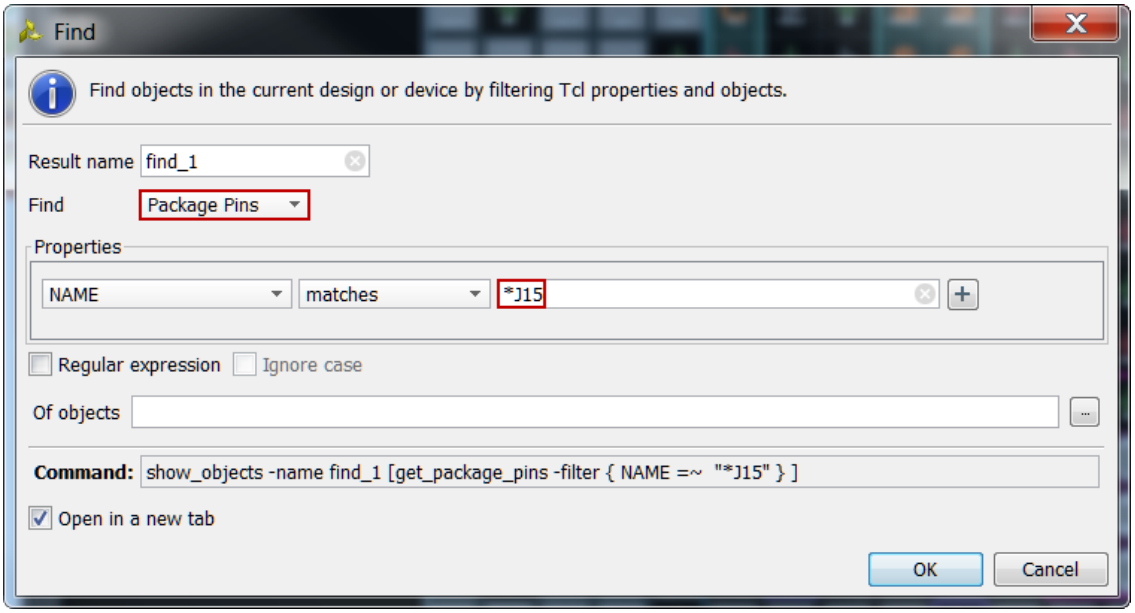



**Figure 4. Finding a package pin**

Notice that the J15 pin is highlighted in the Device view and the corresponding entry is displayed in the Package Pins tab.

**2-1-8.** Assign **rxd_pin** to *J15* with *LVCMOS33* standard.

**2-1-9.** Similarly add the **rst_pin** input assigning it to **P16** location with *LVCMOS33* standard.

## 2-2. Assign output pins led_pins[3] to led_pins[0] to D18, G14, M15, and M14 locations creating them as a vector and assigning them using Tcl command set_propoerty. They all will be LVCMOS33.

**2-2-1.** In the I/O Ports tab, click on the create I/O port button on the left vertical ribbon.

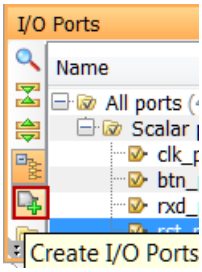



**Figure 5. Create I/O Ports button**

The Create I/O Ports form will be displayed.

**2-2-2.** Type **led_pins** in the *Name* field, select *Output* direction, click on the check-box of **Create bus**, set the msb to **3**, and select **LVCMOS33** I/O standard, and click **OK**.

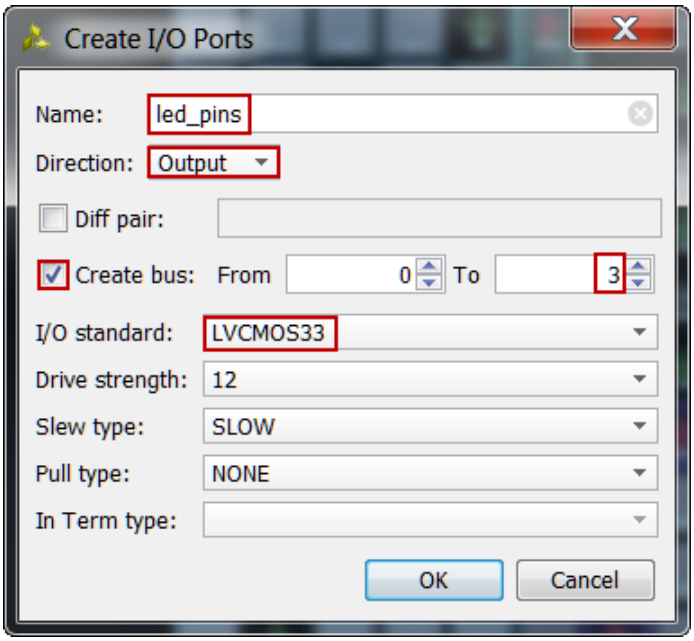



**Figure 6. Creating I/O ports for the led_pins output**

The led_pins entries will be created and displayed in the I/O Ports tab. Notice that the I/O standard and directions are already set, leaving only the pin locations to be assigned.

**2-2-3.** In the Tcl console, type the following commands to assign the pin locations:

```
set_property PACKAGE_PIN M14 [get_ports led_pins[0]]
set_property PACKAGE_PIN M15 [get_ports led_pins[1]]
set_property PACKAGE_PIN G14 [get_ports led_pins[2]]
set_property PACKAGE_PIN D18 [get_ports led_pins[3]]
```

**2-2-4.** Select **File > Save Constraints**.

The Save Constraints form will be displayed.

**2-2-5.** Enter **uart_led** in the *File name* field, and click **OK**.

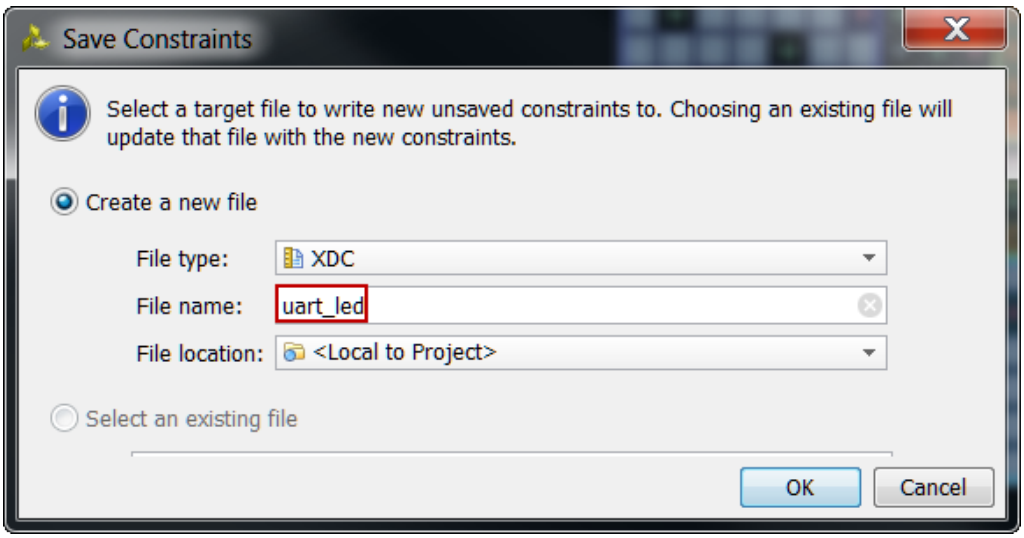



**Figure 7. Accessing clocking wizard**

The uart_led.xdc file will be created and added to the Sources tab.

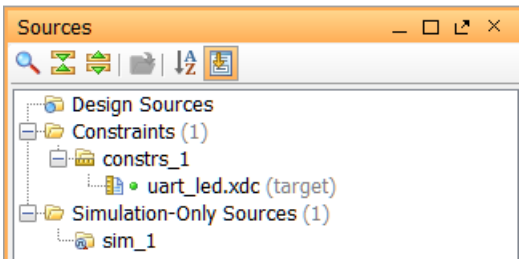



**Figure 8. The uart_led.xdc file added to the source tree**

**2-2-6.** Expand the **I/O Planning > I/O Design** in the *Flow Navigator* pane.

**2-2-7.** Click on **Report DRC** and click **OK**. Notice the design rules checker is run and some warnings are reported. Ignore the warnings.

**2-2-8.** Click on **Report Noise** and click **OK**. Notice the noise analysis is done on the output pins only (led_pins) and the results are displayed.

**2-2-9.** Click on **Migrate to RTL**.

The Migrate to RTL form will be displayed with Top RTL file field showing c:/xup/fpga_flow/labs/lab5/io_1.v entry.

**2-2-10.** Change *io_1.v* to **uart_led.v**, and click **OK**

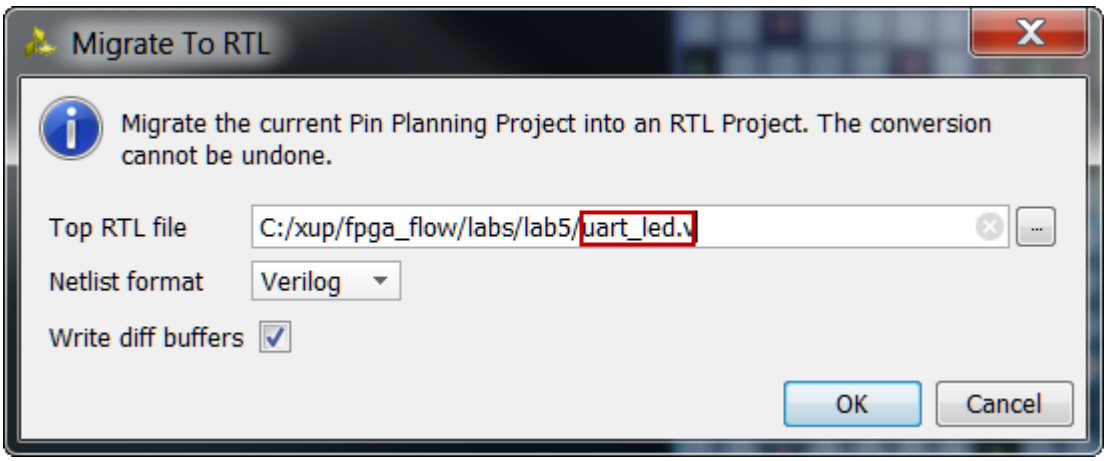



**Figure 9. Assigning top-level file name**

**2-2-11.** Select the **Hierarchy** tab and notice that the *uart_led.v* file has been added to the project with top-level module name as **ios**. If you double-click the entry, you will see the module name with the ports listing.

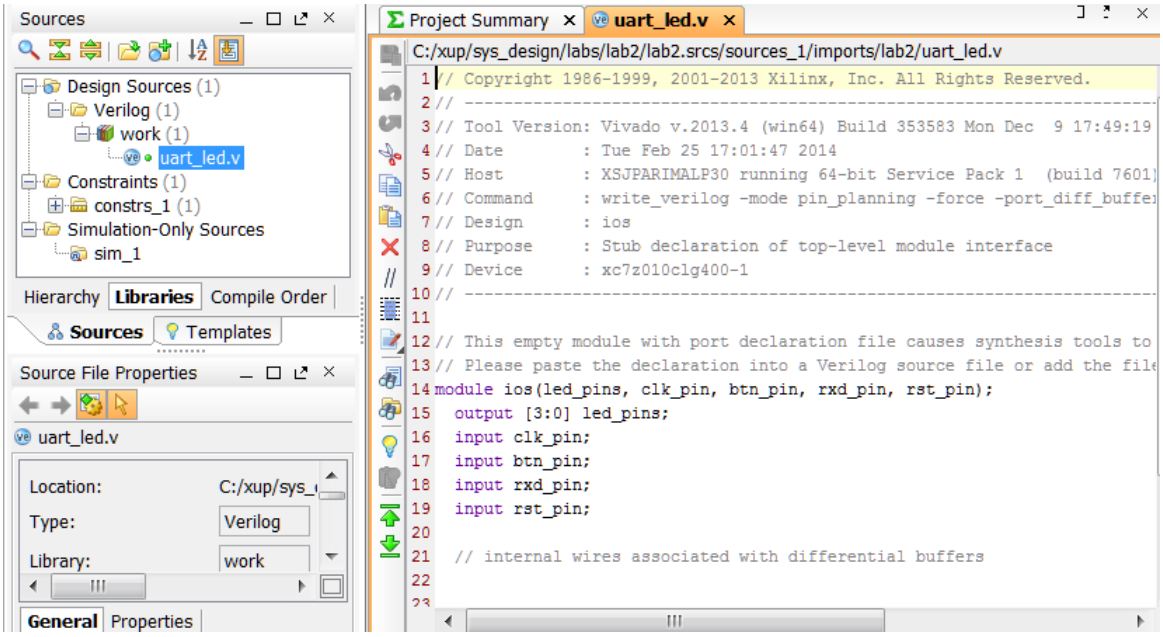



**Figure 10. The top-level module content and the design hierarchy after migrating to RTL**

## 2-3. Add the provided source files (from c:\xup\sys_design\sources\lab2) to the project. Copy the uart_led.txt (located in the c:\xup\sys_design\sources\lab2) content into the top-level (uart_led.v) source file.

**2-3-1.** Click on **Add Sources** in the *Flow Navigator*.

**2-3-2.** In the *Add Sources* form, select *Add or Create Design Sources*, and click **Next**.

**2-3-3.** Click **Add Files…**

**2-3-4.** Browse to **c:\xup\sys_design\sources\lab2** and select all *.v* files (led_ctrl, uart_rx, meta_harden, uart_baud_gen, uart_rx_ctl), and click **OK**.

**2-3-5.** Click **Finish**.

**2-3-6.** Using Windows Explorer, browse to **c:\xup\sys_design\sources\lab2** and open uart_led.txt using any text editor. Copy the content of it and paste it in uart_led.v (around line 20) in the Vivado project.

**2-3-7.** Select **File > Save File.**

## Synthesize and Enter Timing Constraints Step 3

### 3-1. Synthesize the design.

**3-1-1.** Click on the **Run Synthesis** in the *Flow Navigator* pane.

When synthesis is completed a form with three options will be displayed.

**3-1-2.** Select *Open Synthesized Design* and click **OK**.

**3-1-3.** Select **Synthesis > Synthesized Design > Edit Timing Constraints** in the Flow Navigator pane.

The Timing Constraints editor will open showing that there are no timing constraints entered so far.

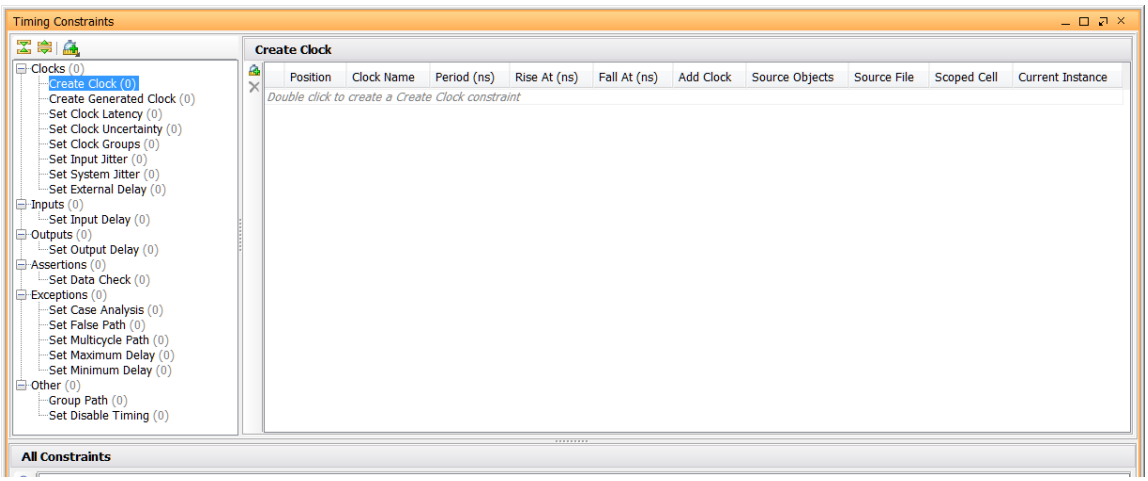



**Figure 11. Timing Constraints editor**

**3-1-4.** Double-click on the **Create Clock** in the left pane of the editor..

**3-1-5.** In the *Clock Name* field, enter **clk_pin**.

**3-1-6.** In the *Source objects*, click on the corresponding browse button.

**3-1-7.** In the *Specify Clock Source Object* form, make sure that *ports* is selected as the *File names of type*.

**3-1-8.** Click on the **Find** button which will list four results on the left side, select **clk_pin**, and click on the *right* arrow to assign as the clock source, and then click **OK** to accept it and close the form.

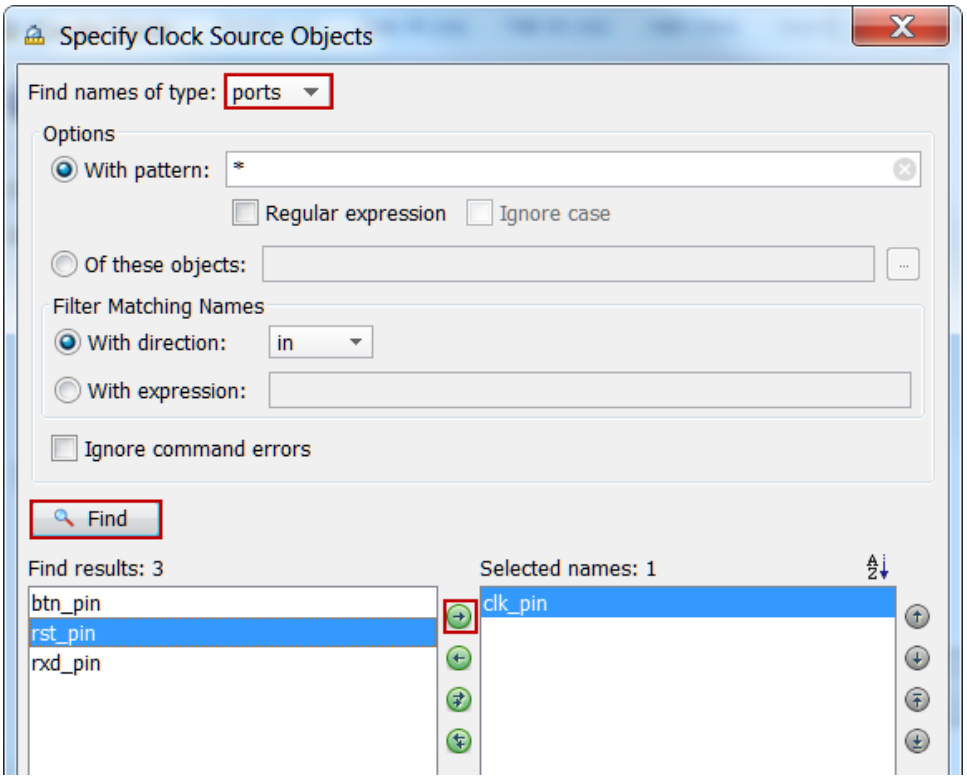



**Figure 12. Specifying clk_pin as the clock source object**

**3-1-9.** Set the period to be **8** ns, rise at 0 ns, and fall at **4** ns. Click **OK**.

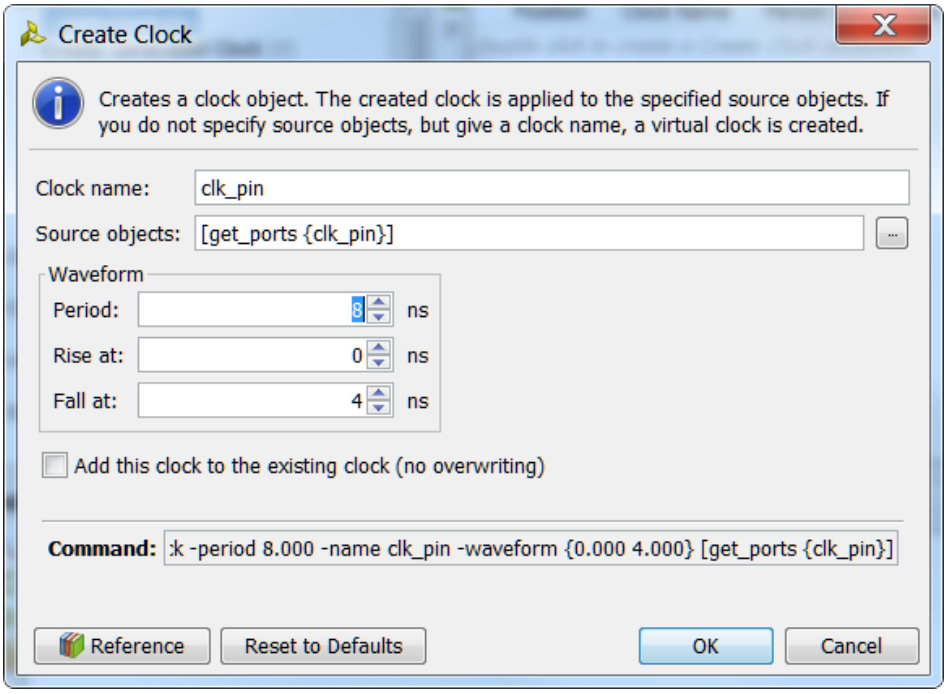



**Figure 13. Assigning period, rise time, and fall time to the clock**

The period constraint will be created. Notice the tcl command for this constraint is displayed.

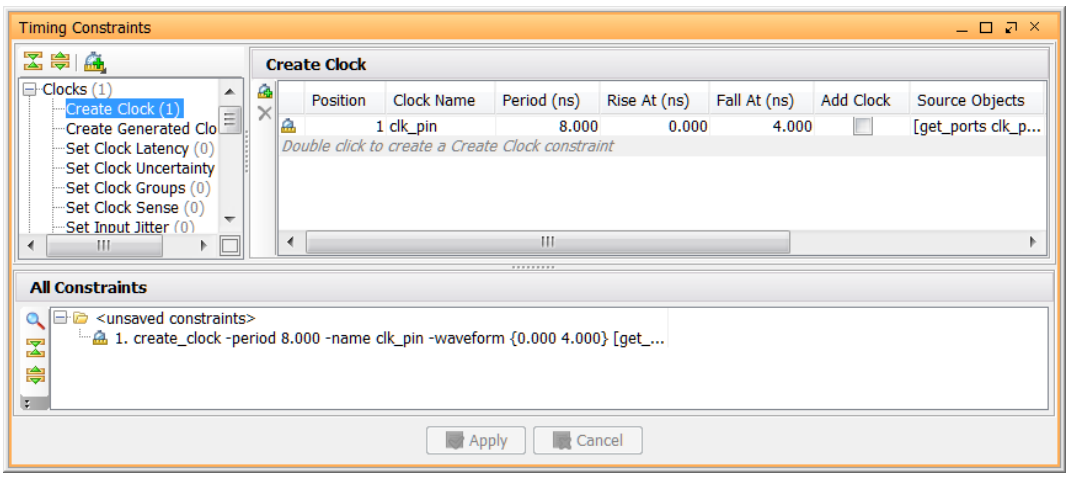



**Figure 14. The period constraint**

## 3-2. Assign Input Setup Delay of 0 ns to the btn_pin, rxd_pin, and rst_pin input with respect to the clk_pin port.

**3-2-1.** In the *Timing Constraints* window, under the *Inputs* category, double-click **Set Input Delay (0)**.

The *Set Input Delay* dialog box opens.

**3-2-2.** Click the browse button next to the *Clock* field.

The *Specify Clock* dialog box opens.

**3-2-3.** Ensure that **clocks** is selected from the *Find names of type* drop-down list.

**3-2-4.** Click **Find.** In the Find results section, select **clk_pin,** and click the *right* arrow to select **clk_pin,** and click **OK.**

**3-2-5.** Click the browse button next to the *Objects (Ports)* field.

The *Specify Delay Objects* dialog box opens.

**3-2-6.** Ensure that **ports** is selected from the Find names of type drop-down list. Click **Find** to see the four ports listed in the left side.

**3-2-7.** Select *btn_pin, rst_pin*, and *rxd_pin*, and click the right arrow, and click **OK**.

**3-2-8.** Leave the *Delay value* field to 0 ns as the setup requirement, and click **OK**.

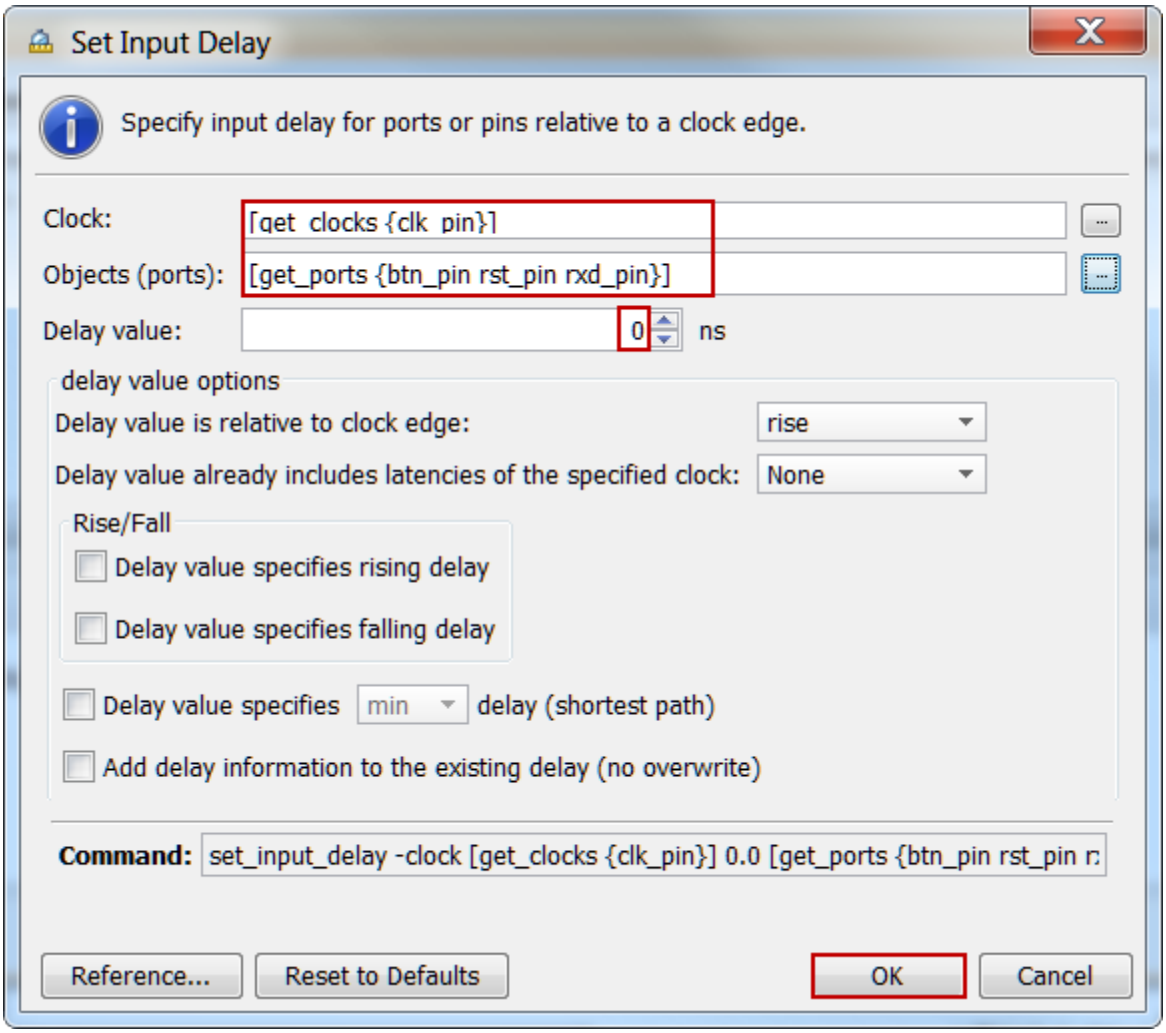



**Figure 15. Assigning 0 ns as the input delay to btn_pin, rst_pin, rxd_pin with respect to clk_pin**

## 3-3. Assign -0.5 ns as the minimum hold time on the btn_pin, rxd_pin, and rst_pin input with respect to the clk_pin port.

**3-3-1.** In the Timing Constraints window, under the Inputs category, double-click **Set Input Delay (1)**.

The *Set Input Delay* dialog box opens with the recently entered data

**3-3-2.** Enter **-0.5** in the *Delay* value field.

**3-3-3.** Select the **Delay value specifies <min/max> delay** option.

**3-3-4.** Select **min** from the Delay value specifies <min/max> delay drop-down list.

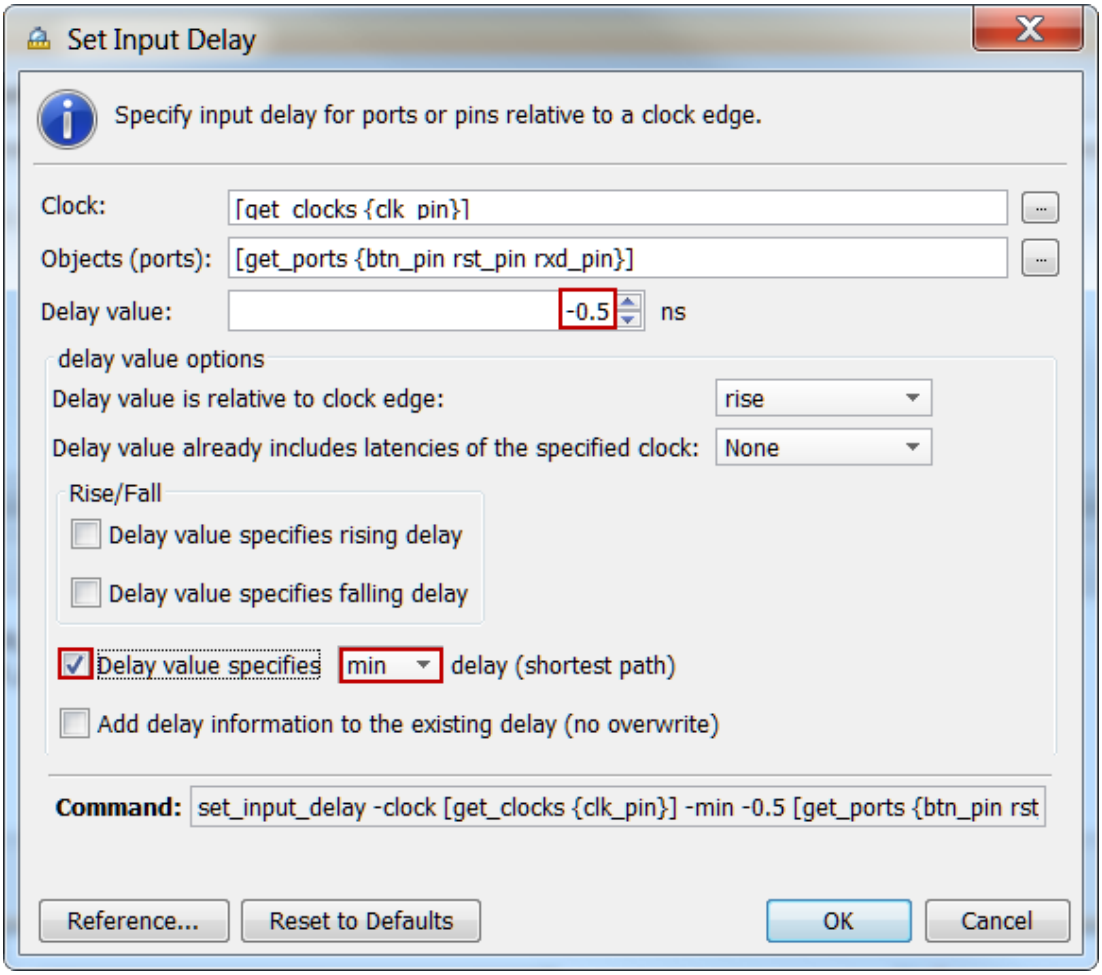



**Figure 16. Assigning hold time to the input pins**

**3-3-5.** Click **OK**.

This will create set_input_delay with a -0.5 ns hold time requirement on the rst_pin, rxd_pin, and btn_pin ports.

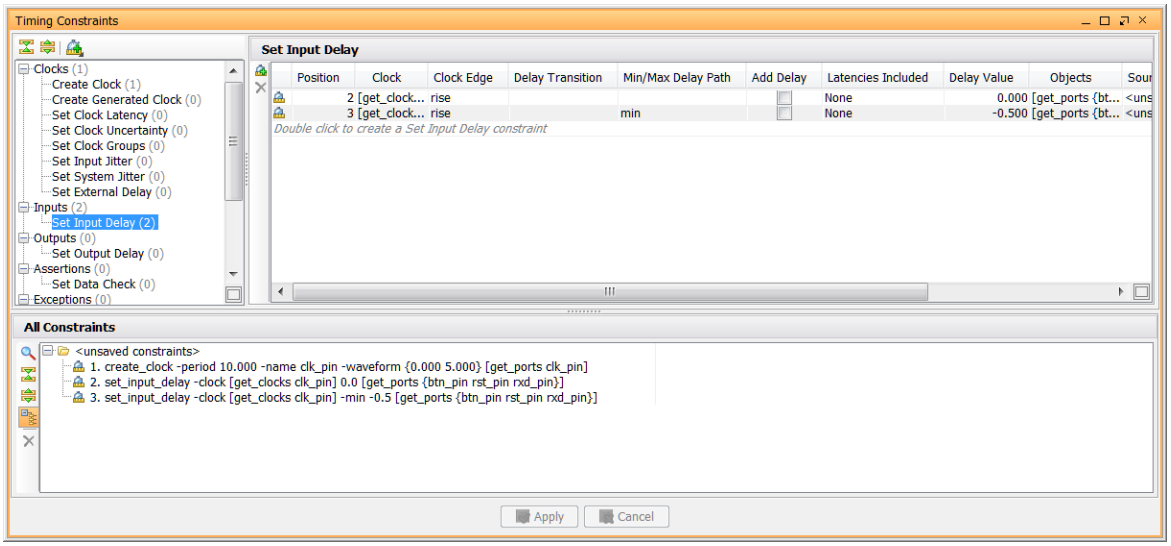



**Figure 17. Period and Input Delay constraints assigned**

## 3-4. Assign 0 ns as the output delay on the led_pins (all four) output with respect to the clk_pin port.

**3-4-1.** In the Timing Constraints window, under the Outputs category, double-click **Set Output Delay (0)**.

The *Set Output Delay* dialog box opens.

**3-4-2.** Click the browse button next to the *Clock* field.

The *Specify Clock* dialog box opens.

**3-4-3.** Ensure that **clocks** is selected from the *Find names of type* drop-down list.

**3-4-4.** Click **Find.** In the Find results section, select **clk_pin,** and click the *right* arrow to select **clk_pin,** and click **OK.**

**3-4-5.** Click the browse button next to the Objects (Ports) field.

The *Specify Delay Objects* dialog box opens.

**3-4-6.** Ensure that **ports** is selected from the *Find names of type* drop-down list. Click **Find** to see the eight output ports (led_pin) listed in the left side.

**3-4-7.** Click the *right double-arrow* to assign all output ports on the right, and click **OK**.

**3-4-8.** Leave the *Delay value* field to 0 ns as the setup requirement, and click **OK**.

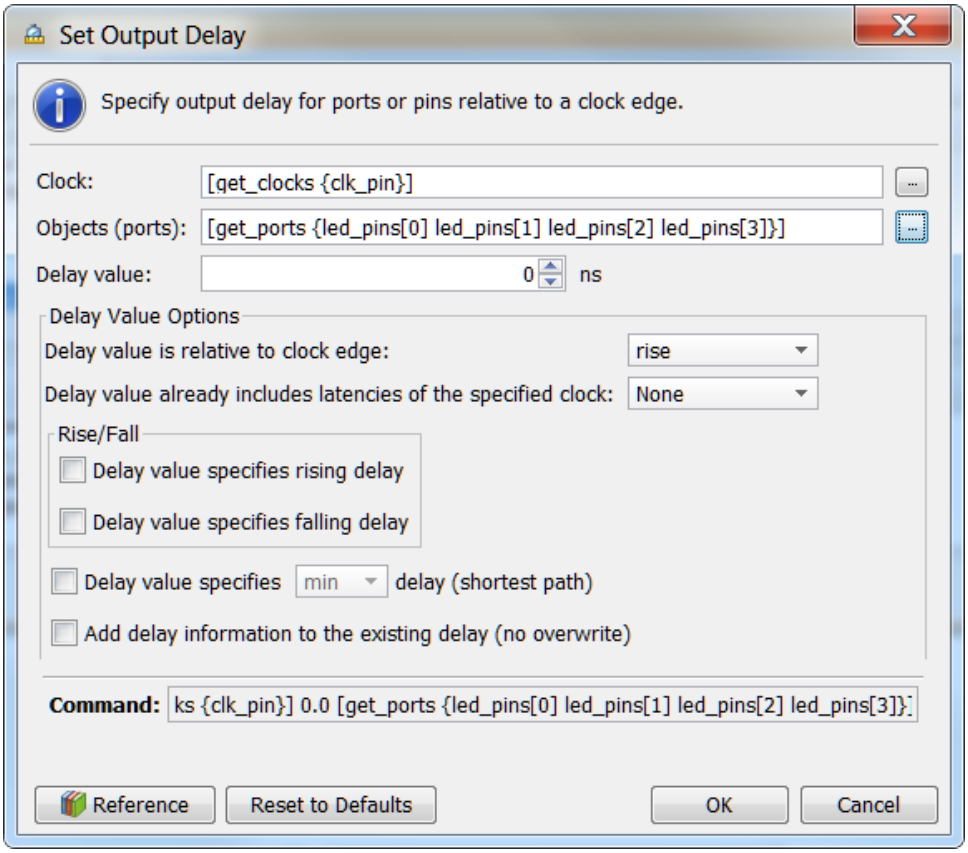



**Figure 18. Assigning all four output ports 0ns output delay**

**3-4-9.** The *All Constraints* window will show the assigned constraints.

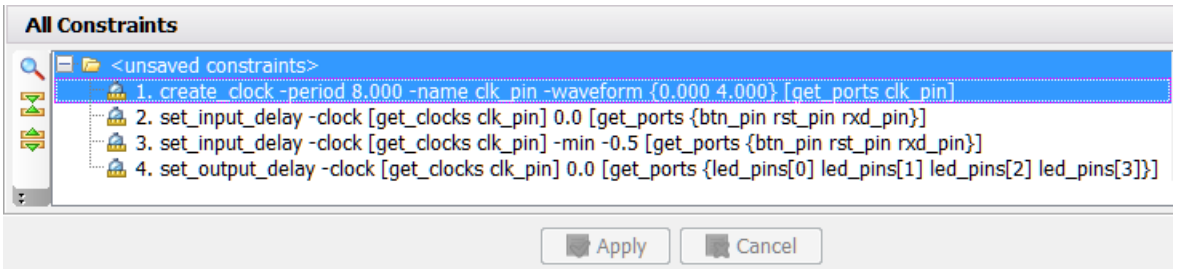



**Figure 19. All created constraints**

**3-4-10.** Select **File** > **Save Constraints**.

The constraints will be added at end of the uart_led.xdc file.

## 3-5. Generate an estimated Timing Report showing both the setup and hold paths in the design.

**3-5-1.** In the Flow Navigator, select **Synthesized Design** > **Report Timing Summary.**

**3-5-2.** In the Options tab, select **min_max** from the Path delay type drop-down list.

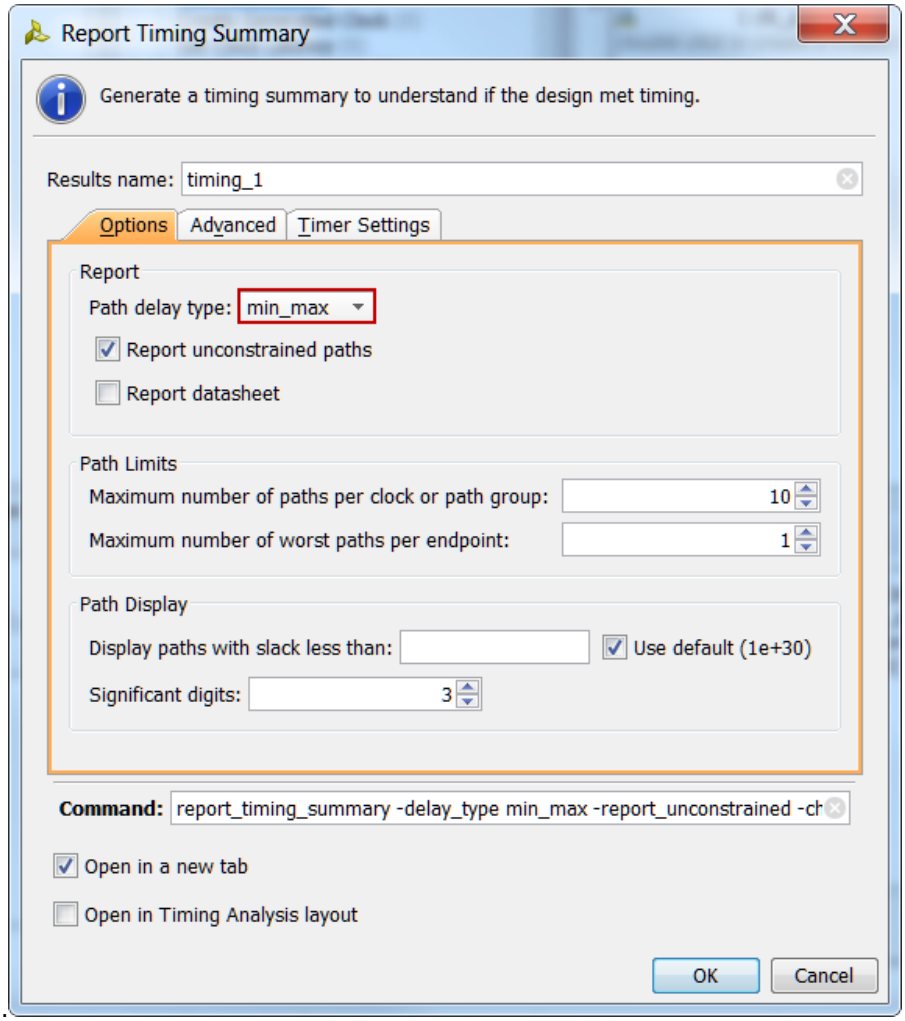



**Figure 20. Performing timing analysis**

**3-5-3.** Click **OK** to run the analysis.

The Timing Results view opens at the bottom of the Vivado IDE.

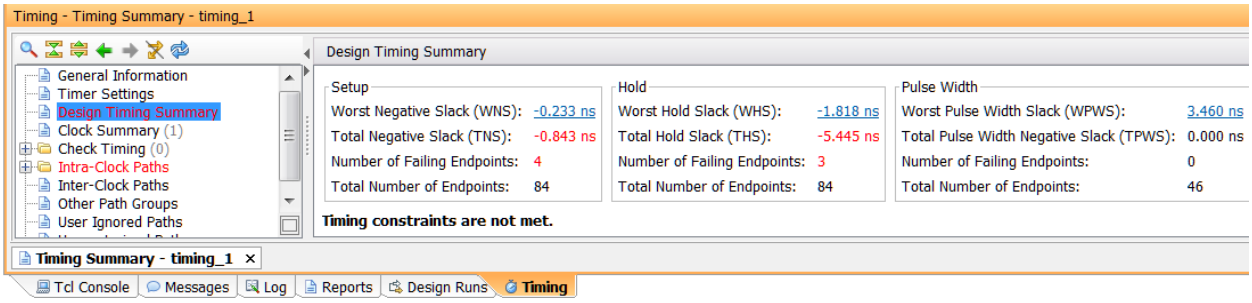



**Figure 21. Timing summary**

The *Design Timing Summary* report provides a brief worst Setup and Hold slack information and Number of failing endpoints to indicate whether the design has met timing or not.

Note that there are four timing failures under the setup check and three timing failures under the hold check.

**3-5-4.** Click on the link next to *Worst Hold Slack* (WHS) to see the list of failing paths.

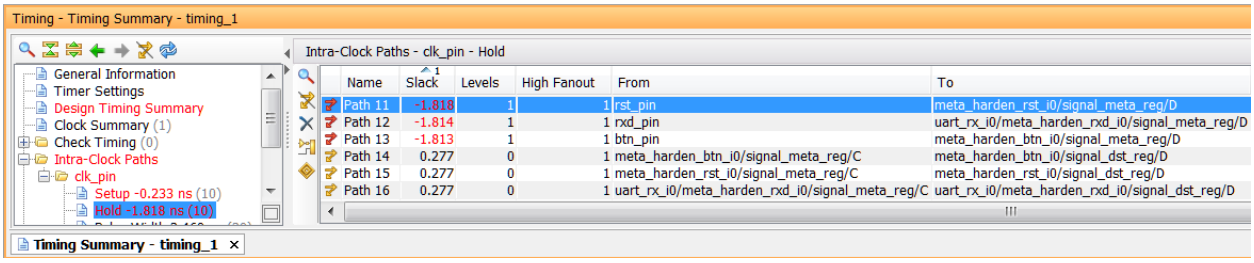
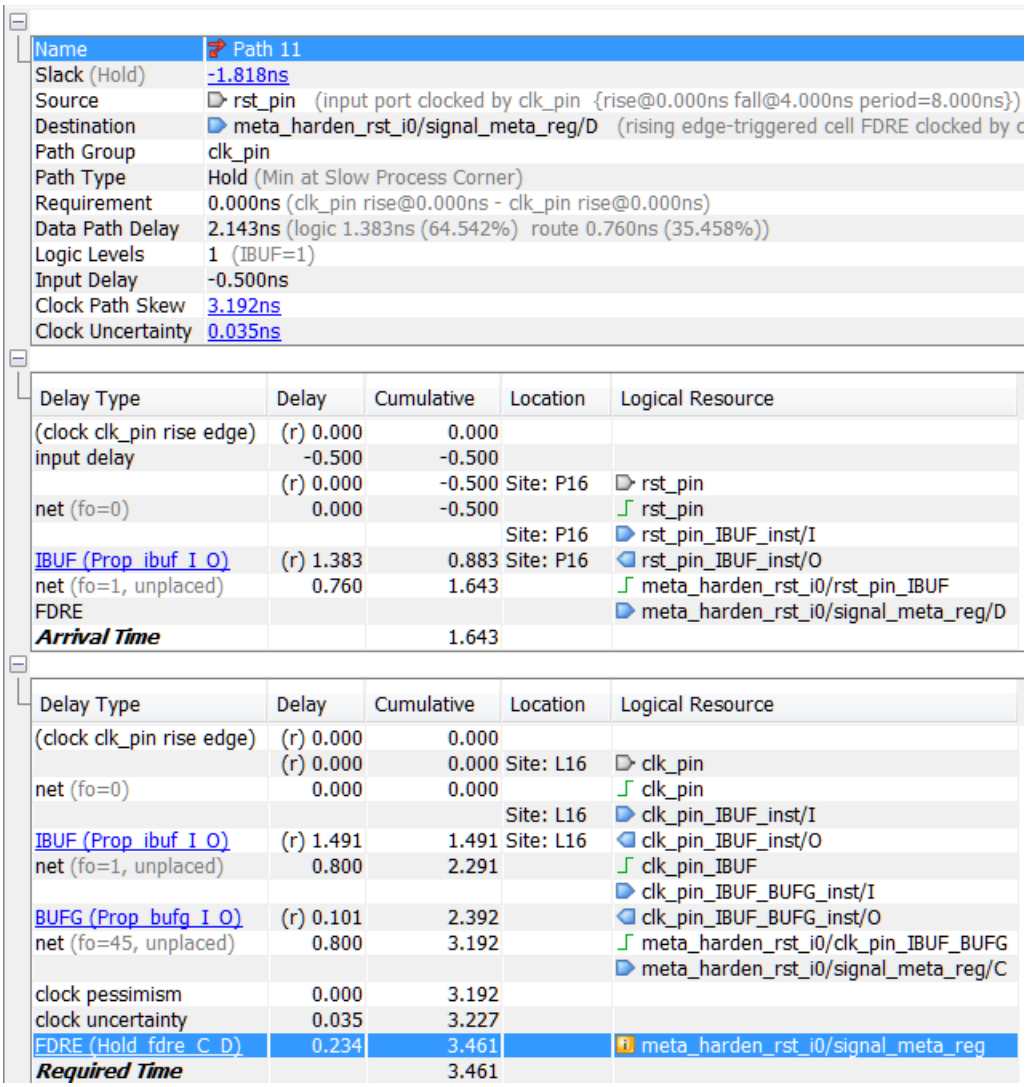



**Figure 22. The list of paths showing hold violations**

**3-5-5.** Double-click on the Path 11 to see the actual path detail.



| | |
|---|---|
| Name | Path 11 |
| Slack (Hold) | -1.818ns |
| Source | rst_pin (input port clocked by clk_pin {rise@0.000ns fall@4.000ns period=8.000ns}) |
| Destination | meta_harden_rst_i0/signal_meta_reg/D (rising edge-triggered cell FDRE clocked by c |
| Path Group | clk_pin |
| Path Type | Hold (Min at Slow Process Corner) |
| Requirement | 0.000ns (clk_pin rise@0.000ns - clk_pin rise@0.000ns) |
| Data Path Delay | 2.143ns (logic 1.383ns (64.542%) route 0.760ns (35.458%)) |
| Logic Levels | 1 (IBUF=1) |
| Input Delay | -0.500ns |
| Clock Path Skew | 3.192ns |
| Clock Uncertainty | 0.035ns |

| Delay Type | Delay | Cumulative | Location | Logical Resource |
|---|---|---|---|---|
| (clock clk_pin rise edge) | (r) 0.000 | 0.000 | | |
| input delay | -0.500 | -0.500 | | |
| | (r) 0.000 | -0.500 | Site: P16 | rst_pin |
| net (fo=0) | 0.000 | -0.500 | | rst_pin |
| | | | Site: P16 | rst_pin_IBUF_inst/I |
| IBUF (Prop_ibuf_I_O) | (r) 1.383 | 0.883 | Site: P16 | rst_pin_IBUF_inst/O |
| net (fo=1, unplaced) | 0.760 | 1.643 | | meta_harden_rst_i0/rst_pin_IBUF |
| FDRE | | | | meta_harden_rst_i0/signal_meta_reg/D |
| ***Arrival Time*** | | 1.643 | | |

| Delay Type | Delay | Cumulative | Location | Logical Resource |
|---|---|---|---|---|
| (clock clk_pin rise edge) | (r) 0.000 | 0.000 | | |
| | (r) 0.000 | 0.000 | Site: L16 | clk_pin |
| net (fo=0) | 0.000 | 0.000 | | clk_pin |
| | | | Site: L16 | clk_pin_IBUF_inst/I |
| IBUF (Prop_ibuf_I_O) | (r) 1.491 | 1.491 | Site: L16 | clk_pin_IBUF_inst/O |
| net (fo=1, unplaced) | 0.800 | 2.291 | | clk_pin_IBUF |
| | | | | clk_pin_IBUF_BUFG_inst/I |
| BUFG (Prop_bufg_I_O) | (r) 0.101 | 2.392 | | clk_pin_IBUF_BUFG_inst/O |
| net (fo=45, unplaced) | 0.800 | 3.192 | | meta_harden_rst_i0/clk_pin_IBUF_BUFG |
| | | | | meta_harden_rst_i0/signal_meta_reg/C |
| clock pessimism | 0.000 | 3.192 | | |
| clock uncertainty | 0.035 | 3.227 | | |
| FDRE (Hold_fdre_C_D) | 0.234 | 3.461 | | meta_harden_rst_i0/signal_meta_reg |
| ***Required Time*** | | 3.461 | | |

**Figure 23. Failing hold path**

**3-5-6.** Select Path11, right-click and select Schematic.

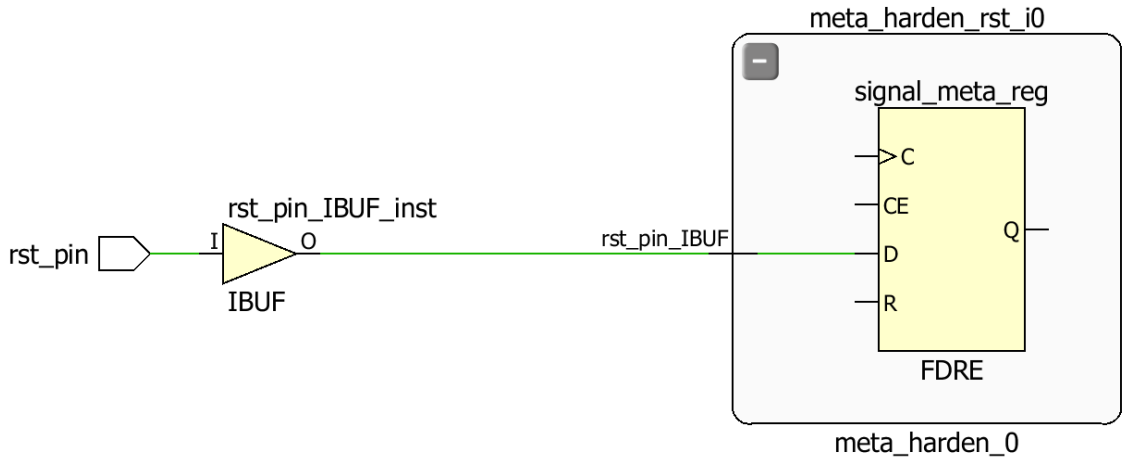



**Figure 24. The schematic of the failing path**

You can see that the failing path is at the input of rst_pin. Similarly, Path 12 and Path 13 are of btn_pin and rxd_pin.

## Implement and Analyze Timing Summary — Step 4

### 4-1. Implement the design.

**4-1-1.** Click on the **Run Implementation** in the *Flow Navigator* pane.

**4-1-2.** Click **Yes** and run the synthesis first before running the implementation process.

When the implementation is completed, a dialog box will appear with three options.

**4-1-3.** Select the *Open Implemented Design* option and click **OK**.

**4-1-4.** Click **Yes** if you are prompted to close the synthesized design.

### 4-2. Generate a timing summary report.

**4-2-1.** In the *Flow Navigator* pane, under *Implementation* > *Implemented Design*, click **Report Timing Summary**.

**4-2-2.** Click **OK** to generate the report using the default settings.

The Design Timing Summary window opens at the bottom in the Timing tab.

Note that failing timing paths are indicated in red.

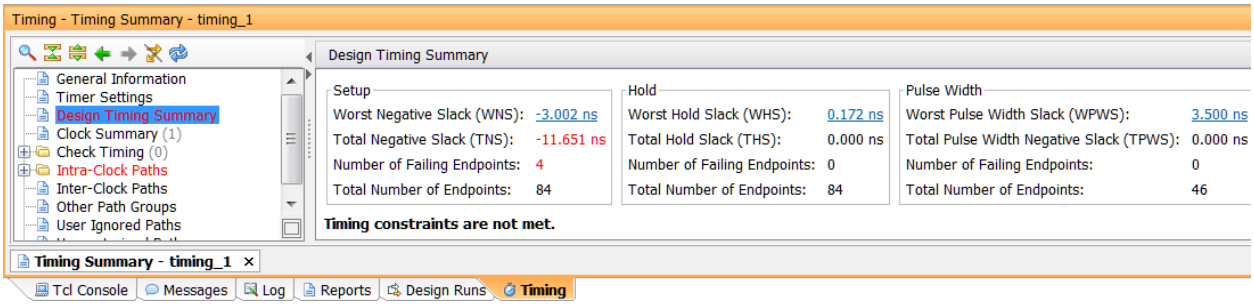



**Figure 25. Failing setup paths**

**4-2-3.** Click on the *WNS* to see the failing paths.

**4-2-4.** Double-click on the 1st failing path and see the detailed analysis. Note that about 5.192 ns was clock arrival delay to the register's clock port. 5.774 ns (10.966-5.192) is spent in the data input side, and 7.965 ns was the required output delay, with violation of about -3.001 (10.966-7.965) ns.

The output path delay can be reduced by placing the register in IOB.

**4-2-5.** Apply the constraint by typing the following command in the Tcl console.

set_property IOB TRUE [get_ports led_pins[*]]

**4-2-6.** Select **File > Save Constraints**.

**4-2-7.** Click on **Run Implementation**.

**4-2-8.** Click **Yes** and then **OK** to reset the synthesis run, perform the synthesis, and run the implementation.

## Generate the Bitstream and Verify the Functionality (Optional) Step 5

### 5-1. Generate the bitstream.

**5-1-1.** In the Flow Navigator, under *Program and Debug*, click **Generate Bitstream.**

**5-1-2.** The `write_bitstream` command will be executed (you can verify it by looking in the Tcl console).

**5-1-3.** Click **Cancel** when the bitstream generation is completed.

### 5-2. Connect a micro-usb cable between the PmodUSBUart module and insert the module into the top-row of the JE PMOD. Connect the board with another micro-usb cable and power it ON. Open a hardware session, and program the FPGA.

**5-2-1.** Connect a micro-USB cable between the PmodUSBUart module and the host PC USB port.

**5-2-2.** Plug-in the PmodUSBUart module into the top-row of the JE PMOD connector (below the slide-switch 4).

**5-2-3.** Make sure that another micro-USB cable is connected to the JTAG PROG connector (next to the power supply connector). Make sure that the JP7 is set to select USB power. Turn ON the power.

**5-2-4.** Select the *Open Hardware Manager* option and click **OK**.

The Hardware Manager window will open indicating “unconnected” status.

**5-2-5.** Click on the **Open New Hardware Target** link.

You can also click on the Open Recent Hardware Target link if the board was already targeted before. In this case skip to step 5-2-10.

**5-2-6.** Click **Next** to see the Vivado CSE Server Name form.

**5-2-7.** Click **Next** with the localhost port selected.

The JTAG cable which uses the digilent_plugin should be detected and identified as a hardware target. It will also show the hardware devices detected in the chain.

**5-2-8.** Click **Next** twice and **Finish**.

**5-2-9.** The Hardware Manager status changes from Unconnected to the server name and the device is highlighted. Also notice that the Status indicates that it is not programmed.

**5-2-10.** Select the device and verify that the **ios.bit** is selected as the programming file in the General tab.

## 5-3. Start a terminal emulator program such as TeraTerm or HyperTerminal. Select an appropriate COM port (you can find the correct COM number using the Control Panel). Set the COM port for 115200 baud rate communication. Program the FPGA and verify the functionality.

**5-3-1.** Start a terminal emulator program such as TeraTerm or HyperTerminal.

**5-3-2.** Select an appropriate COM port (you can find the correct COM number using the Control Panel).

**5-3-3.** Set the COM port for 115200 baud rate communication.

**5-3-4.** Right-click on the FPGA entry in the Hardware window and select Programming Device…

**5-3-5.** Click on the **Program** button.

The programming bit file be downloaded and the DONE light will be turned ON indicating the FPGA has been programmed.

**5-3-6.** Verify the functionality as you did in the previous lab, by typing some characters into the terminal, and watching the corresponding values appear on the LEDs.

**5-3-7.** When satisfied, close the terminal emulator program and power OFF the board.

**5-3-8.** Select **File > Close Hardware Manager**. Click **OK** to close it.

**5-3-9.** When done, close the **Vivado** program by selecting **File > Exit** and click **OK**.

## Conclusion

In this lab, you learned how to create an I/O Planning project and assign the pins via the Device view, Package Pins tab, and the Tcl commands. You then exported to the rtl project where you added the provided source files. Next you created timing constraints and performed post-synthesis and post-implementation timing analysis.

# Embedded System Design using IP Integrator

## Introduction

This lab guides you through the process of using Vivado and IP Integrator to create a simple ARM Cortex-A9 based processor design targeting the ZYBO board. You will use IP Integrator to create the hardware block diagram and SDK (Software Development Kit) to create an example application to verify the hardware functionality.

## Objectives

After completing this lab, you will be able to:

- Create a Vivado project for a Zynq system
- Use the IP Integrator to create a hardware system
- Use IP Catalog to use AXI GPIO peripheral to extend the design
- Use SDK to create a standard memory test project
- Run the test application on the board

## Procedure

This lab is separated into steps that consist of general overview statements that provide information on the detailed instructions that follow. Follow these detailed instructions to progress through the lab.

## Design Description

The purpose of the lab exercise is to walk you through a complete hardware and software processor system design. The following diagram represents the completed design (**Figure 1**).

In this lab, you will use IP Integrator to create a processing system based design consisting of the following:

- ARM Cortex A9 core (PS)
- UART for serial communication
- DDR3 controller for external DDR3_SDRAM memory
- AXI Interconnect block
- Two instances of GPIO peripheral to connect push-buttons and slide switches

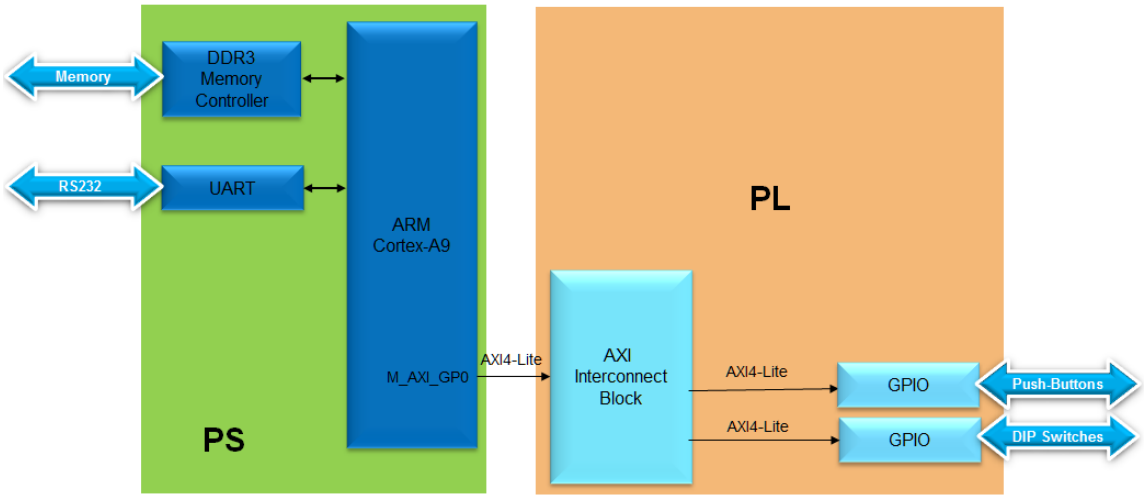
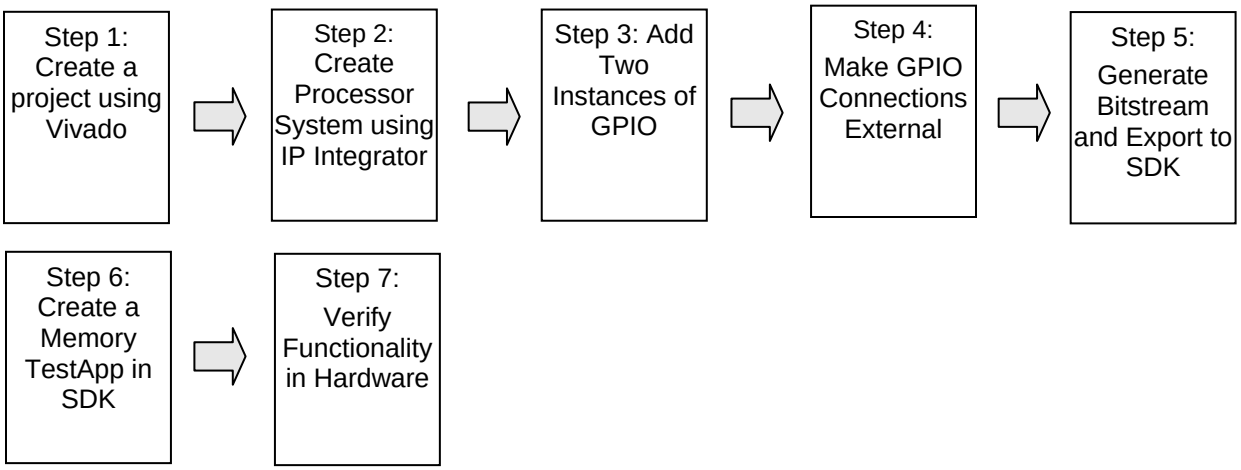



**Figure 1. Processor Design of this Lab**

## General Flow for this Lab




## Create a Vivado Project Step 1

### 1-1. Launch Vivado and create an empty project targeting the ZYBO (having xc7z010clg400-1 device) and using the Verilog language.

**1-1-1.** Open Vivado by selecting **Start > All Programs > Xilinx Design Tools > Vivado 2013.4 > Vivado 2013.4**

**1-1-2.** Click **Create New Project** to start the wizard. You will see the *Create a New Vivado Project* dialog box. Click **Next**.

**1-1-3.** Click the Browse button of the *Project Location* field of the **New Project** form, browse to **c:\xup\sys_design\labs**, and click **Select**.

**1-1-4.** Enter **lab3** in the *Project Name* field. Make sure that the *Create Project Subdirectory* box is checked. Click **Next**.

**1-1-5.** Select **RTL Project** in the *Project Type* form, and click **Next**.

**1-1-6.** Select **Verilog** as the *Target language* and as the *Simulator language* in the *Add Sources* form, and click **Next**.

**1-1-7.** Click **Next** two more times to skip *Adding Existing IP* and *Add Constraints*

**1-1-8.** In the *Default Part* form, select *Parts*, and using various filters shown in the figure below, select **xc7z010clg400-1** part as it is on the ZYBO board. Click **Next**.

**1-1-9.** Check the *Project Summary* and click **Finish** to create an empty Vivado project.

## Create the System Using the IP Integrator Step 2

### 2-1. Use the IP Integrator to create a new Block Design, and generate the ARM Cortex-A9 processor based hardware system, targeting the ZYBO.

**2-1-1.** In the *Flow Navigator* pane, click **Create Block Design** under IP Integrator

**2-1-2.** Enter **system** for the design name and click **OK**

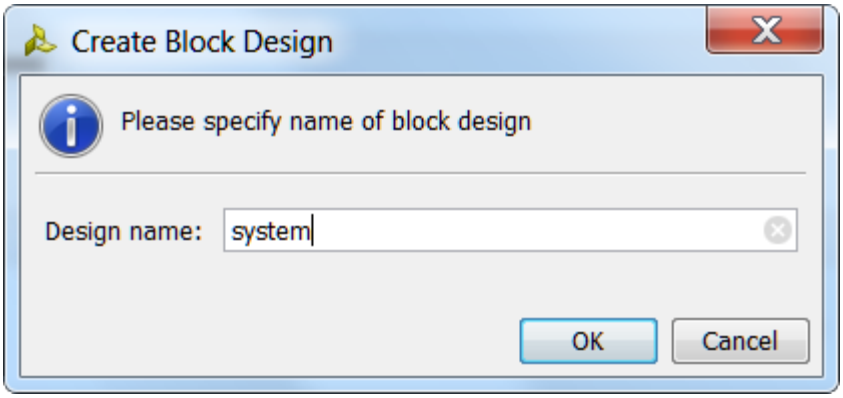



**Figure 2. Create New Block Diagram**

**2-1-3.** IP from the catalog can be added in different ways. Click on *Add IP* in the message at the top of the *Diagram* panel, or click the *Add IP icon* in the block diagram side bar, press Ctrl + I, or right-click anywhere in the Diagram workspace and select Add IP

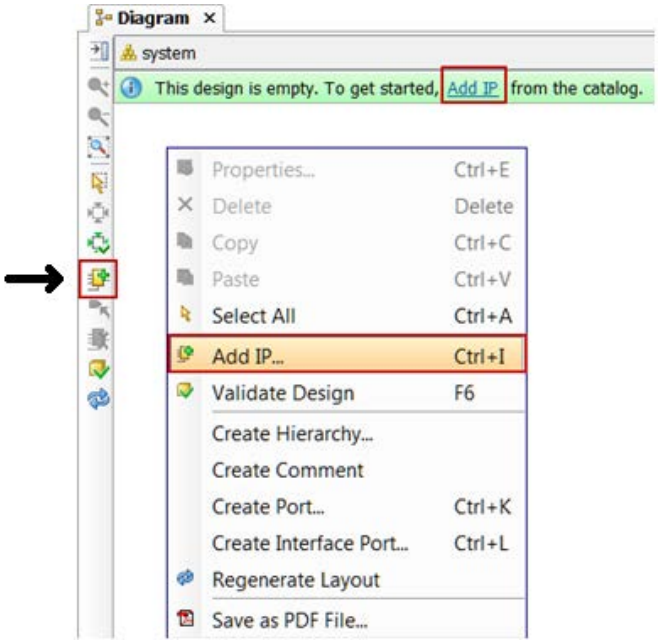



**Figure 3. Add IP to Block Diagram**

**2-1-4.** Once the IP Catalog is open, type “zy” into the Search bar, find and double click on **ZYNQ7 Processing System** entry, or click on the entry and hit the Enter key to add it to the design.

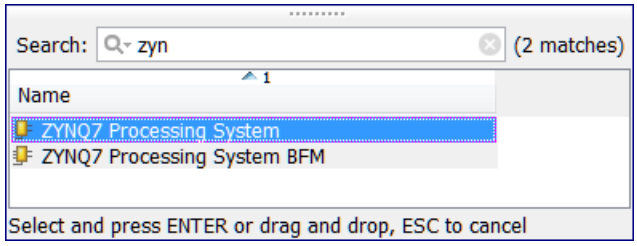



**Figure 4. Add Zynq block to the design**

**2-1-5.** Notice the message at the top of the Diagram window that *Designer Assistance* is available.

**2-1-6.** Double-click on the added block to open its *Customization* window.

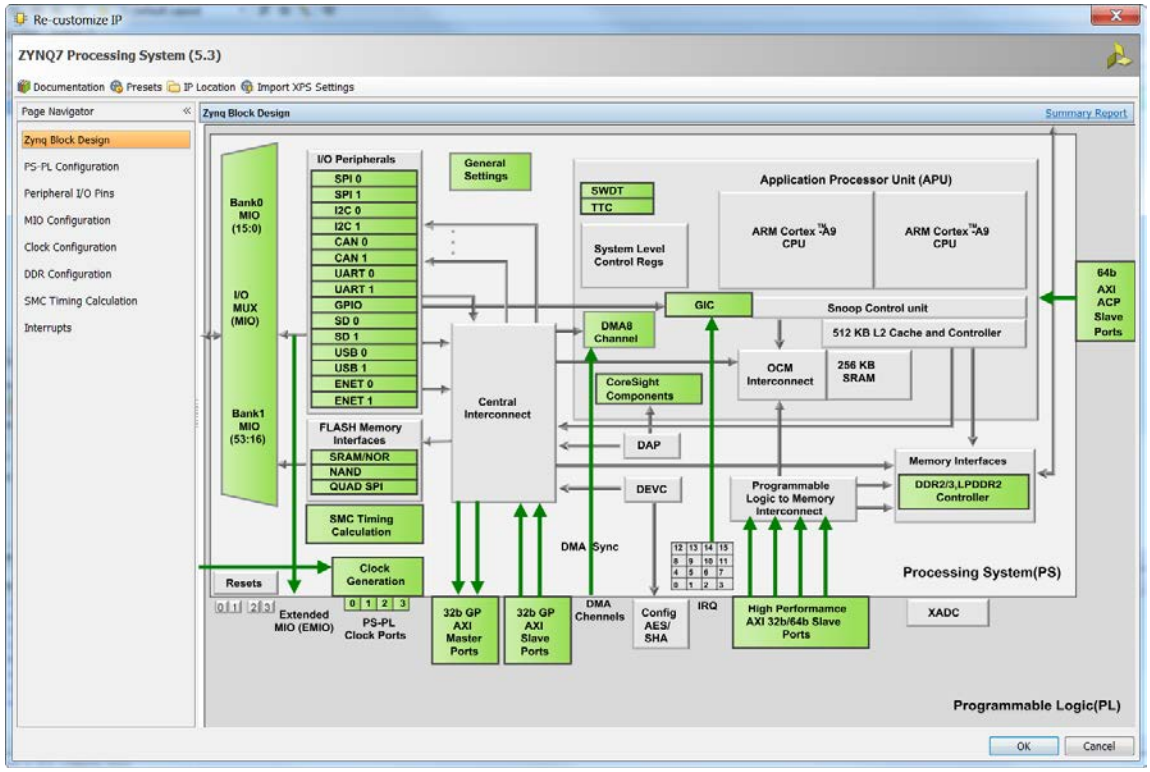



**Figure 5. Default configuration form with no peripheral selected**

**2-1-7.** Click on the **Import XPS Settings** button on the top tools bar to import the setting for the ZYBO board using the provided xml file.

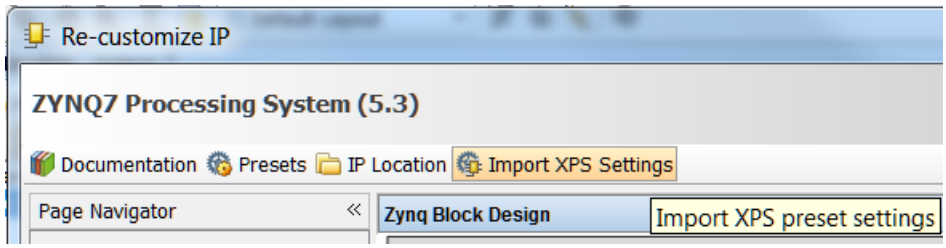



**Figure 6. Importing xml file for the ZYBO board**

**2-1-8.** Click on the browse button, browse to c:\xup\sys_design\sources\lab3, select the provided *ps7_system_prj.xml* file, and click **OK**.

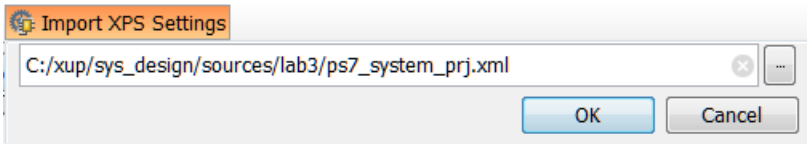



**Figure 7. Selecting the xml file**

**2-1-9.** Click **OK**.

Notice now the *Customization* window shows selected peripherals (with tick marks).

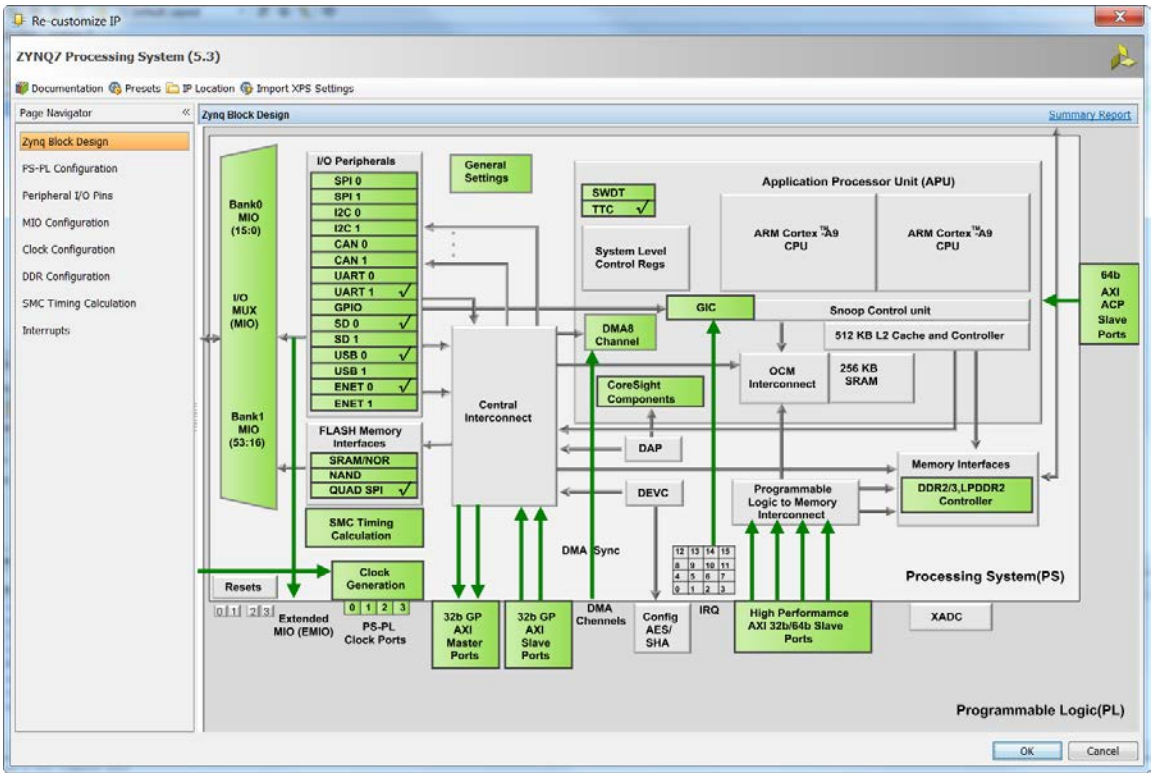



**Figure 8. Imported peripherals settings**

**2-1-10.** Click **OK** to close the *Customization* window for now.

## 2-2. Configure the processing block with just UART 1 peripheral enabled.

**2-2-1.** Click on **Run Block Automation** and select **/processing_system7_1**


**Figure 9. Designer Assistance message**

**2-2-2.** Click **OK** when prompted to run automation

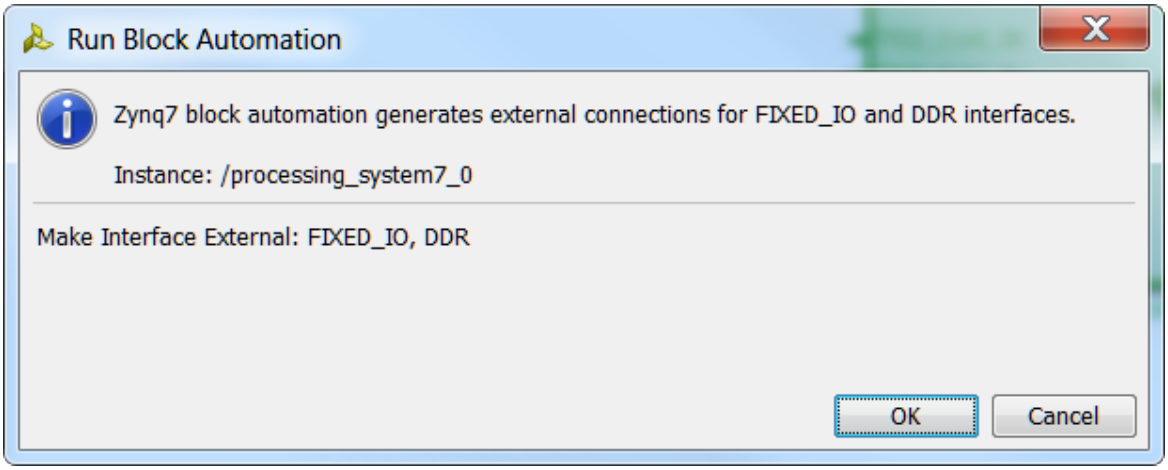



**Figure 10. Run Block Automation**

Once Block Automation has been complete, notice that ports have been automatically added for the DDR and Fixed IO, and some additional ports are now visible. The imported configuration for the Zynq related to the ZYBO board has been applied which will now be modified.

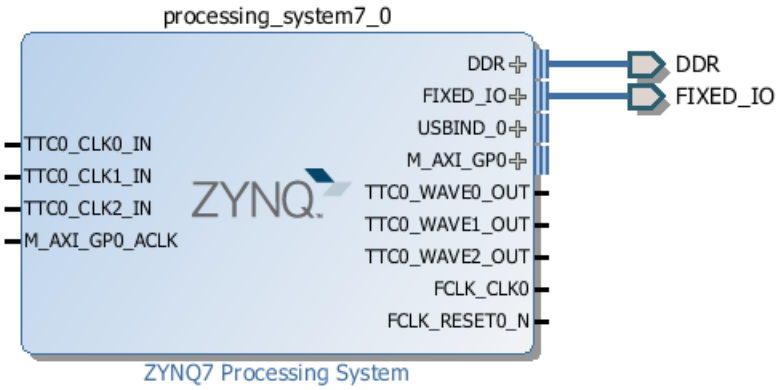



**Figure 11. Zynq Block with DDR and Fixed IO ports**

**2-2-3.** In the block diagram, double click on the *Zynq* block to open the *Customization* window for the Zynq processing system.

**2-2-4.** A block diagram of the Zynq should now be open, showing various configurable blocks of the Processing System.

At this stage, the designer can click on various configurable blocks (highlighted in green) and change the system configuration.

Only the UART is required for this lab, so all other peripherals will be deselected.

**2-2-5.** Click on one of the peripherals (in green) in the *IOP Peripherals* block, or select the *MIO Configuration* tab on the left to open the configuration form

**2-2-6.** Expand I/O peripherals if necessary, and deselect all the *I/O peripherals* except *UART 1.*
i.e. Remove: *ENET 0*
*USB 0*
*SD 0*

Expand **Memory Interfaces** to deselect *Quad SPI Flash*
Expand **Application Processor Unit** to disable *Timer 0*.

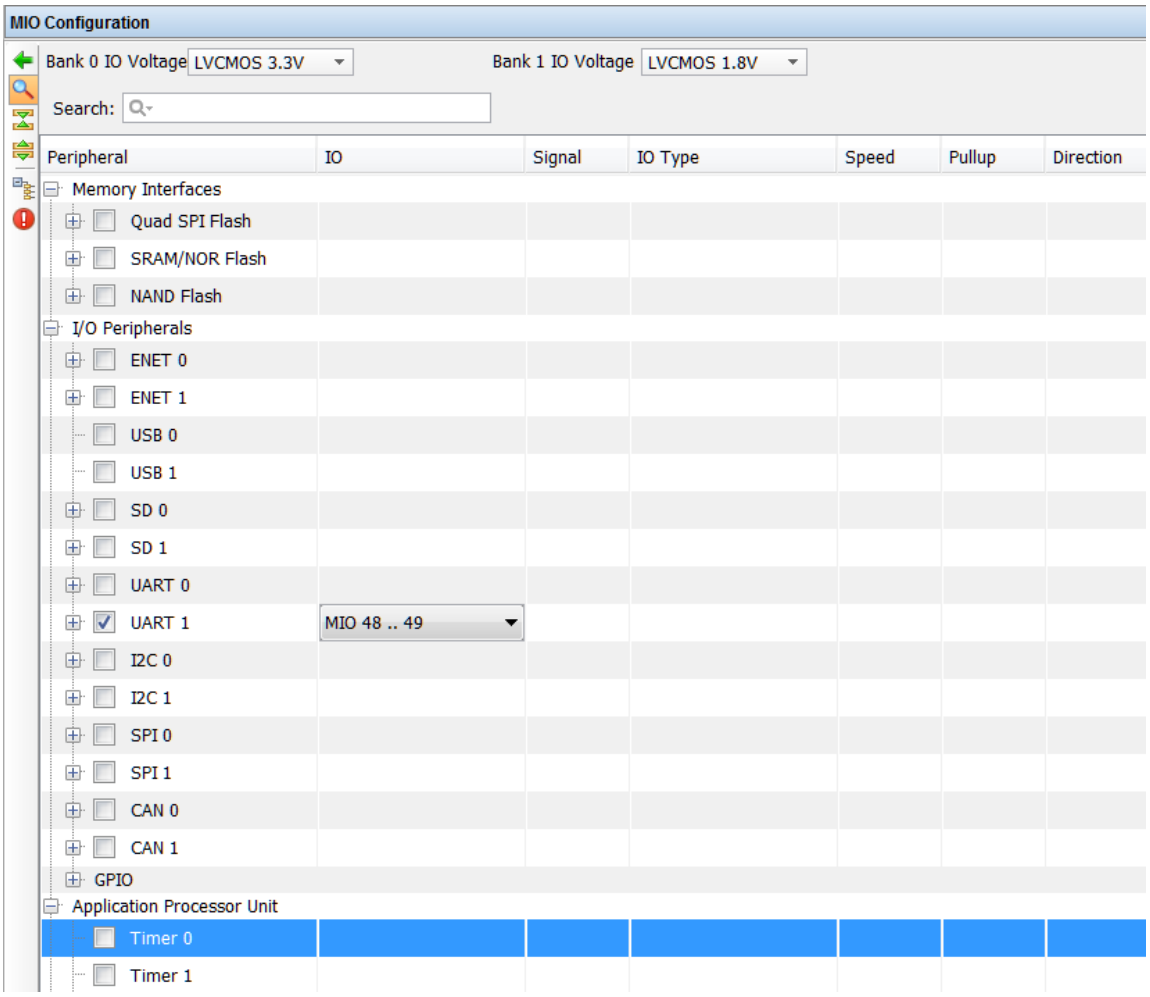



**Figure 12. Selecting only UART 1**

**2-2-7.** Click on the *Clock Configuration*, expand *PL Fabric Clocks*, and observe that **FCLK_CLK0** is enabled with **100** MHz frequency.

**2-2-8.** Click **OK**.

We left the rest of the configuration as is since we want to add two GPIO peripherals in the PL section which will be connected through the GP0 master interface, using FCLK_CLK0 as the clock source and FCLK_RESET0_N as the reset control signal.

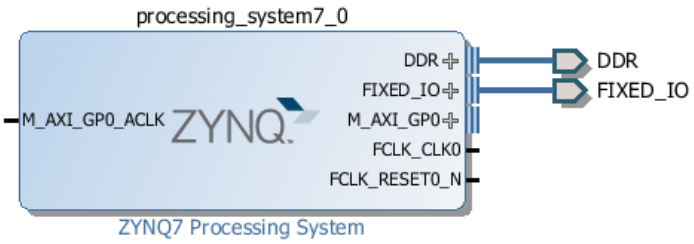



**Figure 13. Updated Zynq Block**

## Add Two Instances of GPIO Step 3

### 3-1. Add two instances of the GPIO Peripheral from the IP catalog to the processor system.

**3-1-1.** Click the Add IP icon and search for “gp”.

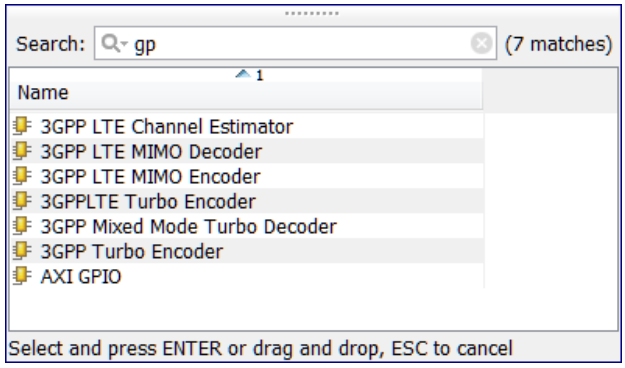



**Figure 14. Add GPIO IP**

**3-1-2.** Double-click the **AXI GPIO** to add the core to the design. The core will be added to the design and the block diagram will be updated.


**Figure 15. Zynq system with AXI GPIO added**

**3-1-3.** Click on the **AXI GPIO** block to select it, and in the properties tab, change the name to **sw_4bit**

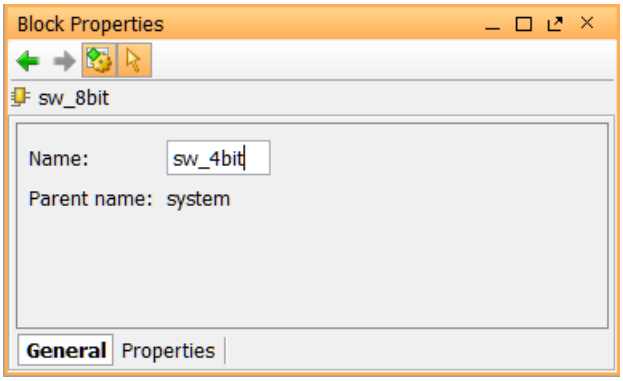



**Figure 16. Change AXI GPIO default name**

**3-1-4.** Double click on the **AXI GPIO** block to open the customization window.

If during the project creation, the target board had been selected then Vivado would have knowledge of available resources on the board. You could then have selected peripherals and generated the corresponding constraints based on the selected board. The GUI would have looked similar to the figure shown below.

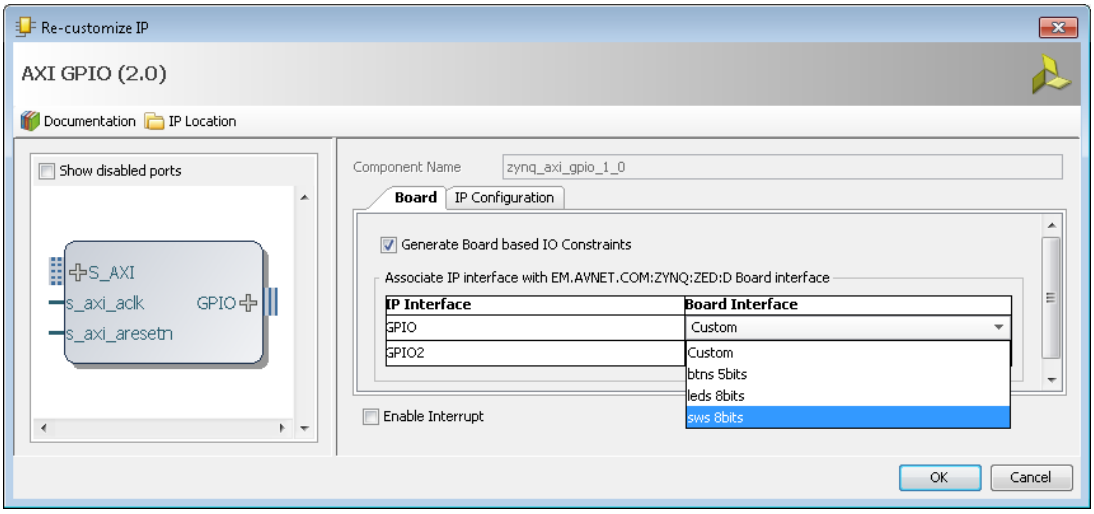



**Figure 17. Configuring GPIO instance**

Since the ZYBO board was not selected during the project creation, the GPIO has to be configured and the constraints will have to be explicitly added.

**3-1-5.** Select the All Inputs option as the switches are input only type devices. Also, set the GPIO width to 4 as the ZYBO board has four switches.

Notice that the peripheral can be configured for two channels, but, since we want to use only one channel without interrupt, leave the *GPIO Supports Interrupts* and *Enable Channel 2 unchecked*.

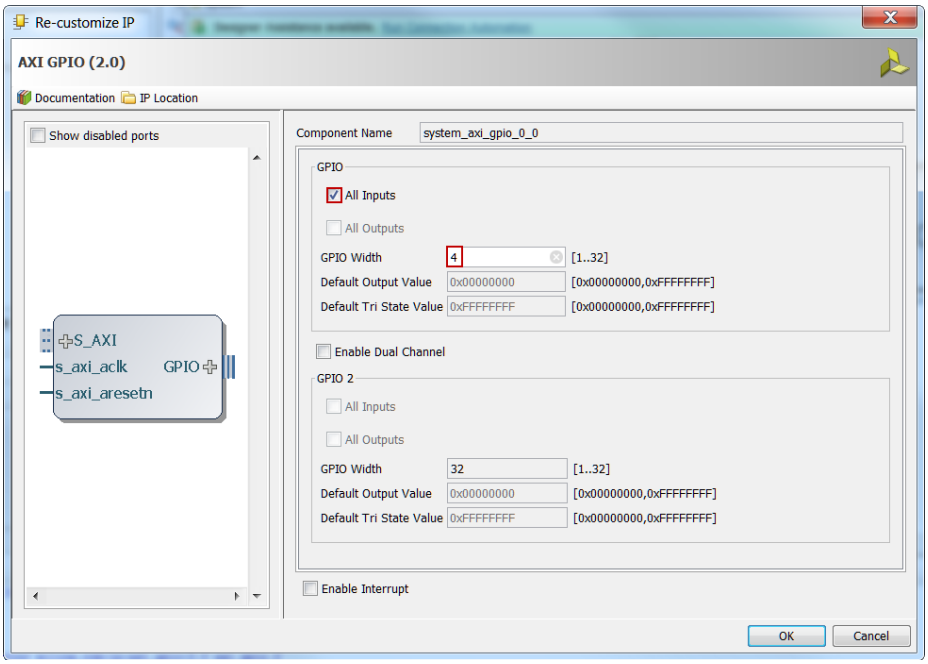



**Figure 18. Configuring GPIO instance**

**3-1-6.** Click **OK** to close the customization window

**3-1-7.** Notice that *Design assistance* is available. Click on **Run Connection Automation**, and select **/sw_4bit/S_AXI**

**3-1-8.** Click **OK** when prompted to automatically connect the master and slave interfaces

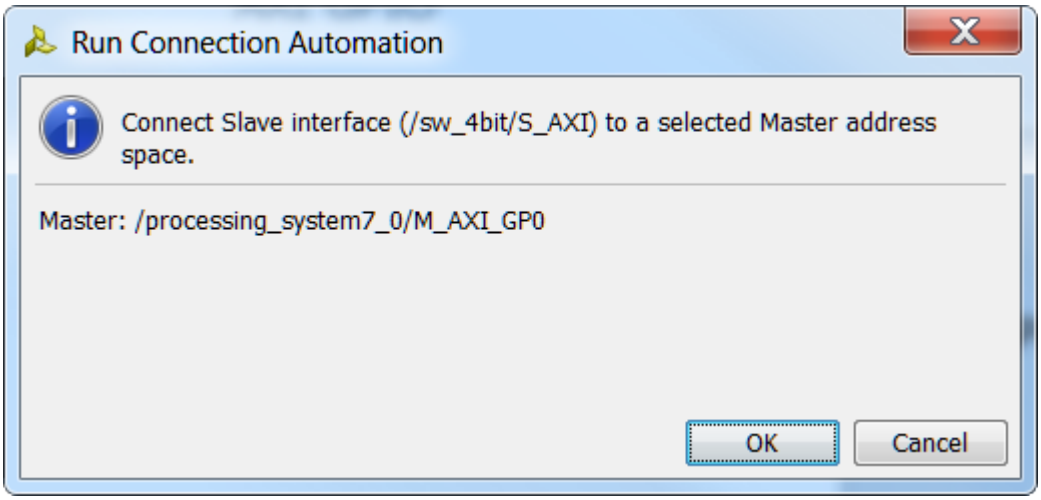



**Figure 19. Run connection automation**

**3-1-9.** Notice two additional blocks, *Processor System Reset*, and *AXI Interconnect* have automatically been added to the design. (The blocks can be dragged to be rearranged, or the design can be redrawn.)

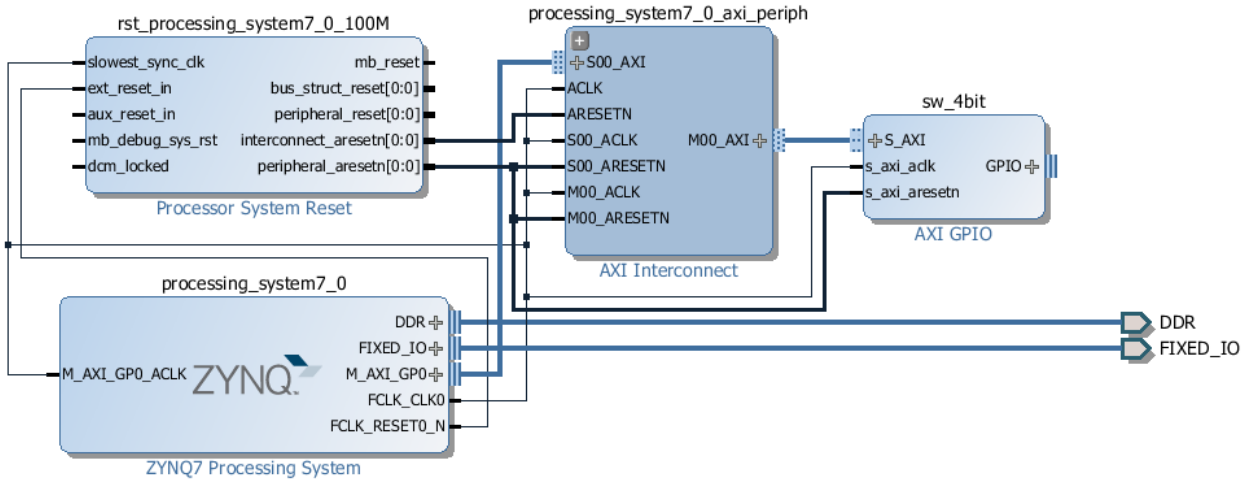



**Figure 20. Design with SW_4bit automatically connected**

**3-1-10.** Add another instance of the *GPIO* peripheral (**Add IP**). Name it as **btns_4bit**

**3-1-11.** Configure it to be all inputs with width of **4**.

At this point connection automation could be run, or the block could be connected manually. This time the block will be connected manually.

**3-1-12.** Double click on the AXI Interconnect and change the *Number of Master* Interfaces to **2** and click **OK**

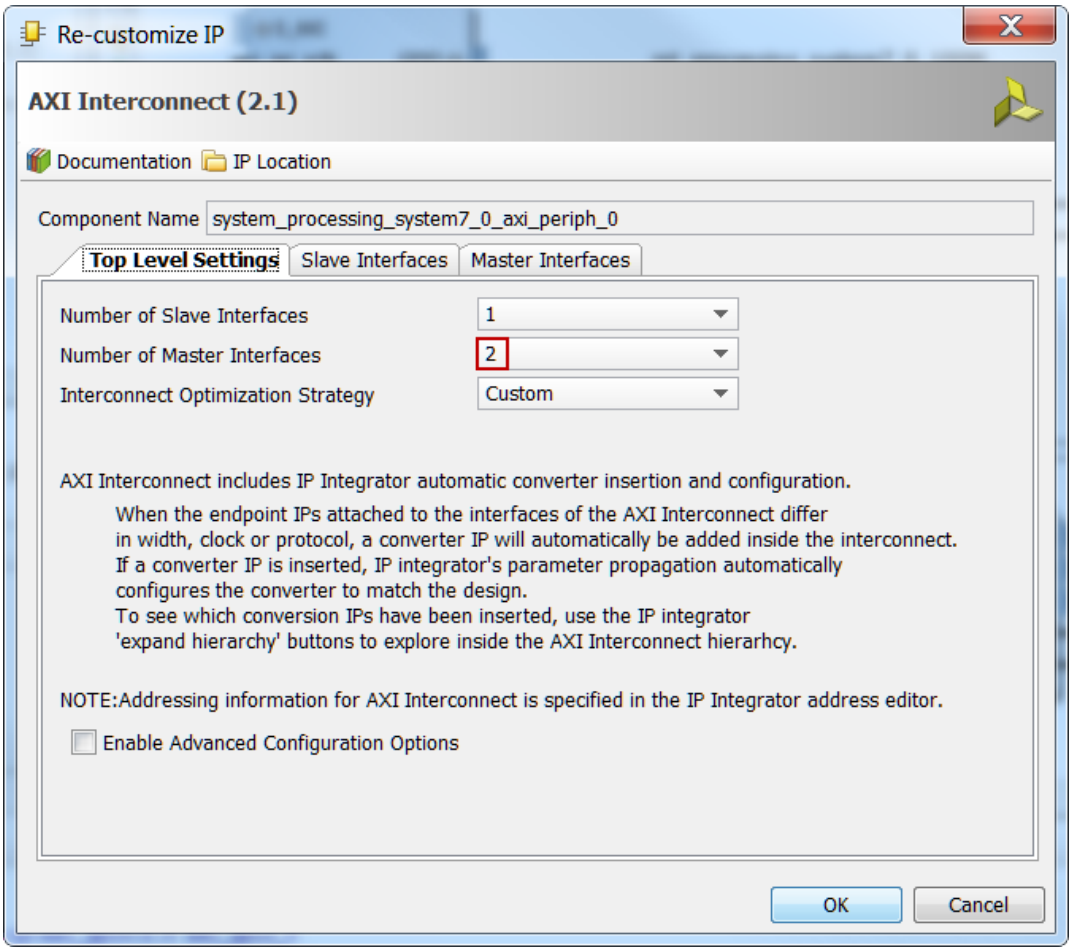



**Figure 21. Add master port to AXI Interconnect**

**3-1-13.** Click on the *s_axi* port of the new AXI GPIO block, and drag the pointer towards the AXI Interconnect block. The message *Found 1* interface should appear, and a green tick should appear beside the *M01_AXI* port on the AXI Interconnect indicating this is a valid port to connect to. Drag the pointer to this port and release the mouse button to make the connection.

**3-1-14.** In a similar way, connect the following ports:
*btns_4bit* **s_axi_aclk** -> *Zynq7 Processing System* **FCLK_CLK0**
*btns_4bit* **s_axi_aresetn** -> *Proc Sys Reset* **peripheral_aresetn**
*AXI Interconnect* **M01_ACLK** -> *Zynq7 Processing System* **FCLK_CLK0**
*AXI Interconnect* **M01_ARESETN** -> *Proc Sys Reset* **peripheral_aresetn**

The block diagram should look similar to this:

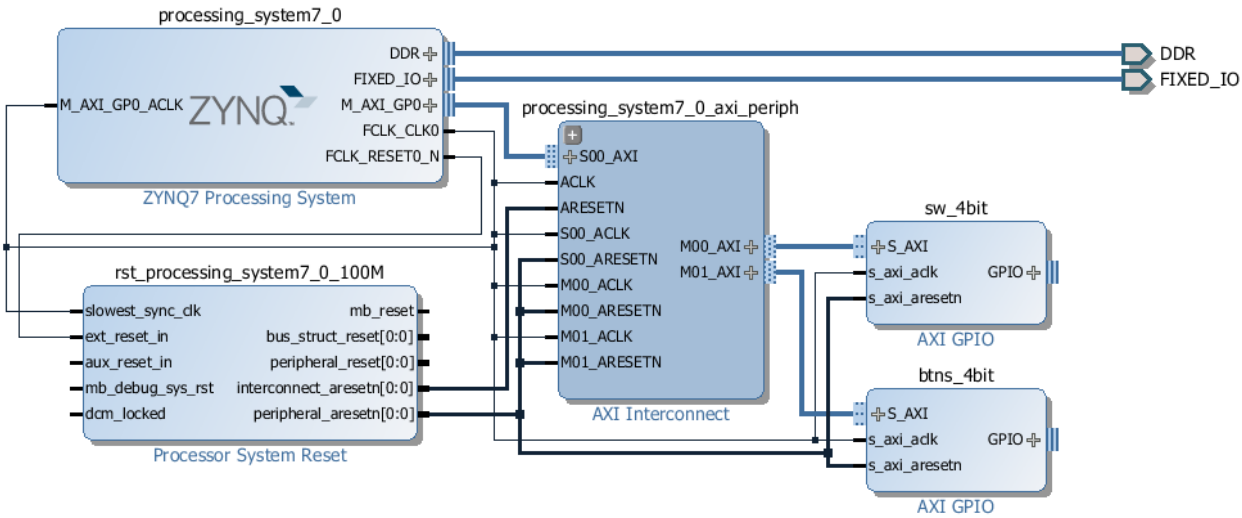



**Figure 22. System Assembly View after Adding the Peripherals**

**3-1-15.** Click on the *Address Editor* tab, and expand **processing_system7_0 > Data > Unmapped Slaves** if necessary

**3-1-16.** Notice that sw_4bit has been automatically assigned an address, but btns_4bit has not (since it was manually connected). Right click on *btns_4bit* and select **Assign Address** or click on the button.

Note that both peripherals are assigned in the address range of 0x40000000 to 0x7FFFFFFF (GP0 range).

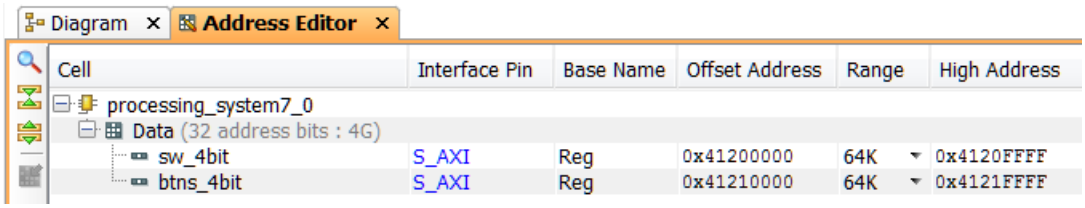


Diagram | Address Editor

| Cell | Interface Pin | Base Name | Offset Address | Range | High Address |
|---|---|---|---|---|---|
| processing_system7_0 | | | | | |
| Data (32 address bits : 4G) | | | | | |
| sw_4bit | S_AXI | Reg | 0x41200000 | 64K | 0x4120FFFF |
| btns_4bit | S_AXI | Reg | 0x41210000 | 64K | 0x4121FFFF |

**Figure 23. Peripherals Memory Map**

## Make GPIO Peripheral Connections External — Step 4

**4-1. The push button and dip switch instances will be connected to corresponding pins on the ZYBO board. This can be done manually, or using Designer Assistance. The location constraints will have to be explicitly added as the tools are not aware of the ZYBO board. You will add the provided constraints. Normally, one would consult the ZYBO board user manual to find this information.**

**4-1-1.** In the Diagram view, notice that *Designer Assistance* is available. Since the tools are not board aware we will ignore it, and we will manually create the ports and connect.

**4-1-2.** Right-Click on the gpio port of the *sw_4bit* instance and select **Make External** to create the external port. This will create the external port named *gpio* and connect it to the peripheral.

**4-1-3.** Select the *gpio* port and change the name to **sw_4bit** in its properties form.

The width of the interface will be automatically determined by the upstream block.

**4-1-4.** Similarly, create the external port on the btns_4bit block and naming it as **btns_4bit**

**4-1-5.** Run Design Validation (**Tools -> Validate Design**) and verify there are no errors.

The design should now look similar to the diagram below

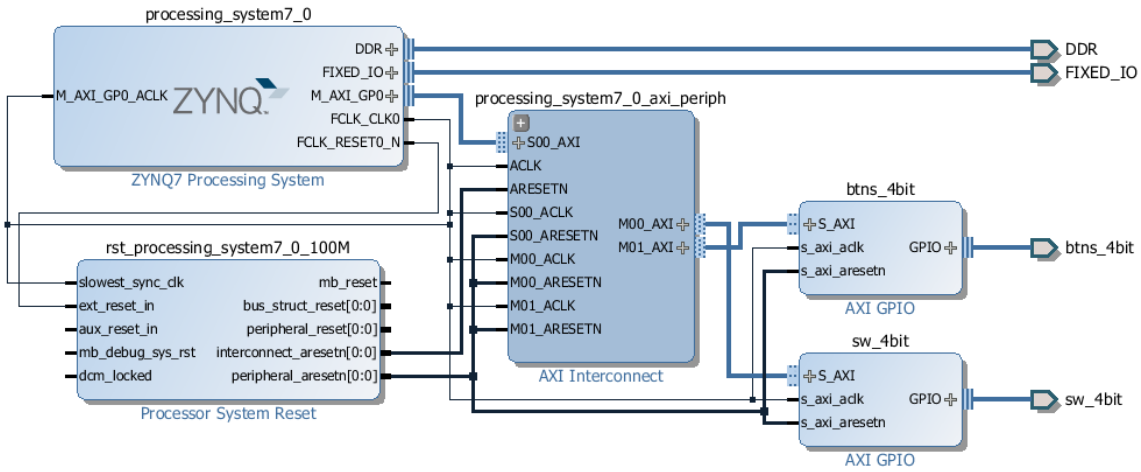



**Figure 24. Completed design**

**4-1-6.** In the *sources* view, Right Click on the block diagram file, **system.bd**, and select **Create HDL Wrapper** to update the HDL wrapper file. When prompted, click **OK.**

## 4-2. Add the provided lab3.xdc file that has constraints for sw_4bit, synthesize the design, open the I/O Planning layout, and add the btns_4bit constraints using the I/O planning tool.

**4-2-1.** In the Flow Navigator, click **Add Sources**

**4-2-2.** Select the *Add or Create Constraints* option and click **Next**.

**4-2-3.** Click on the **Add Files**… button, browse to **c:\xup\sys_design\sources\lab3** and select *lab3.xdc*

**4-2-4.** Click **OK** and then **Finish**

**4-2-5.** In the Sources window, notice the lab2.xdc entry under the constraints folder.

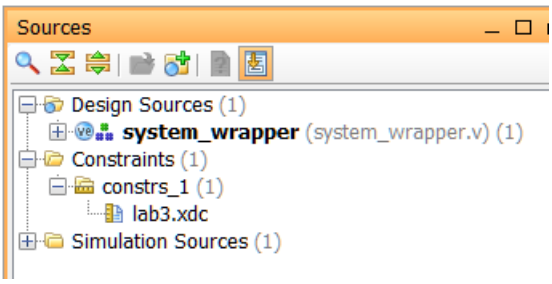



**Figure 25. The constraint file added**

**4-2-6.** In the Flow Navigator, click **Run Synthesis**. (Click **Save** when prompted) and when synthesis completes, select **Open Synthesized Design** and click **OK**

**4-2-7.** In the shortcut Bar, select **I/O Planning** from the *Layout* dropdown menu

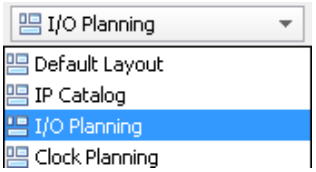



**Figure 26. Switch to the IO planning view**

**4-2-8.** In the I/O ports tab, expand *btns_4bit_tri_i*, and notice pins have not been assigned to this peripheral. The *sw_4bit_tri_i* have been assigned pin locations, along with the other Fixed ports in the design. Notice that the I/O Std of btns_4bit_tri_i have default to LVCMOS18 whereas sw_4bit_tri_i have been assigned LVCMOS33. Since they reside in the same bank, the two standards are conflicting with each other.

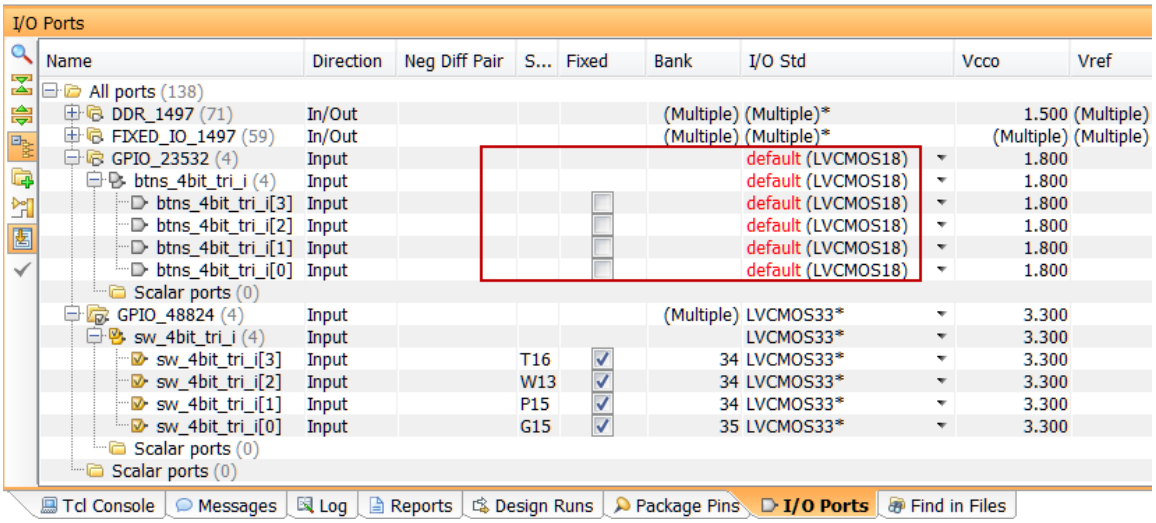



**Figure 27. The IP port pin constraints**

**4-2-9.** In the *I/O Ports* pane, click in the Site column of **btns_4bit_tri_i[3]** row, enter **Y16.**

**4-2-10.** Click in the *I/O Std* column of the same row.

Notice that LVCMOS18 is displayed in Red under the IO Standard column. It is shown in Red because of mismatch IO standard.

**4-2-11.** Click on the *LVCMOS18* entry to see a pop-up window. Scroll down and select **LVCMOS33** and then hit **Enter**. The LVCMOS33 is displayed in black.

**4-2-12.** Similarly, assign *V16, P16,* and *R18* to **btns_4bit_tri_i[2], btns_4bit_tri_i[1],** and **btns_4bit_tri_i[0]**. Assign LVCMOS33 standard too.

At this stage the I/O Ports tab should look like as shown below.

| | | | | | | |
|---|---|---|---|---|---|---|
| GPIO_23532 (4) | Input | | | | 34 | LVCMOS33* |
| btns_4bit_tri_i (4) | Input | | | | 34 | LVCMOS33* |
| btns_4bit_tri_i[3] | Input | | Y16 | ✓ | 34 | LVCMOS33* |
| btns_4bit_tri_i[2] | Input | | V16 | ✓ | 34 | LVCMOS33* |
| btns_4bit_tri_i[1] | Input | | P16 | ✓ | 34 | LVCMOS33* |
| btns_4bit_tri_i[0] | Input | | R18 | ✓ | 34 | LVCMOS33* |
| Scalar ports (0) | | | | | | |
| GPIO_48824 (4) | Input | | | | (Multiple) | LVCMOS33* |
| sw_4bit_tri_i (4) | Input | | | | | LVCMOS33* |
| sw_4bit_tri_i[3] | Input | | T16 | ✓ | 34 | LVCMOS33* |
| sw_4bit_tri_i[2] | Input | | W13 | ✓ | 34 | LVCMOS33* |
| sw_4bit_tri_i[1] | Input | | P15 | ✓ | 34 | LVCMOS33* |
| sw_4bit_tri_i[0] | Input | | G15 | ✓ | 35 | LVCMOS33* |
| Scalar ports (0) | | | | | | |

**Figure 28. After pin assignments done to the btns_4bit_tri_i ports**

**4-2-13.** Select **File > Save Constraints** and click **OK** when Out of Date Design warning message is displayed.

**4-2-14.** Click **OK** to save the constraints to *lab3.xdc* file.

## Generate the Bitstream and Export to SDK Step 5

### 5-1. Generate IP Integrator Outputs, the top-level HDL, and start SDK by exporting the hardware.

**5-1-1.** Click on the **Generate Block Design > system.bd** in the Flow Navigator, to generate the output products of various IPs added to the project. Click **Generate**.

**5-1-2.** Click on **Generate Bitstream**, and click **Yes** if prompted to Launch Implementation (Click **Yes** if prompted to save the design)

**5-1-3.** Select **Open Implemented Design** option when the bitstream generation process is complete and click **OK**. (Click **Yes** if prompted to close the synthesized design.)

**You should have the block design and the implemented design open (since we have a portion of the design in the PL section) before you export the hardware to SDK.**

**5-1-4.** Start SDK by clicking **File > Export > Export Hardware for SDK.**

**5-1-5.** The *Export Hardware for SDK* GUI will be displayed. Select the **Launch SDK** box, and ensure that *Export Hardware* is already selected, and click **OK** to export to, and launch SDK. (**Save** the design if prompted.)

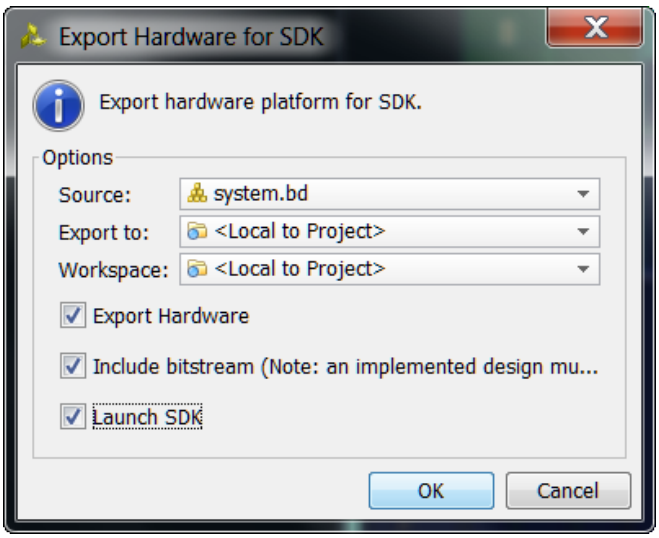
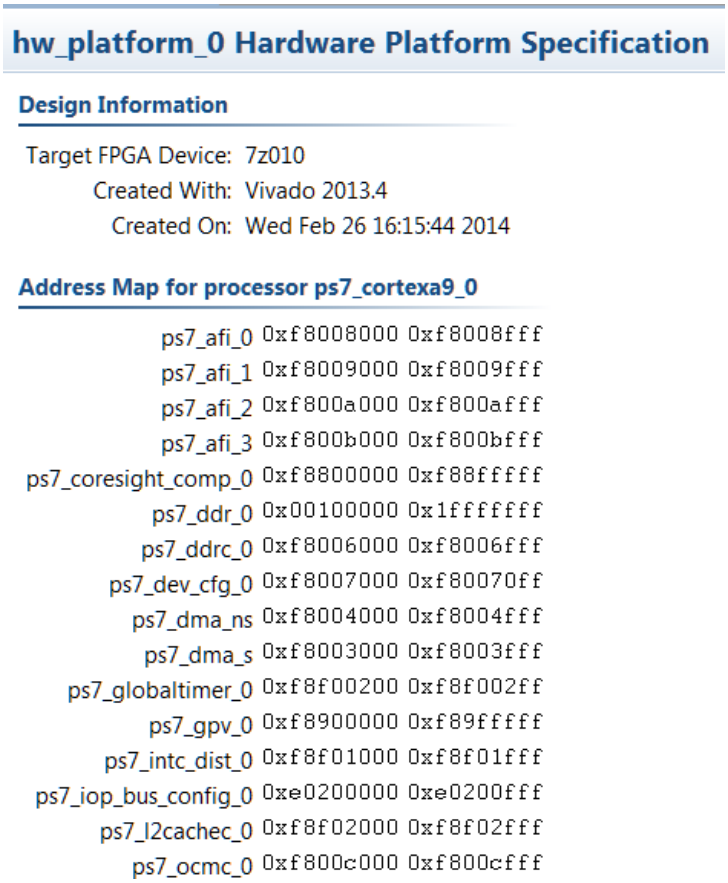



**Figure 29. Exporting to SDK**

SDK should now be open. If only the Welcome panel is visible, close or minimize this panel to view the *Project Explorer* and *Preview* panel. A Hardware platform project has been created, and the *hw_platform_0* folder should exist in the Project Explorer panel. The .xml file for the Hardware platform should be open in the preview pane. Double click system.xml to open it if it is not.

Basic information about the hardware configuration of the project can be found in the .xml file, along with the Address maps for the PS systems, and driver information.



**hw_platform_0 Hardware Platform Specification**

**Design Information**

Target FPGA Device: 7z010
Created With: Vivado 2013.4
Created On: Wed Feb 26 16:15:44 2014

**Address Map for processor ps7_cortexa9_0**

| | | |
|---|---|---|
| ps7_afi_0 | 0xf8008000 | 0xf8008fff |
| ps7_afi_1 | 0xf8009000 | 0xf8009fff |
| ps7_afi_2 | 0xf800a000 | 0xf800afff |
| ps7_afi_3 | 0xf800b000 | 0xf800bfff |
| ps7_coresight_comp_0 | 0xf8800000 | 0xf88fffff |
| ps7_ddr_0 | 0x00100000 | 0x1fffffff |
| ps7_ddrc_0 | 0xf8006000 | 0xf8006fff |
| ps7_dev_cfg_0 | 0xf8007000 | 0xf80070ff |
| ps7_dma_ns | 0xf8004000 | 0xf8004fff |
| ps7_dma_s | 0xf8003000 | 0xf8003fff |
| ps7_globaltimer_0 | 0xf8f00200 | 0xf8f002ff |
| ps7_gpv_0 | 0xf8900000 | 0xf89fffff |
| ps7_intc_dist_0 | 0xf8f01000 | 0xf8f01fff |
| ps7_iop_bus_config_0 | 0xe0200000 | 0xe0200fff |
| ps7_l2cachec_0 | 0xf8f02000 | 0xf8f02fff |
| ps7_ocmc_0 | 0xf800c000 | 0xf800cfff |

**Figure 30. SDK C/C++ development view**

## Generate TestApp Application in SDK Step 6

### 6-1. Generate a software platform project with default settings and default software project name. Create an application with the provided source file.

**6-1-1.** From the *File* menu select **File** > **New** > **Board Support Package**

**6-1-2.** Click **Finish** with the *standalone* OS selected.

**6-1-3.** Click **OK** to generate the board support package named **standalone_bsp_0**.

**6-1-4.** From the *File* menu select **File** > **New** > **Application Project**

**6-1-5.** Name the project **TestApp** and in the *Board Support Package* section, select *Use existing* to select **standalone_bsp_0,** and click **Next**

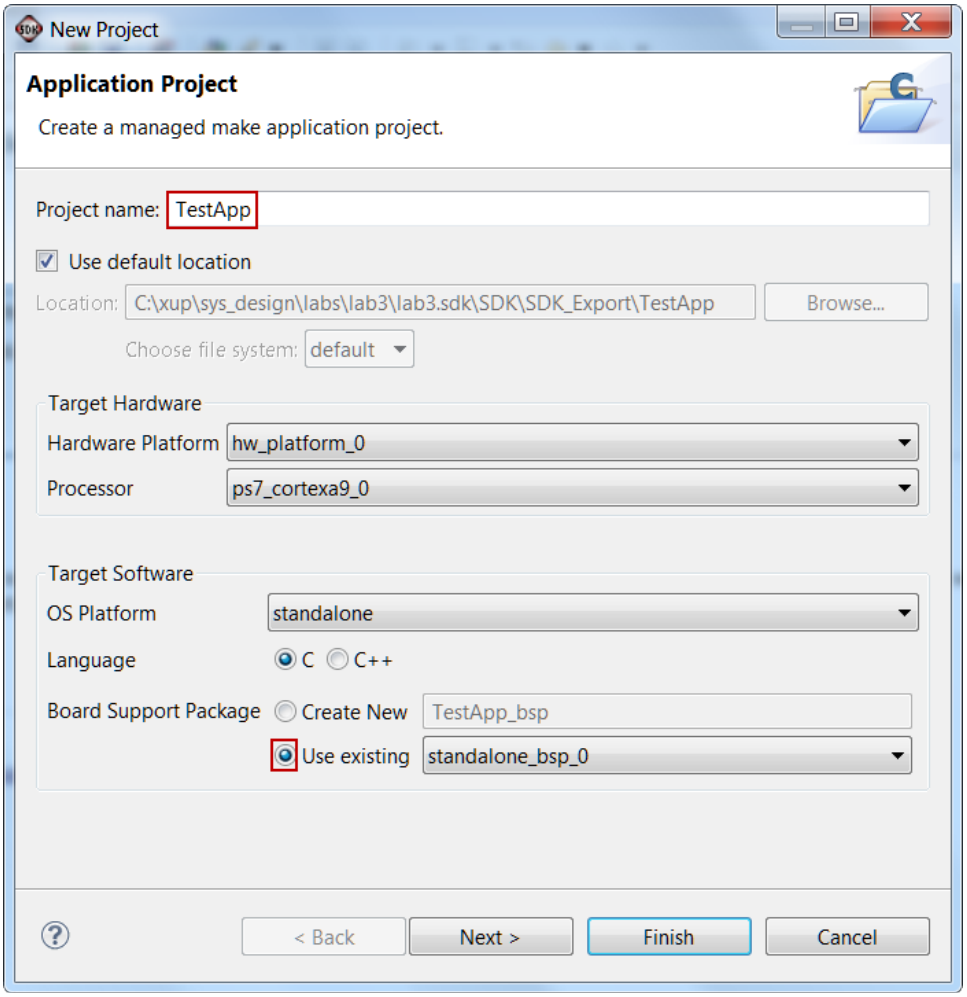



**Figure 31. Application project settings**

**6-1-6.** Select **Empty Application** and click **Finish**

This will create a new Application project, and a new Board Support Package Project

**6-1-7.** The library generator will run in the background and will create the **xparameters.h** file in the **C:\xup\sys_design\labs\lab3\lab3.sdk\SDK\SDK_Export\standalone_bsp_0\ps7_cortexa9_0\include** directory

**6-1-8.** Expand **TestApp** in the project view, and right-click on the **src** folder, and select **Import**

**6-1-9.** Expand **General** category and double-click on **File System**

**6-1-10.** Browse to **c:\xup\sys_design\sources\lab3** folder.

**6-1-11.** Select **lab3.c** and click **Finish.**

A snippet of the source code is shown in figure below.

```
#include "xparameters.h"
#include "xgpio.h"
#include "xutil.h"


//====================================================

int main (void)
{

    XGpio dip, push;
    int i, psb_check, dip_check;

    xil_printf("-- Start of the Program --\r\n");

    XGpio_Initialize(&dip, XPAR_SW_4BIT_DEVICE_ID);
    XGpio_SetDataDirection(&dip, 1, 0xffffffff);

    XGpio_Initialize(&push, XPAR_BTNS_4BIT_DEVICE_ID);
    XGpio_SetDataDirection(&push, 1, 0xffffffff);

    while (1)
    {
      psb_check = XGpio_DiscreteRead(&push, 1);
      xil_printf("Push Buttons Status %x\r\n", psb_check);
      dip_check = XGpio_DiscreteRead(&dip, 1);
      xil_printf("DIP Switch Status %x\r\n", dip_check);

      for (i=0; i<9999999; i++);
    }

}
```

**Figure 32. Snippet of source code**

## Test in Hardware Step 7

### 7-1. Make sure that the JP7 is set to select USB power. Connect the board with a micro-usb cable and power it ON. Establish the serial communication using SDK's Terminal tab.

**7-1-1.** Make sure that the JP7 is set to select USB power.

**7-1-2.** Make sure that a micro-USB cable is connected to the JTAG PROG connector (next to the power supply connector). Turn ON the power.

**7-1-3.** Select the Terminal tab. If it is not visible then select **Window > Show view > Terminal**.

**7-1-4.** Click on and if required, select appropriate COM port (depends on your computer), and configure it with the parameters as shown. (These settings may have been saved from previous lab).

### 7-2. Program the FPGA by selecting Xilinx Tools > Program FPGA and assigning system.bit file. Run the TestApp application and verify the functionality.

**7-2-1.** Select **Xilinx Tools** > **Program FPGA**

**7-2-2.** Make sure that the *system_wrapper.bit* is selected, and click **OK**.

**7-2-3.** Click **Program** to download the hardware bitstream. When FPGA is programmed, the DONE LED (green color) will be lit.

**7-2-4.** Select **TestApp** in *Project Explorer*, right-click and select **Run As > Launch on Hardware (GDB)** to download the application, execute ps7_init, and execute TestApp.elf.

**7-2-5.** You should see the something similar to the following output on Terminal console.

```
DIP Switch Status 6
Push Buttons Status 8
DIP Switch Status 6
Push Buttons Status 8
DIP Switch Status 6
Push Buttons Status 8
DIP Switch Status 6
Push Buttons Status 8
DIP Switch Status 6
Push Buttons Status 8
```

**Figure 33. SDK Terminal output**

**7-2-6.** Select *Console* tab and click on the *Terminate* button ( ) to stop the program.

**7-2-7.** Close SDK and Vivado programs by selecting **File > Exit** in each program.

**7-2-8.** Power OFF the board.

## Conclusion

Vivado and the IP Integrator allow base embedded processor systems and applications to be generated very quickly. GPIO peripherals were added from the IP catalog and connected to the Processing System through the 32b Master GP0 interface. The peripherals were configured and external FPGA connections were established. Pin location constraints were applied to connect the peripherals to the push buttons and DIP switches of the ZYBO. After the system has been defined, the hardware can be exported and SDK can be invoked from Vivado. Software development is done in SDK which provides several application templates including memory tests. You verified the operation of the hardware by downloading a test application, executing on the processor, and observing the output in the serial terminal window.

# Creating and Adding Custom IP to the System

## Introduction

This lab guides you through the process of creating and adding a custom peripheral to a processor system by using the Vivado IP Packager. You will create an AXI4Lite interface peripheral.

## Objectives

After completing this lab, you will be able to:

- Use the Create IP feature of Vivado to create a custom peripheral
- Modify the functionality of the IP
- Use the IP Packager feature of Vivado to package the custom peripheral
- Add the custom peripheral to your design
- Add pin location constraints

## Procedure

This lab is separated into steps that consist of general overview statements that provide information on the detailed instructions that follow. Follow these detailed instructions to progress through the lab.

This lab comprises 5 primary steps: You will use the Create IP feature to create a peripheral, then add the desire functionality and package the IP using IP Packager, import, add and connect the IP in the design, and generate a software project and verify the design in hardware.

## Design Description

You will extend the Lab 3 hardware design by creating and adding an AXI peripheral (refer to LED_IP in **Figure 1**) to the system, and connecting it to the LEDs on the ZYBO. You will use the IP Packager to generate the custom IP. Next, you will connect the peripheral to the system and add pin location constraints to connect the LED display controller peripheral to the on-board LED display. You will then generate the bitstream. The generated bitstream is then exported to SDK and then you will develop a software application and analyze the generated application's object code.

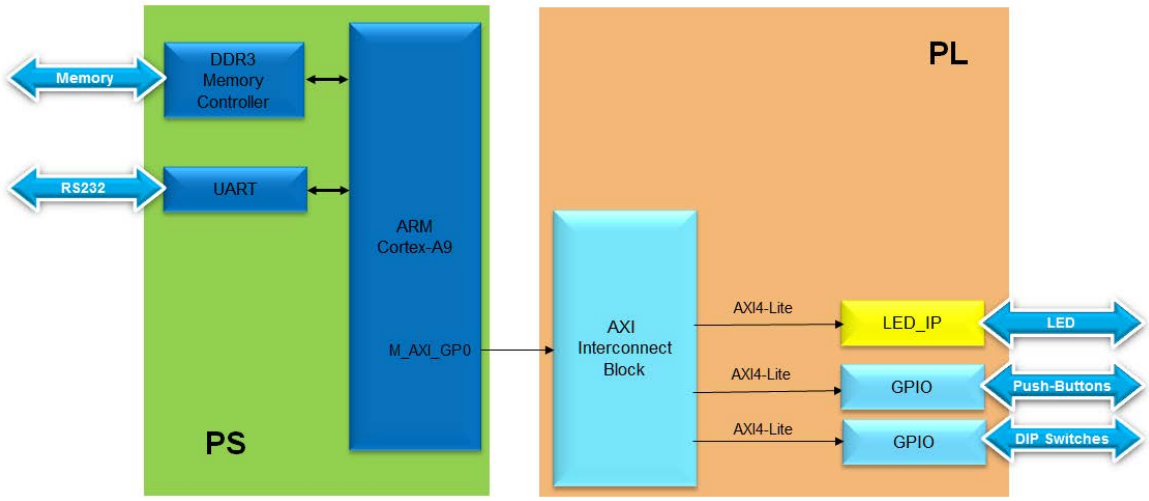
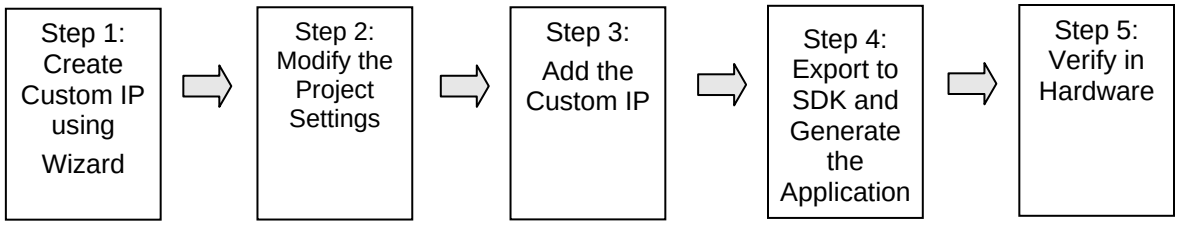



**Figure 1 Design Updated from Previous Lab**

## General Flow for this Lab




## Create a Custom IP using the Create and Package IP Wizard Step 1

### 1-1. Use the Manage IP project to start creating the custom IP.

**1-1-1.** Open Vivado by selecting **Start > All Programs > Xilinx Design Tools > Vivado 2013.4 > Vivado 2013.4**

**1-1-2.** Click **Manage IP** and select *New IP Location* and click **Next** in the *New IP Location* window

**1-1-3.** Select **Verilog** as the *Target Language*, **Mixed** as the *Simulator language*, and for *IP location*, type *C:\xup\sys_design\labs\led_ip* and click **Finish** (leave other settings as defaults)

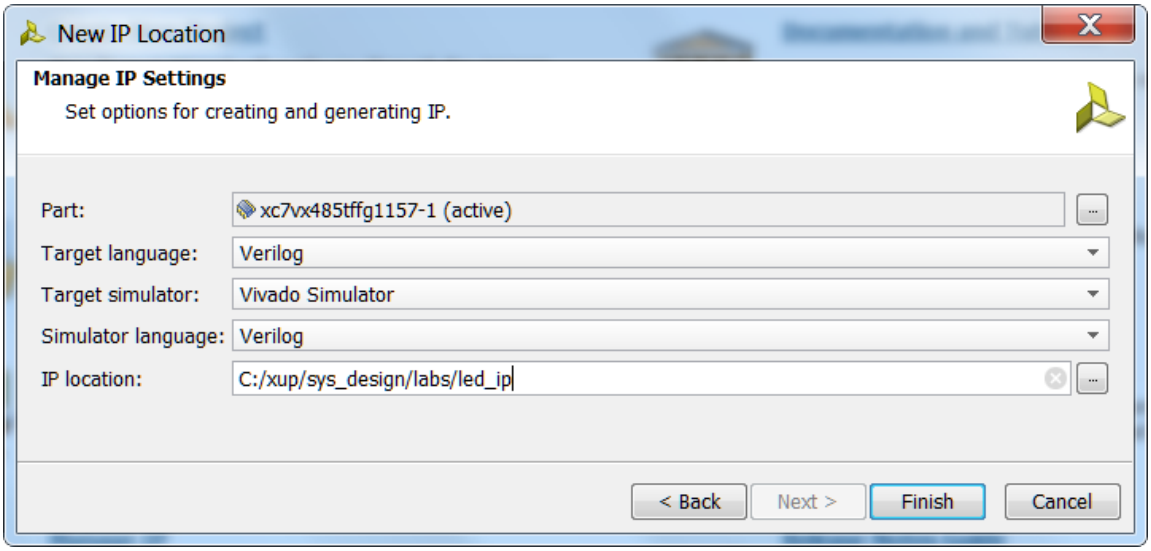



**Figure 2. New IP location**

A Virtex 7 part is chosen for this project, but later compatibility for other devices will be added to the packaged IP later.

**1-1-4.** Click **OK** to create the *led_ip* directory.

### 1-2. Run the Create and Package IP Wizard

**1-2-1.** Select *Tools > Create and Package IP…*

**1-2-2.** In the window, click **Next**

**1-2-3.** Select *Create New AXI4 Peripheral*, specify the *IP Definition location* as *C:/xup/sys_design/labs/led_ip* and click **Next**

**1-2-4.** Fill in the details for the IP

*Name:* **led_ip**
*Display Name*: **led_ip_v1_0**
(Fill in a description, Vendor Name, and URL)

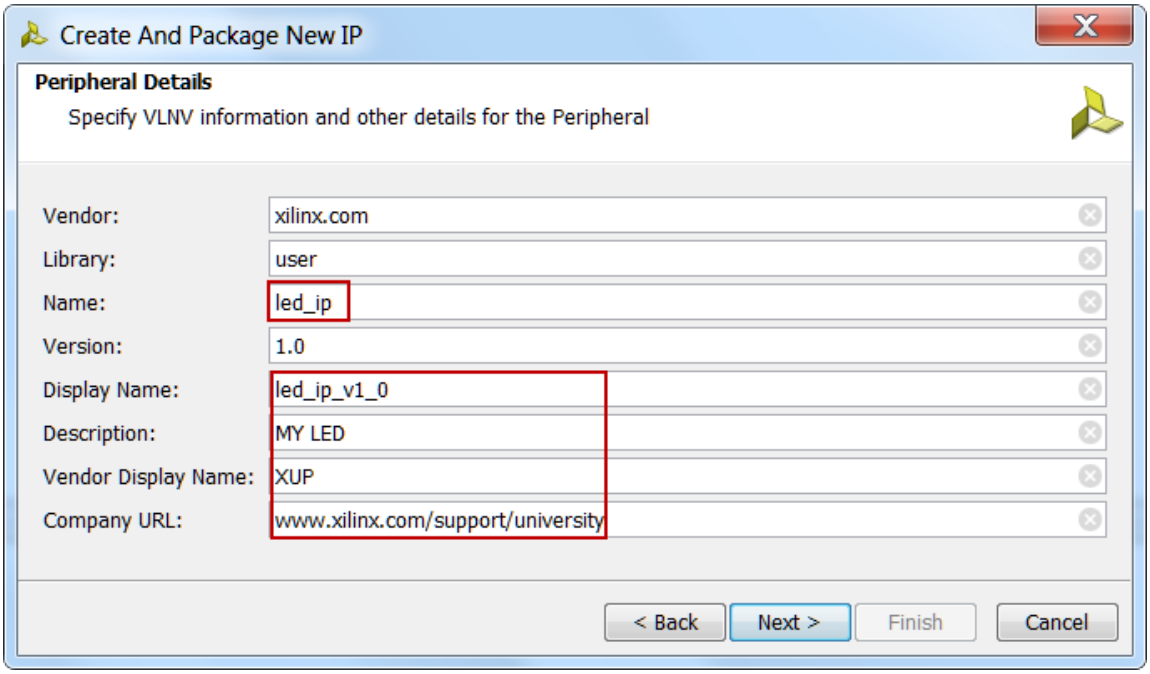



**Figure 3. Create and Package New IP**

**1-2-5.** Click **Next**

**1-2-6.** Change the Name of the interface to **S_AXI**

**1-2-7.** Leave the other settings as default and click **Next** (Lite interface, Slave mode, Data Width 32, Registers 4)

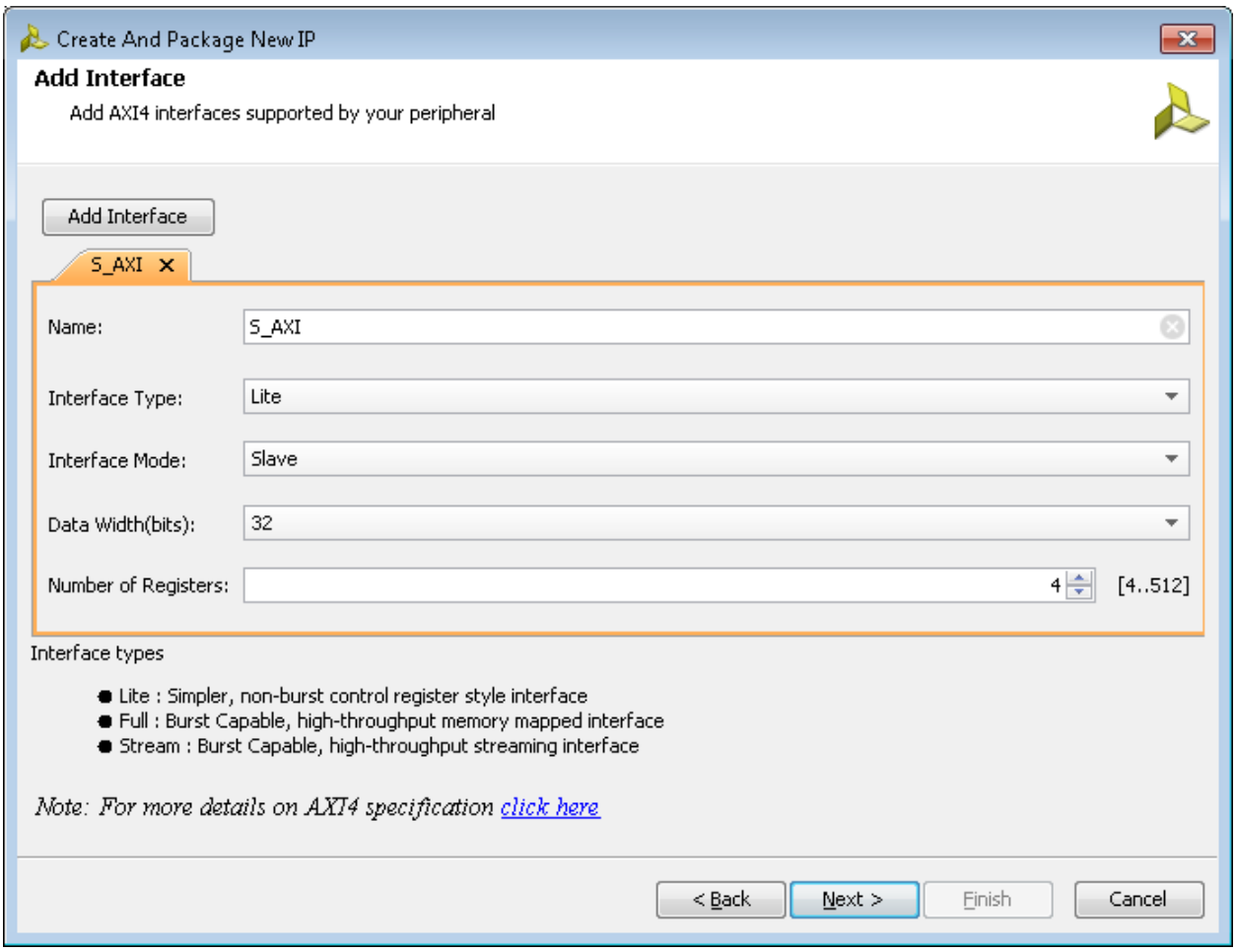



**Figure 4. AXI Interface settings**

**1-2-8.** Select *Generate Drivers* and click **Next**

**1-2-9.** Select *Add IP to Catalog and open IP in editing session* selected and click **Finish.**

## 1-3. Add files to the custom IP.

**1-3-1.** In the sources panel, double-click the **led_ip_v1_0.v** file and scroll down to line 15 where last port is defined.

**1-3-2.** Add the line:

**output wire [3:0] LED,**

(Notice the extra comma needed at the end of line)

```
14        // Users to add ports here
15        output wire [3:0] LED,
16        // User ports ends
17        // Do not modify the ports beyond this line
```

**1-3-3.** Insert the following at line ~48:

**. LED(LED),**

```
43 // Instantiation of Axi Bus Interface S_AXI
44     led_ip_v1_0_S_AXI # (
45         .C_S_AXI_DATA_WIDTH(C_S_AXI_DATA_WIDTH),
46         .C_S_AXI_ADDR_WIDTH(C_S_AXI_ADDR_WIDTH)
47     ) led_ip_v1_0_S_AXI_inst (
48         .LED(LED),
49         .S_AXI_ACLK(s_axi_aclk),
```

**1-3-4.** Save the file by selecting **File > Save File**

**1-3-5.** Expand *led_ip_v1_0* in the sources view if necessary, and open **led_ip_v1_0_S_AXI.v**

**1-3-6.** Add the LED port to this file too, at line 15

```
14         // Users to add ports here
15         output wire [3:0] LED,
16         // User ports ends
17         // Do not modify the ports beyond this line
```

**1-3-7.** Scroll down to ~line 397 and insert the following code to instantiate the user logic for the LED IP

(This code can be typed directly, or copied from the user_logic_instantiation.txt file in the lab4 source folder.)

```
397     lab4_user_logic U1(
398         .S_AXI_ACLK(S_AXI_ACLK),
399         .slv_reg_wren(slv_reg_wren),
400         .axi_awaddr(axi_awaddr[C_S_AXI_ADDR_WIDTH-1:ADDR_LSB]),
401         .S_AXI_WDATA(S_AXI_WDATA),
402         .S_AXI_ARESETN(S_AXI_ARESETN),
403         .LED(LED)
404     );
```

Check all the signals that are being connected and where they originate.

**1-3-8.** Save the file by selecting **File > Save File**

**1-3-9.** Click on the *Add Sources* in the Flow Navigator pane, select *Add or Create Design Sources*, browse to *c:\xup\sys_design\sources\lab4*, select the **lab4_user_logic.v** file and click **OK,** and then click **Finish** to add the file.

Check the contents of this file to understand the logic that is being implemented. Notice the formed hierarchy.

**1-3-10.** Click **Run Synthesis** and *Save* if prompted. (This is to check the design synthesizes correctly before adding it to the main design.

**1-3-11.** **Check the *Messages* tab for any errors and correct if necessary before moving to the next step**

When Synthesis completes successfully, click **Cancel**.

It is important that all source files that the IP will use be located under the IP project.

## 1-4. Update the IP project with the necessary source files in the appropriate directory

**1-4-1.** Expand the source hierarchy, if needed, and select lab4_user_logic entry.

**1-4-2.** Right-click and select **Remove File from the Project**.

**1-4-3.** Click OK to remove the file.

Similarly, remove all the files which were added from outside the project directory. In this lab, you don't have more files to be removed.

**1-4-4.** Using Windows Explorer, copy the *lab3_user_logic.v* file from the **c:\xup\sys_design\sources\lab4** directory and paste it in the current IP project directory at **c:\xup\sys_design\labs\led_ip\led_ip_1.0\hdl** directory.

**1-4-5.** Click on the *Add Sources* in the Flow Navigator pane, select *Add or Create Design Sources*, browse to *c:\xup\sys_design\labs\led_ip\led_ip_1.0\hdl*, select the **lab4_user_logic.v** file and click **OK,** and then click **Finish** to add the file.

Make sure that you select the file from the IP project directory and not the sources\lab3 directory.

## 1-5. Package the IP

**1-5-1.** Click on the **Package IP – led_ip** tab

**1-5-2.** For the IP to appear in the IP catalog in particular categories, the IP must be configured to be part of those categories. To change which categories the IP will appear in the IP catalog click the browse box on the *Categories* line. This opens the Choose IP Categories window

**1-5-3.** For the exercise purpose, uncheck the **AXI Peripheral** box and check the **Basic Elements** and click **OK**.

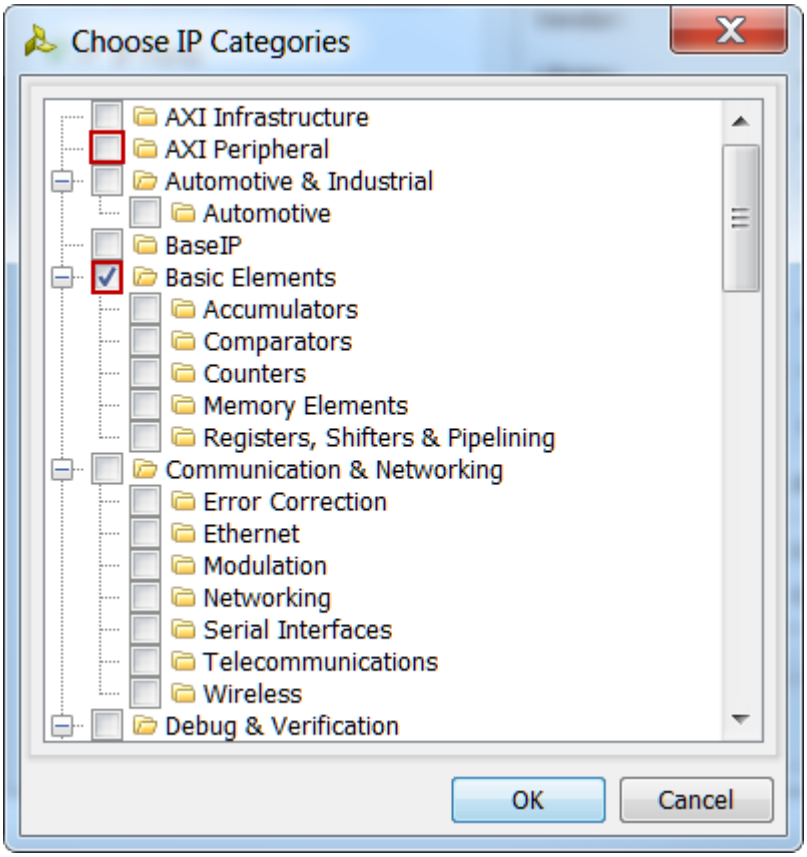



**Figure 5. Specify the category for IP Packager IP**

**1-5-4.** Select **IP Compatibility**. This shows the different Xilinx FPGA Families that the IP supports. The value is inherited from the device selected for the project.

**1-5-5.** Right click in the *Family Support table* and select **Add Family…** from the menu.

**1-5-6.** Select the **Zynq** family as we will be using this IP on the ZYBO, and click **OK**.

**1-5-7.** You can also customize the address space and add memory address space using the **IP Addressing and Memory** category. We won't make any changes.

**1-5-8.** Click on **IP File Groups** *and click Merge changes from IP File Groups Wizard*

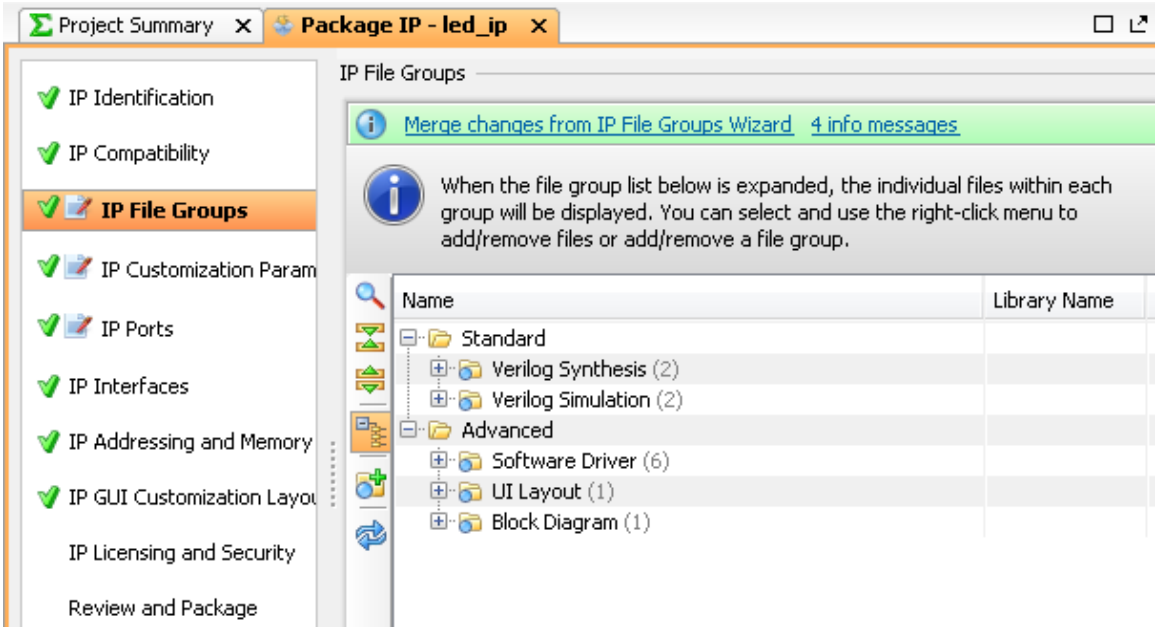



**Figure 6. IP File Groups**

This is to update the IP Packager with the changes that were made to the IP and the *lab4_user_logic.v* file that was added to the project.

**1-5-9.** Do the same for IP Customization Parameters (*Merge changes from IP Customization Parameters Wizard*)

Select the **IP Ports** view and notice that it shows the user created *LED* port

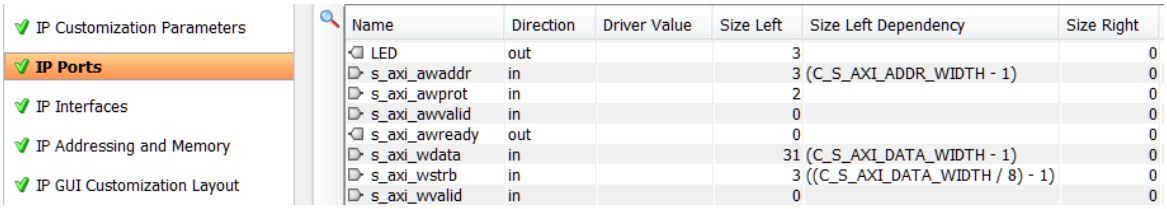



**Figure 7. IP Ports**

**1-5-10.** Select *Review and Package*, and click **Re-Package IP**

Click **OK** if a warning message is presented.

**1-5-11.** In the Vivado window click **File > Close Project**

## Modify the Project Settings Step 2

### 2-1. Open the previous project, or use the lab3 project from the labsolution directory, and save the project as lab4. Set Project Settings to point to the created IP repository.

**2-1-1.** Start Vivado if necessary and open either the lab3 project you created in the previous lab or the lab3 project in the labsolution directory

**2-1-2.** Select **File > Save Project As …** to open the *Save Project As* dialog box. Enter **lab4** as the project name. Make sure that the *Create Project Subdirectory* option is checked, the project directory path is c:\xup\sys_design\labs\ and click **OK**.

This will create the lab4 directory and save the project and associated directory with lab4 name.

**2-1-3.** Click **Project Settings** in the *Flow Navigator* pane.

**2-1-4.** Select **IP** in the left pane of the *Project Settings* form.

**2-1-5.** Click on the **Add Repository…** button, browse to **c:\xup\sys_design\labs\led_ip** and click **Select**.

The led_ip_v1_0 IP will appear the **IP in the Selected Repository** window.

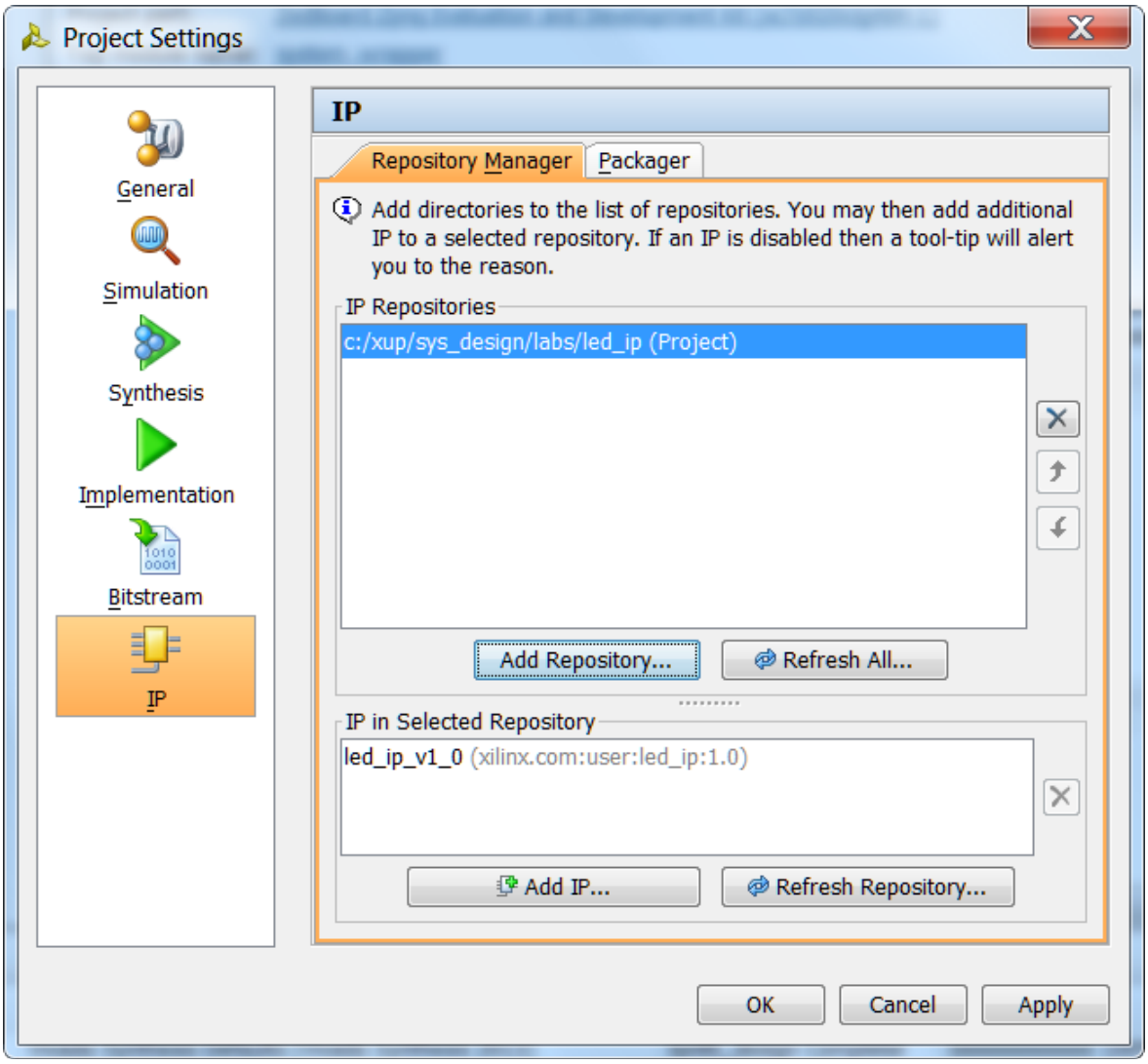



**Figure 8. Specify IP Repository**

**2-1-6.** Click **OK**.

## Add the Custom IP and the Constraints Step 3

### 3-1. Add led_ip to the design and connect to the AXI4Lite interconnect in the IP Integrator. Make internal and external port connections. Establish the LED port as external FPGA pins.

**3-1-1.** Click **Open Block Design** under **IP Integrator** in the Flow Navigator pane, and select **system.bd** to open IP Integrator

**3-1-2.** Click the Add IP icon and search for **led_ip_v1_0** in the catalog by typing “led” in the search field.

**3-1-3.** Double-click **led_ip_v1_0** to add the core to the design.

**3-1-4.** Select the IP in the block diagram and change the instance name to **led_ip** in the properties view

**3-1-5.** Click on **Run Connection Automation**, select **/led_ip/S_AXI** and click **OK** to automatically make the connection from the AXI Interconnect to the IP.

**3-1-6.** Click the regenerate button ( ) to redraw the diagram.

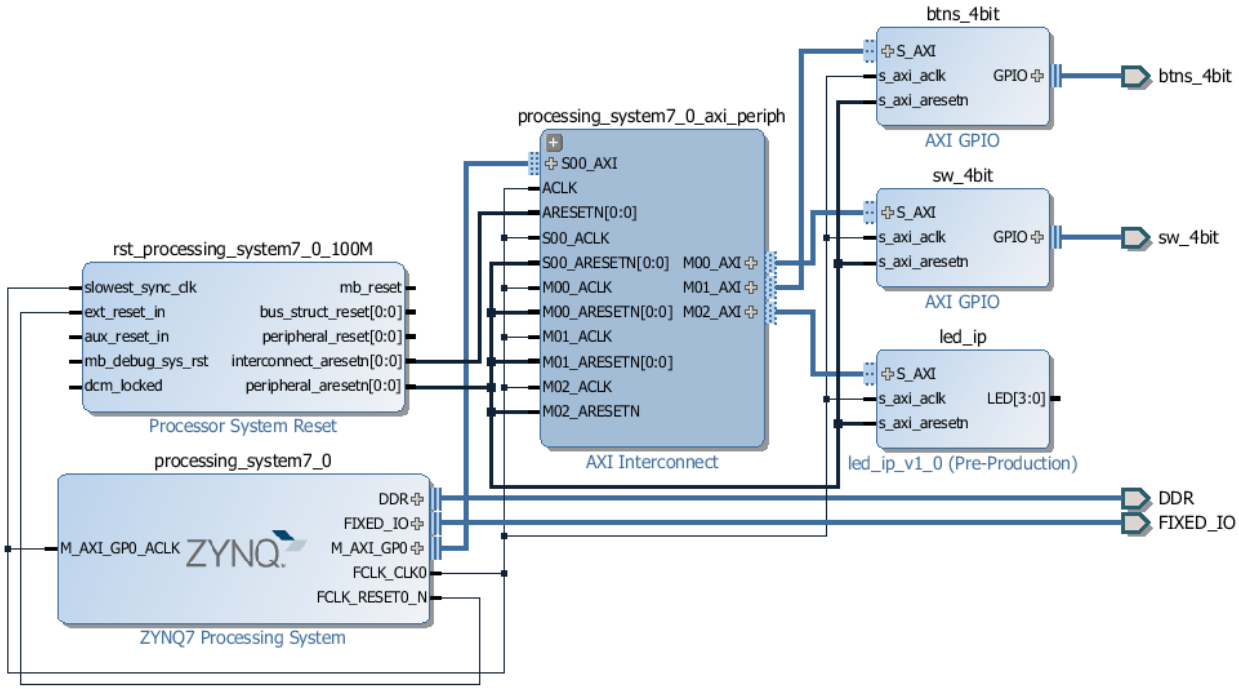



**Figure 9. LED IP Block added and connected**

**3-1-7.** Select the *LED[3:0]* port on the led_ip instance (by clicking on its pin), right-click and select **Make External**.

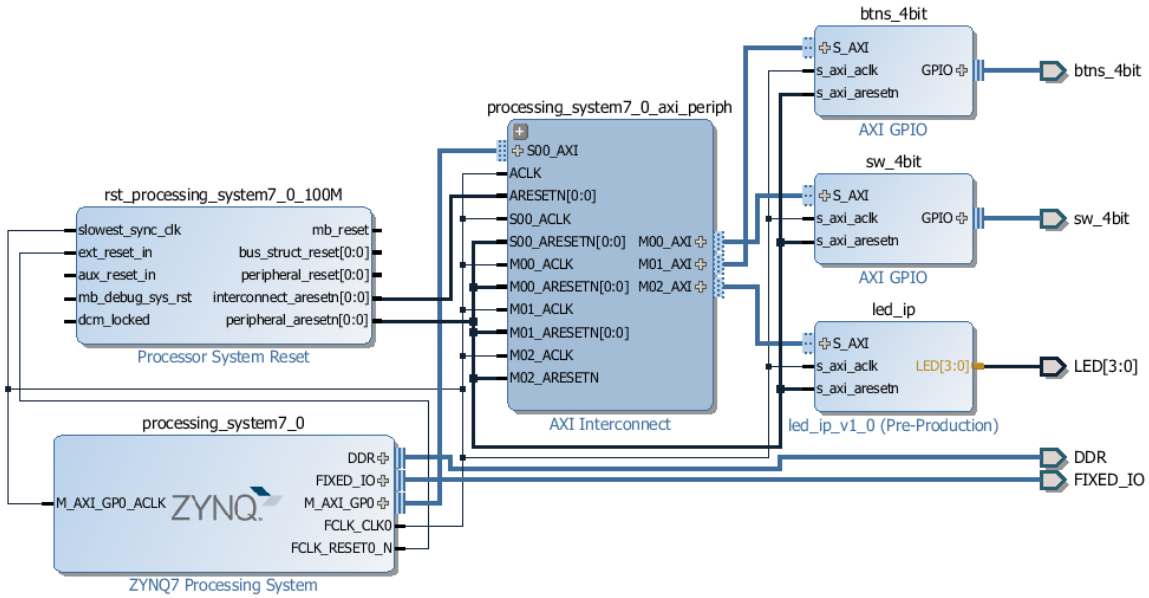



**Figure 10. LED external port added and connected**

**3-1-8.** Select the **Address Editor** tab and verify that an address has been assigned to lep_ip.

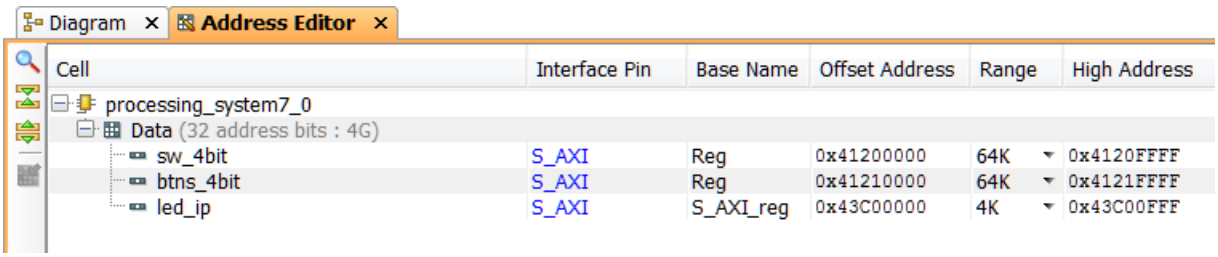


Diagram | Address Editor

| Cell | Interface Pin | Base Name | Offset Address | Range | High Address |
|---|---|---|---|---|---|
| processing_system7_0 | | | | | |
| Data (32 address bits : 4G) | | | | | |
| sw_4bit | S_AXI | Reg | 0x41200000 | 64K | 0x4120FFFF |
| btns_4bit | S_AXI | Reg | 0x41210000 | 64K | 0x4121FFFF |
| led_ip | S_AXI | S_AXI_reg | 0x43C00000 | 4K | 0x43C00FFF |

**Figure 11. Address assigned for led_ip**

**3-1-9.** Select **Tools > Validate Design** to run the design rules checker. There should not be any violations.

## 3-2. Update the top-level wrapper and add the provided lab4_system.xdc constraints file.

**3-2-1.** In the *sources* view, right-click on the block diagram file, **system.bd**, and select **Create HDL Wrapper** to update the HDL wrapper file, and when prompted, select *Let Vivado Manage Wrapper and auto-update* and click **OK**

The system_wrapper.vhd file will be updated to include the new IP and ports. Double-click on the system-wrapper content to verify that LED port has been added.

**3-2-2.** Click **Add Sources** in the *Flow Navigator* pane, select **Add or Create Constraints**, and click **Next**.

**3-2-3.** Click the **Add Files** button, browse to the **c:\xup\sys_design\sources\lab4** folder, select **lab4_system.xdc**

**3-2-4.** Click **Finish** to add the file.

**3-2-5.** Expand Constraints folder in the *Sources* pane, and double-click **lab4_system.xdc** file entry to see its content. This file contains the pin locations and IO standards for the LEDs on the ZYBO. This information can usually be found in the manufacturer's datasheet for the board.

**3-2-6.** Click on the **Generate Bitstream** in the Flow Navigator to run the synthesis, implementation, and bitstream generation processes. (Click **Save** if prompted.)

**3-2-7.** Click **Yes** to run the synthesis process again as the design has changed.

**3-2-8.** When the build completes, click **OK** if prompted to open the Implemented design.

## Export to SDK and create Application Project Step 4

### 4-1. Export the hardware along with the generated bitstream to SDK .

To Export the hardware, the block diagram must be open and the Implemented design must be open.

**4-1-1.** If it is not already open, click **Open Block Design** (under IP Integrator in the Flow Navigator)

**4-1-2.** Click **Open Implemented Design** (under Implementation)

**4-1-3.** Start *SDK* by clicking **File > Export > Export Hardware for SDK**...

The export GUI will be displayed.

**4-1-4.** Check all three checkboxes, including the **Launch SDK** box and click **OK**.

**4-1-5.** If prompted, click **YES** to overwrite the platform created by the previous lab

### 4-2. Create an empty project called lab4 using standalone_bsp software platform project. Import lab4.c file from the c:\xup\sys_design\sources directory

The hw_platform and bsp projects from the previous lab will automatically rebuild to include the led_ip. Verify this by checking the *system.mss* for the **led_ip** peripheral.

**<u>If you don't see it then close the SDK and re-export the design, invoking SDK again.</u>**

**4-2-1.** To tidy up the workspace and save unnecessary building of a project that is not being used, right click on the **TestApp** project from the previous lab, and click **Close Project**, as this project will not be used in this lab. It can be reopened later if needed.

**4-2-2.** Select **File > New** > **Application Project.**

**4-2-3.** Enter **lab4** as the *Project Name*, and for *Board Support Package*, choose **Use Existing (***standalone_bsp_0* should be the only option).

**4-2-4.** Click **Next**, and select *Empty Application* and click **Finish.**

**4-2-5.** Expand **lab4** in the project view, and right-click in the *src folder* and select **Import.**

**4-2-6.** Expand **General** category and double-click on **File System.**

**4-2-7.** Browse to **c:\xup\sys_design\sources\lab4** folder and click **OK.**

**4-2-8.** Select **lab4.c** and click **Finish** to add the file to the project (Ignore any errors for now).

**4-2-9.** Select the **system.mss** tab.

**4-2-10.** Click on **Documentation** link corresponding to **btns_4bit** peripheral under the Peripheral Drivers section to open the documentation in a default browser window. As our led_ip is very similar to GPIO, we look at the mentioned documentation.

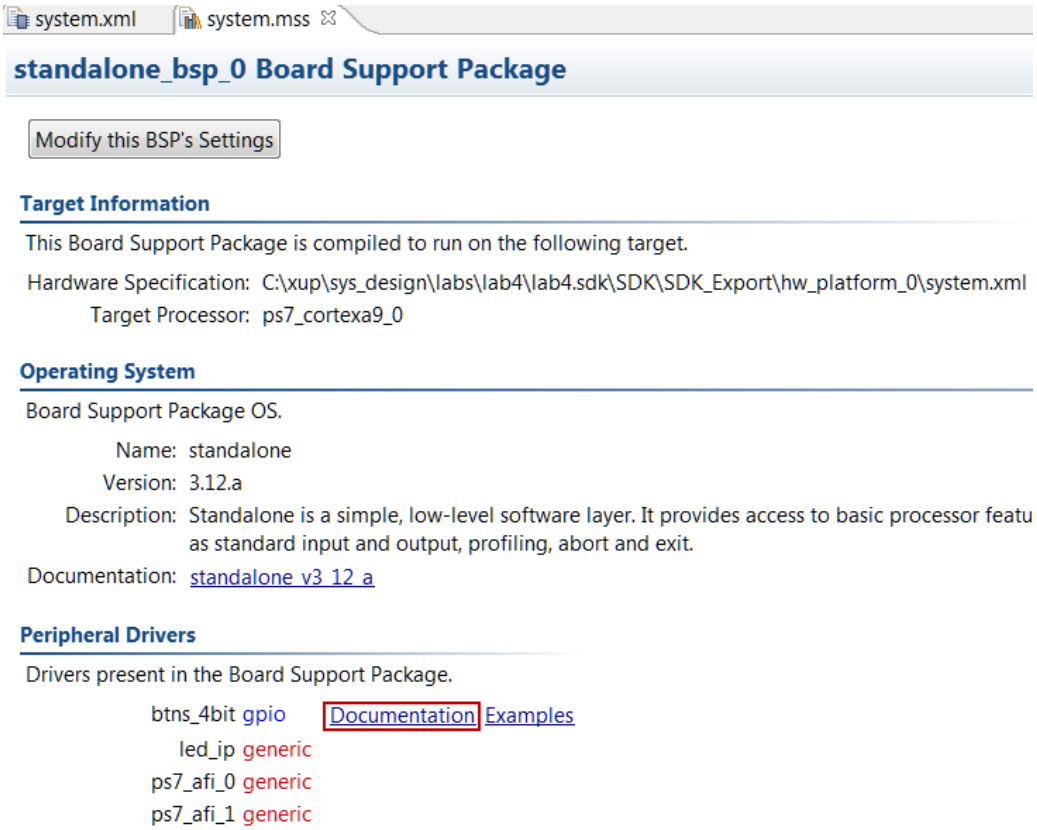



**Figure 11. Accessing device driver documentation**

**4-2-11.** View the various C and Header files associated with the GPIO by clicking **Files** at the top of the page.

**4-2-12.** Click the header file **xgpio.h** and review the list of available function calls for the GPIO.

The following steps must be performed in your software application to enable reading from the GPIO: **1) Initialize the GPIO, 2) Set data direction, and 3) Read the data**

Find the descriptions for the following functions by clicking links:

**XGpio_Initialize (XGpio *InstancePtr, u16 DeviceId)**

***InstancePtr*** is a pointer to an XGpio instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the XGpio API must be made with this pointer.

***DeviceId*** is the unique id of the device controlled by this XGpio component. Passing in a device id associates the generic XGpio instance to a specific device, as chosen by the caller or application developer.

**XGpio_SetDataDirection (XGpio * InstancePtr, unsigned Channel, u32 DirectionMask)**

***InstancePtr*** is a pointer to the XGpio instance to be worked on.

***Channel*** contains the channel of the GPIO (1 or 2) to operate on.

**DirectionMask** is a bitmask specifying which bits are inputs and which are outputs. Bits set to 0 are output and bits set to 1 are input.

**XGpio_DiscreteRead(XGpio *InstancePtr, unsigned channel)**

***InstancePtr*** is a pointer to the XGpio instance to be worked on.

***Channel*** *contain*s the channel of the GPIO (1 or 2) to operate on

**4-2-13.** Double-click on **lab4.c** in the Project Explorer view to open the file. This will populate the **Outline** tab. Open the header file **xparameters.h** by double-clicking on **xparameters.h** in the **Outline** tab

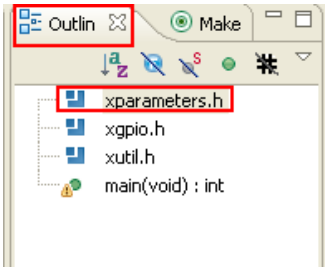



**Figure 12. Double-Click the generated header file**

The xparameters.h file contains the address map for peripherals in the system. This file is generated from the hardware platform description from Vivado. Find the following #define used to identify the **dip** peripheral:

#define XPAR_SW_4BIT_DEVICE_ID 1 **Note: The number might be different**

Notice the other #define XPAR_SW_4BIT* statements in this section for the 4 bit SW peripheral, and in particular the address of the peripheral defined by: `XPAR_SW_4BIT_BASEADDR`

**4-2-14.** Modify line 15 of lab4.c to use this macro (#define) in the *XGpio_Initialize* function.

```
#include "xparameters.h"
#include "xgpio.h"
#include "xutil.h"

//=====================================================

int main (void)
{

   XGpio dip, push;
   int i, psb_check, dip_check;

   xil_printf("-- Start of the Program --\r\n");

   XGpio_Initialize(&dip, XPAR_DIP_DEVICE_ID); // Modify this
   XGpio_SetDataDirection(&dip, 1, 0xffffffff);

   XGpio_Initialize(&push, XPAR_PUSH_DEVICE_ID); // Modify this
   XGpio_SetDataDirection(&push, 1, 0xffffffff);

   while (1)
   {
      psb_check = XGpio_DiscreteRead(&push, 1);
      xil_printf("Push Buttons Status %x\r\n", psb_check);
      dip_check = XGpio_DiscreteRead(&dip, 1);
      xil_printf("DIP Switch Status %x\r\n", dip_check);

      // output dip switches value on LED_ip device

      for (i=0; i<9999999; i++);
   }
}
```

**Figure 13. Highlighting the code to initialize the SW_4BIT as input, and read from it**

**4-2-15.** Do the same for to the BTNS_4BIT; find the macro (#define) for the BTNS_4BIT peripheral in xparameters.h, and modify line 18 in lab4.c, and save the file.

The project will be rebuilt. If there are any errors, check and fix your code. Your C code will eventually read the value of the switches and output it to the **led_ip.**

## 4-3. Assign the led_ip driver from the *driver* directory to the led_ip instance.

**4-3-1.** Select **Xilinx Tools > Repositories.**

**4-3-2.** Click on **New** button of *Local Repositories*, browse to *C:\xup\sys_design\labs\led_ip\led_ip_1.0* and click **OK**, and click **OK** again to close the Preferences window

**4-3-3.** Select **standalone_bsp_0** in the project view, right-click, and select **Board Support Package Settings.**

**4-3-4.** Select *drivers* on the left (under *Overview*)

**4-3-5.** Select *Generic* under the *Driver* column for *led_ip* to access the dropdown menu. From the dropdown menu, select **led_ip**, and click **OK.**

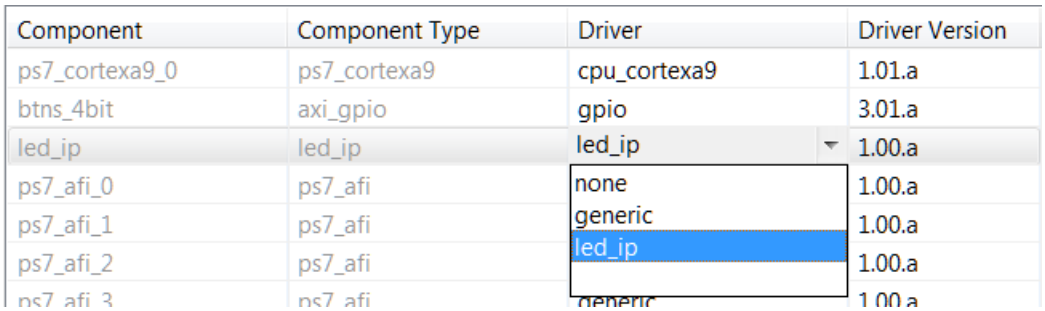



**Figure 14. Assign led_ip driver**

## 4-4. Examine the Driver code

The driver code was generated automatically when the IP template was created. The driver includes higher level functions which can be called from the user application. The driver will implement the low level functionality used to control your peripheral.

**4-4-1.** In windows explorer, browse to *C:\xup\sys_design\labs\led_ip\led_ip_1.0*\drivers\led_ip_v1_00_a\src. Notice the files in this directory and open led_ip.c. This file only includes the header file for the IP.

**4-4-2.** Close led_ip.c and open the header file led_ip.h and notice the macros:
LED_IP_mWriteReg( … )
LED_IP_mReadReg( … )

e.g: search for the macro name LED_IP_mWriteReg:

```
/**
 *
 * Write a value to a LED_IP register. A 32 bit write is performed.
 * If the component is implemented in a smaller width, only the least
 * significant data is written.
 *
 * @param   BaseAddress is the base address of the LED_IP device.
 * @param   RegOffset is the register offset from the base to write to.
 * @param   Data is the data written to the register.
 *
 * @return  None.
 *
 * @note
 * C-style signature:
 *    void LED_IP_mWriteReg(Xuint32 BaseAddress, unsigned RegOffset,
Xuint32 Data)
 *
 */
#define LED_IP_mWriteReg(BaseAddress, RegOffset, Data) \
Xil_Out32((BaseAddress) + (RegOffset), (Xuint32)(Data))
```

For this driver, you can see the macros are aliases to the lower level functions Xil_Out32( ) and Xil_Out32( ). The macros in this file make up the higher level API of the led_ip driver. If you are writing your own driver for your own IP, you will need to use low level functions like these to read and write from your IP as required. The low level hardware access functions are wrapped in your driver making it easier to use your IP in an Application project.

**4-4-3.** Modify your C code (see figure below, or you can find modified code in **lab4_soln.c** from **sources** folder) to echo the dip switch settings on the LEDs by using the led_ip driver API macros, and save the application.

**4-4-4.** Include the header file:

```
#include "led_ip.h"
```

**4-4-5.** Include the function to write to the IP (insert before the *for* loop):

```
LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, dip_check);
```

Remember that the hardware address for a peripheral (e.g. the macro `XPAR_LED_IP_S_AXI_BASEADDR` in the line above) can be found in *xparameters.h*

```
#include "xparameters.h"
#include "xgpio.h"
#include "xutil.h"
#include "led_ip.h"
//====================================================

int main (void)
{

   XGpio dip, push;
   int i, psb_check, dip_check;

   xil_printf("-- Start of the Program --\r\n");

   XGpio_Initialize(&dip, XPAR_SW_4BIT_DEVICE_ID); // Modify this
   XGpio_SetDataDirection(&dip, 1, 0xffffffff);

   XGpio_Initialize(&push, XPAR_BTNS_4BIT_DEVICE_ID); // Modify this
   XGpio_SetDataDirection(&push, 1, 0xffffffff);


   while (1)
   {
      psb_check = XGpio_DiscreteRead(&push, 1);
      xil_printf("Push Buttons Status %x\r\n", psb_check);
      dip_check = XGpio_DiscreteRead(&dip, 1);
      xil_printf("DIP Switch Status %x\r\n", dip_check);

      // output dip switches value on LED_ip device
      LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, dip_check);

      for (i=0; i<9999999; i++);
   }
}
```

**Figure 15. The completed C file**

**4-4-6.** Save the file and the program will be compiled again.

## Verify in Hardware Step 5

### 5-1. Connect and power up the board. Set the Terminal tab to communicate at 115200 baud using correct COM port. Program the FPGA, and run the lab4.elf application.

**5-1-1.** Connect and power up the board.

**5-1-2.** Select the Terminal tab. If it is not visible then select **Window > Show view > Terminal**.

**5-1-3.** Click on and select appropriate COM port (depends on your computer), and configure it with 115200 baud rate if necessary

**5-1-4.** In SDK, select **Xilinx Tools > Program FPGA.**

**5-1-5.** Click the **Program** button to program the FPGA.

**5-1-6.** Select **lab4** in *Project Explorer*, right-click and select **Run As > Launch on Hardware** to download the application, execute ps7_init, and execute lab4.elf

Flip the DIP switches and verify that the LEDs light according to the switch settings. Verify that you see the results of the DIP switch and Push button settings in SDK Terminal.

```
DIP Switch Status 7
Push Buttons Status 0
DIP Switch Status 7
Push Buttons Status 0
DIP Switch Status 7
```

**Figure 16. DIP switch and Push button settings displayed in SDK terminal**

Note: Setting the DIP switches and push buttons will change the results displayed.

**5-1-7.** When finished, click on the **Terminate** button in the *Console* tab.

**5-1-8.** Close SDK and Vivado, and power OFF the board.

## Conclusion

Manage IP feature was used to create an AXI Lite IP. The created IP was modified to add the desired functionality. Vivado IP packager was used to update the IP and then the custom IP block was imported into the IP library. The IP block was then added to the system. Connection automation was run, where available, to speed up the design of the system by allowing Vivado to automatically make connections between IP. Pin location constraints were added to the design. The design bitstream was generated.

SDK was used to define, develop, and integrate the software components of the embedded system. The device driver was assigned to the custom IP. The peripheral-specific functional software was created and the executable file was generated.

# Debugging Using Hardware Analyzer

## Introduction

Software and hardware interacts with each other in an embedded system. The Xilinx SDK includes both GNU and the Xilinx Microprocessor Debugger (XMD) as software debugging tools. The hardware analyzer tool allows for hardware debugging by providing access to the internal signals without necessarily bringing them out via the package pins using several types of cores. These powerful cores may reside in the programmable logic (PL) portion of the device and can be configured with several modes that can monitor signals within the design. Vivado provides capabilities of marking any net at several stages of the design flow. In this lab you will be introduced to various debugging cores.

## Objectives

After completing this lab, you will be able to:

- Add the VIO core in the design
- Use VIO to insert stimulus and monitor the response
- Mark nets to debug so the AXI transactions can be monitored
- Add the ILA core in Vivado
- Perform hardware debugging using the hardware analyzer
- Perform software debugging using the SDK

## Procedure

This lab is separated into steps that consist of general overview statements providing information on the subsequent detailed instructions. Follow the (step-by-step) detailed instructions to progress through the lab.

## Design Description

In this lab, you will add a custom core that performs a simple 8-bit addition function. The core used was developed previously using the IP Packager capability of Vivado and is provided as part of the lab source files. The core has additional ports so that stimuli can be brought in and the response can be monitored. This way the core can be tested independently without using the PS or software application. The following block diagram represents the completed design (**Figure 1**).

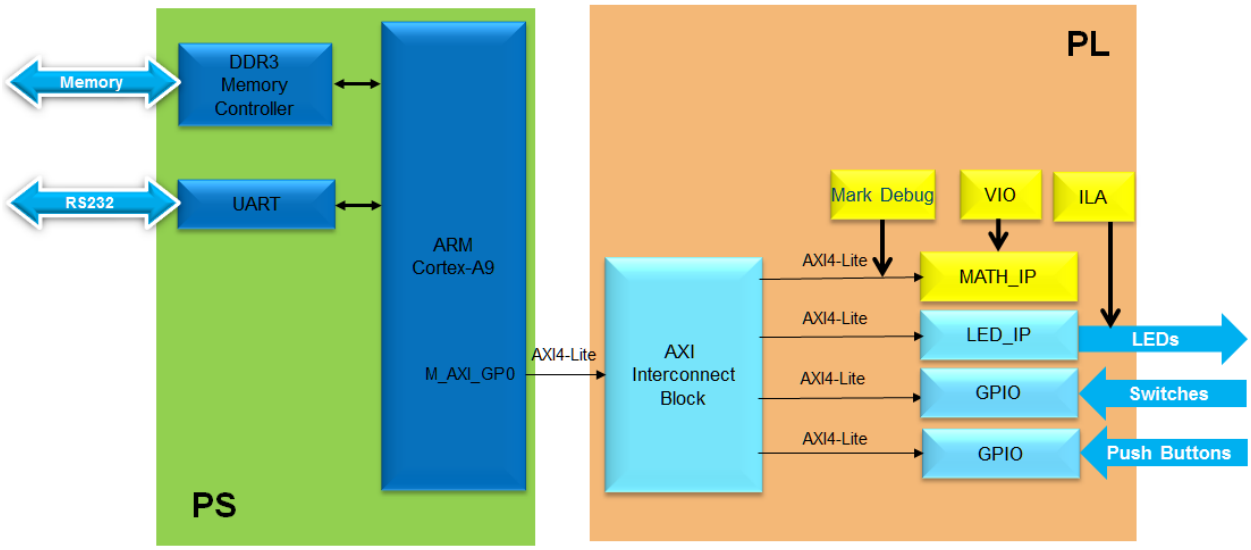
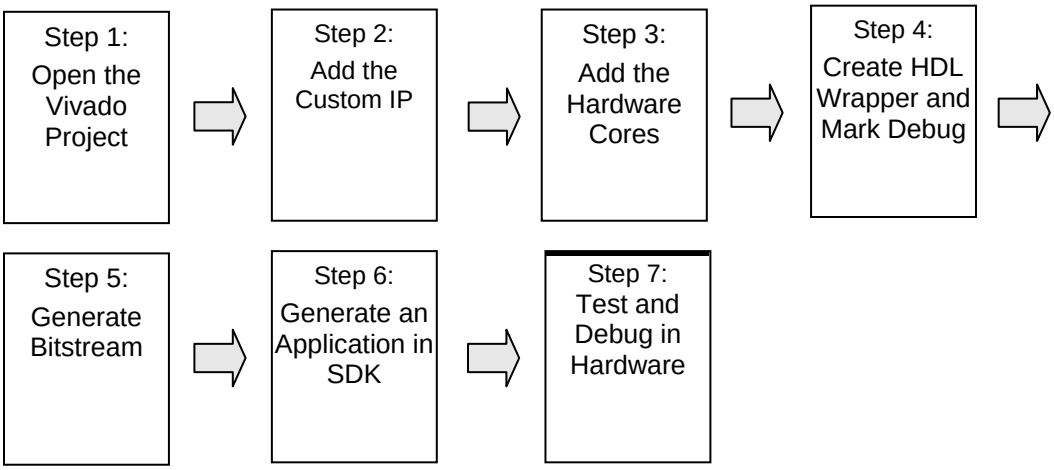



**Figure 1. Completed Design**

## General Flow for this Lab




## Open the Project Step 1

**1-1. Open the Vivado program. Open the *lab4* project you created in the previous lab or use the *lab4* project from the labsolution directory, and save the project as lab5. Set Project Settings to point to the IP repository provided in the *sources\lab5* directory.**

**1-1-1.** Start the Vivado if necessary and open either the lab4 project you created in the previous lab or the lab4 project in the labsolution directory using the **Open Project** link in the Getting Started page.

**1-1-2.** Select **File > Save Project As …** to open the *Save Project As* dialog box. Enter **lab5** as the project name. Make sure that the *Create Project Subdirectory* option is checked, the project directory path is c:\xup\sys_design\labs\ and click **OK**.

This will create the lab5 directory and save the project and associated directory with lab5 name.

**1-1-3.** Click **Project Settings** in the *Flow Navigator* pane.

**1-1-4.** Select **IP** in the left pane of the *Project Settings* form.

**1-1-5.** Click on the **Add Repository…** button, browse to **c:\xup\sys_design\sources\lab5\math_ip** and click **Select**.

**1-1-6.** The *math_ip_v1_0* IP will appear the **IP in Selected Repository** window.

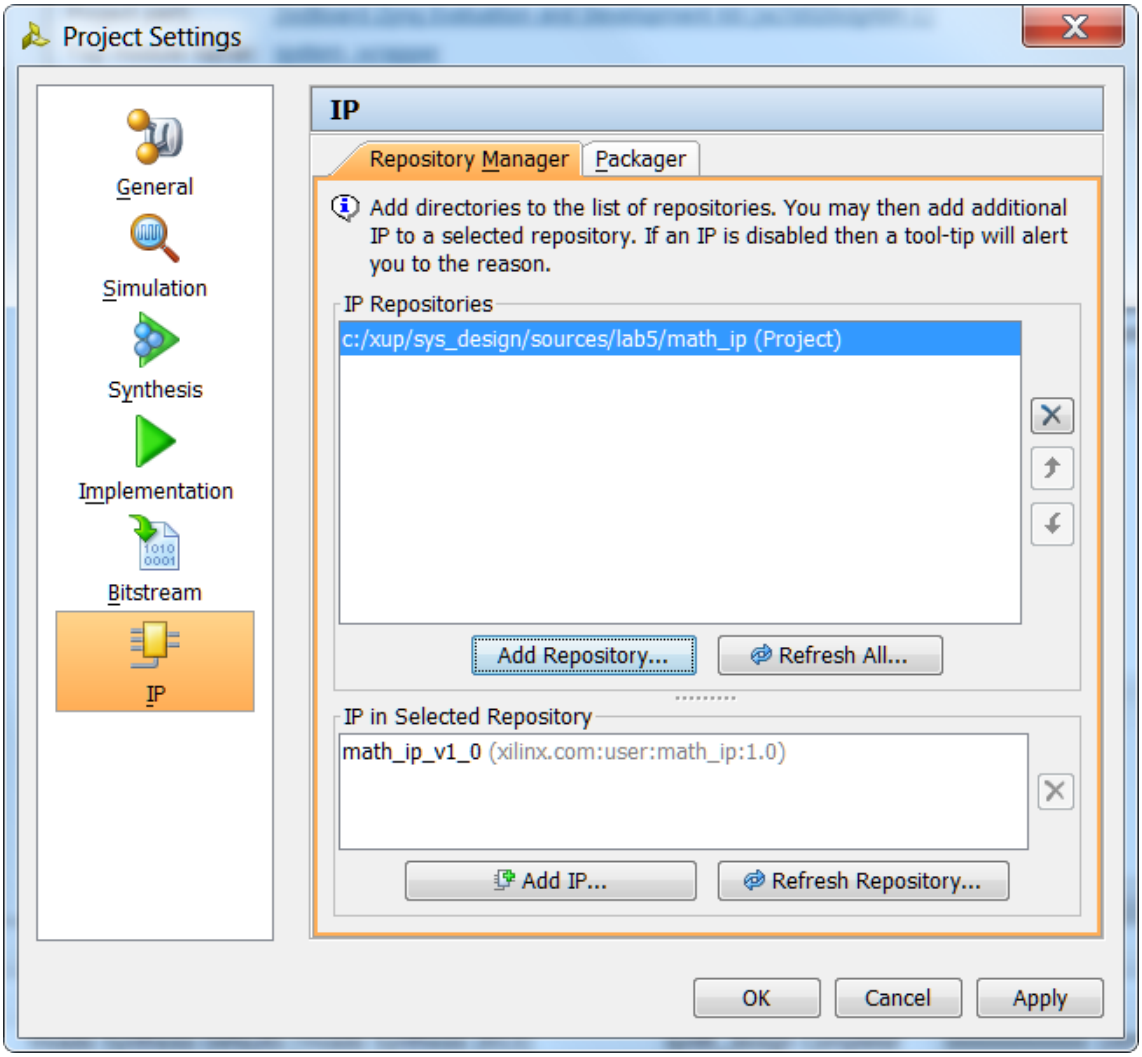



**Figure 2. Specify IP Repository**

**1-1-7.** Click **OK**.

## Add the Custom IP — Step 2

### 2-1. Open the Block Design and add the custom IP to the system.

**2-1-1.** Click **Open Block Design** in the *Flow Navigator* pane, and select **system.bd** to open the block diagram.

**2-1-2.** Click the Add IP icon and search for **math** in the catalog.

**2-1-3.** Double-click the **math_ip_v1_00_0** to add an instance of the core to the design

**2-1-4.** Click on **Run Connection Automation**, and select **/math_ip_0/S_AXI.**

The custom IP consists of a hierarchical design with the lower-level module performing the addition. The higher-level module includes the two slave registers.

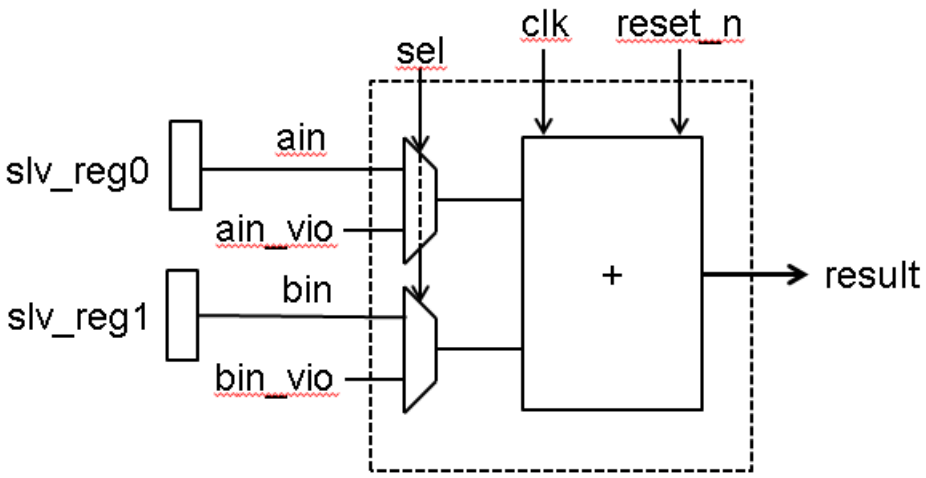



**Figure 3. Custom Core's Main Functional Block**

**2-1-5.** Click **OK** to connect the IP.

## Add the ILA and VIO Cores Step 3

### 3-1. Add the ILA core and connect it to the LED output port.

**3-1-1.** Click the Add IP icon and search for **ila** in the catalog.

**3-1-2.** Double-click on the **ILA (Integrated Logic Analyzer)** to add an instance of it. The *ila_0* instance will be added.

**3-1-3.** Double-click on the *ila_0* instance and select the **Probe Ports** tab.

**3-1-4.** Set the **Probe Width** of *PROBE0* to **4** and click **OK**.

**3-1-5.** Using the drawing tool, connect the **PROBE0** port of the *ila_0* instance to the **LED** port of the *led_ip* instance.

**3-1-6.** Connect the CLK port of the *ila_0* instance to the **s_axi_aclk** port of one of the other instances.

### 3-2. Add the VIO core and connect it to the math_ip ports.

**3-2-1.** Click the Add IP icon and search for **vio** in the catalog

**3-2-2.** Double-click on the **VIO (Virtual Input/Output)** to add an instance of it. The *vio_0* instance will be added.

**3-2-3.** Double-click on the **vio_0** instance to open the configuration form.

**3-2-4.** Set the *Output Probe Count* to **3** and the *Input Probe Count* to **1** in the *General Options* tab.

**3-2-5.** Select the *PROBE_IN Ports* tab and set the *PROBE_IN0* width to **9**.

**3-2-6.** Select the *PROBE_OUT Ports* tab and set *PROBE_OUT0* width to **1**, *PROBE_OUT1* width to **8**, and *PROBE_OUT2* width to **8**.

**3-2-7.** Click **OK**.

**3-2-8.** Connect the VIO instance's ports to the math instance ports as follows:

PROBE_IN0 -> result
PROBE_OUT0 -> sel
PROBE_OUT1 -> ain_vio
PROBE_OUT2 -> bin_vio

**3-2-9.** Connect the **CLK** port of the *vio_1* to **s_axi_aclk** port of one of the other instances.

**3-2-10.** The block diagram should look similar to shown below.

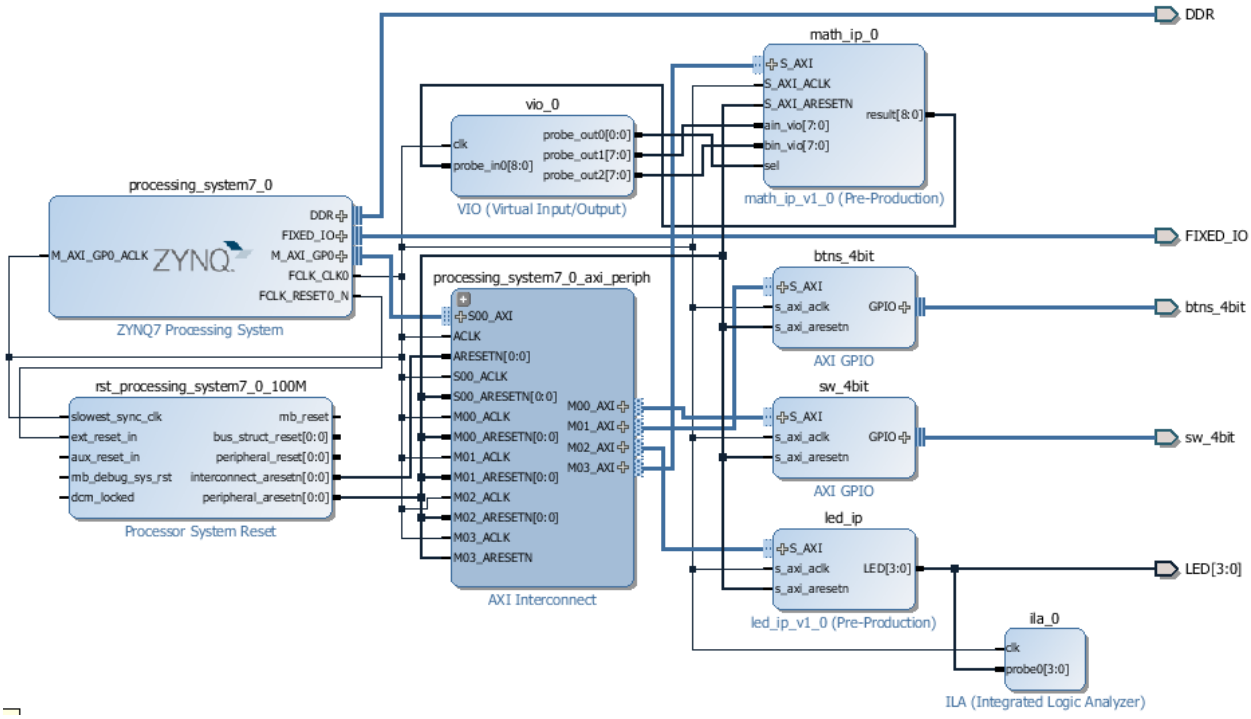



**Figure 4. VIO instance added and connections made**

## 3-3. Mark Debug the S_AXI connection between the AXI Interconnect and math_0 instance. Validate the design.

**3-3-1.** Select the **S_AXI** connection between the AXI Interconnect and the math_ip_0 instance**.**

**3-3-2.** Right-click and select **Mark Debug** to monitor the AXI4Lite transactions.

**3-3-3.** Select **Tools > Validate Design** to run the design rules checker.

Verify that there are no unmapped addresses shown in the *Address Editor* tab.

**3-3-4.** Click **OK**.

## Create HDL Wrapper and Assign Nets for Debugging Step 4

### 4-1. Create the top-level HDL wrapper of the embedded system. Run the synthesis.

**4-1-1.** Select the *Sources >Hierarchy* tab. Expand the *Design Sources,* right-click the *system.bd* and select **Create HDL Wrapper**.

**4-1-2.** Click **OK**.

**4-1-3.** Click **Run Synthesis**.

**4-1-4.** Click **OK** to run the synthesis process (click **Yes** to save the project if prompted).

**4-1-5.** When the synthesis is completed, select the *Open Synthesized Design* option and click **OK**.

### 4-2. Assign nets for debugging.

**4-2-1.** The synthesized design will be opened in the Auxiliary pane and the **Debug** tab will be opened in the *Console* pane.

If the Debug tab is not open then select **Window > Debug**.

Notice that the nets which can be debugged are grouped into Assigned and Unassigned groups. The assigned net groups include nets associated with the VIO and ILA cores, whereas the unassigned nets group includes S_AXI related nets.

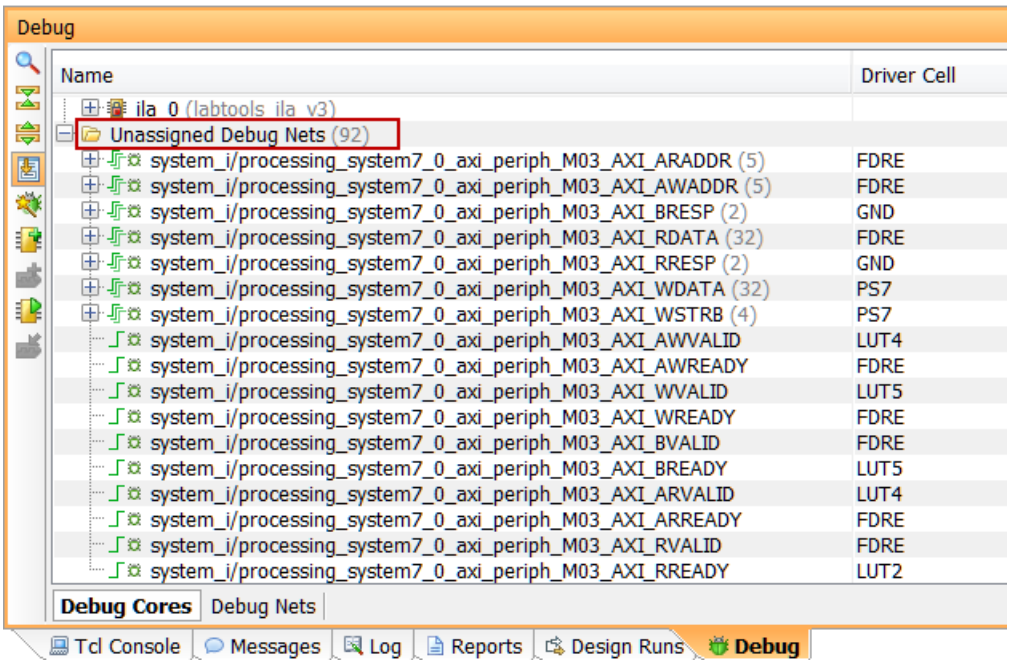



**Figure 5. The Debug tab**

**4-2-2.** Right-click on the *Unassigned Debug Nets* and select the **Set up Debug…** option.

**4-2-3.** Click **Next.**

The nets are listed.

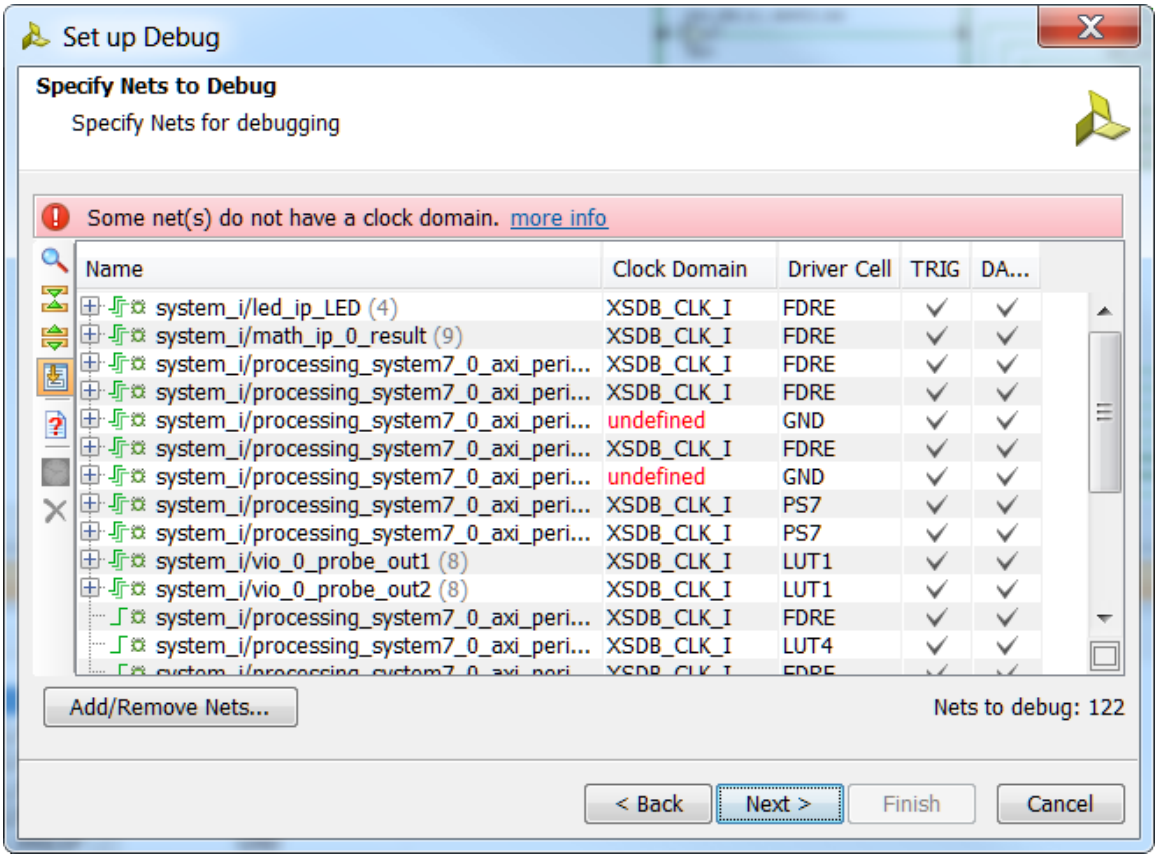



**Figure 6. Nets to debug**

**4-2-4.** Right click on the **BRESP** and **RRESP** (which are driven by GND) and select **Remove Nets**.

**4-2-5.** Click **Next** twice**,** and then **Finish**.

**4-2-6.** In the Design Runs tab, select *synth_1*, right-click, and select *Reset Run*.

**4-2-7.** Click **Yes.**

## Generate the Bitstream Step 5

### 5-1. Generate the bitstream.

**5-1-1.** Click on the **Generate Bitstream** to run the implementation and bit generation processes.

**5-1-2.** Click **Save** to save the project, and **Yes** to run the processes.

**5-1-3.** When the bitstream generation process has completed successfully, a box with three options will appear. Select the **Open Implemented Design** option and click **OK**.

**5-1-4.** Click **Yes** to close the synthesized design, if prompted.

## Generate an Application in SDK Step 6

### 6-1. Start the SDK by exporting the implemented design. Refresh the standalone_bsp_0 project as the hardware has changed.

**6-1-1.** Start the SDK by clicking **File > Export > Export Hardware for SDK.**

The *Export Hardware for SDK* window will appear.

**6-1-2.** Click on the **Launch SDK** check box to launch the SDK session.

**6-1-3.** Click **OK** to export and **Yes** to overwrite the previous project created by lab4.

**6-1-4.** Right-click on *lab4* project and select **Close Project** to close that project.

**6-1-5.** Right-click on the **standalone_bsp_0** project in the Project Explorer view and select **Board Support Package Settings**

**6-1-6.** Select *drivers* in the left pane, click on the drop-down button in the *drivers* column for the *math_ip_0*, and make sure that *generic* driver is selected.

**6-1-7.** Click **OK** to accept the settings and close the form.

### 6-2. Create an empty application project named lab5, and import the provided lab5.c file.

**6-2-1.** Select **File > New** > **Application Project.**

**6-2-2.** In the *Project Name* field, enter **lab5** as the project name.

**6-2-3.** Select the *Use existing* option in the *Board Support Package* field and then click **Next.**

**6-2-4.** Select the **Empty Application** template and click **Finish.**

The lab5 project will be created in the Project Explorer window of the SDK.

**6-2-5.** Select **lab5 > src** in the project view, right-click, and select **Import.**

**6-2-6.** Expand the **General** category and double-click on **File System.**

**6-2-7.** Browse to the **c:\xup\sys_design\sources\lab5** folder.

**6-2-8.** Select **lab5.c** and click **Finish.**

A snippet of the part of the source code is shown in the following figure. It shows that we write operands to the custom core, read the result, and print it. We will use the write transaction as a trigger condition in the Vivado Analyzer.

```
        xil_printf("-- Change slide switches to see corresponding output "
                "on LEDs --\r\n");

        Xil_Out32(XPAR_MATH_IP_0_BASEADDR, 0x12);
        Xil_Out32(XPAR_MATH_IP_0_BASEADDR+4, 0x34);
        i=Xil_In32(XPAR_MATH_IP_0_BASEADDR);
        xil_printf("result=%x\r\n",i);

        while (1)
        {
            sw_check = XGpio_DiscreteRead(&sw, 1);
            LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, sw_check);
            for (i=0; i<9999999; i++); // delay loop
         }

    }
```

**Figure 7. Source Code snippet**

## Test in Hardware Step 7

### 7-1. Connect and power up the board. Establish serial communications using the SDK's Terminal tab. Download the bitstream into the target device. Start the debug session on lab5 project.

**7-1-1.** Connect and power up the board.

**7-1-2.** Select the Terminal tab. If it is not visible then select **Window > Show view > Terminal**.

**7-1-3.** Click on and select the appropriate COM port (depending on your computer), and configure it as you did it in Lab 1.

**7-1-4.** Select **Xilinx Tools** > **Program FPGA.**

**7-1-5.** Click **Program**.

**7-1-6.** Select the **lab5** project in *Project Explorer*, right-click and select **Debug As > Launch on Hardware (System Debugger)** to download the application, execute ps7_init, and display a dialog box asking to switch the perspective to the Debug perspective.

**7-1-7.** Click **Yes** and the perspective will change. The program execution starts and suspends at the entry point.

### 7-2. Start the hardware session from Vivado.

**7-2-1.** Switch to Vivado.

**7-2-2.** Click on **Open Hardware Manager** from the *Program and Debug* group of the *Flow Navigator* pane to invoke the analyzer.

**7-2-3.** Click on the **Open a New Hardware Target** link to establish the connection with the board.

**7-2-4.** Click **Next** twice to use the default settings.

The JTAG chain will be scanned and the device will be detected.

**7-2-5.** Click **Next** twice and then **Finish**.

The hardware session will open showing the **Debug Probes** tab.

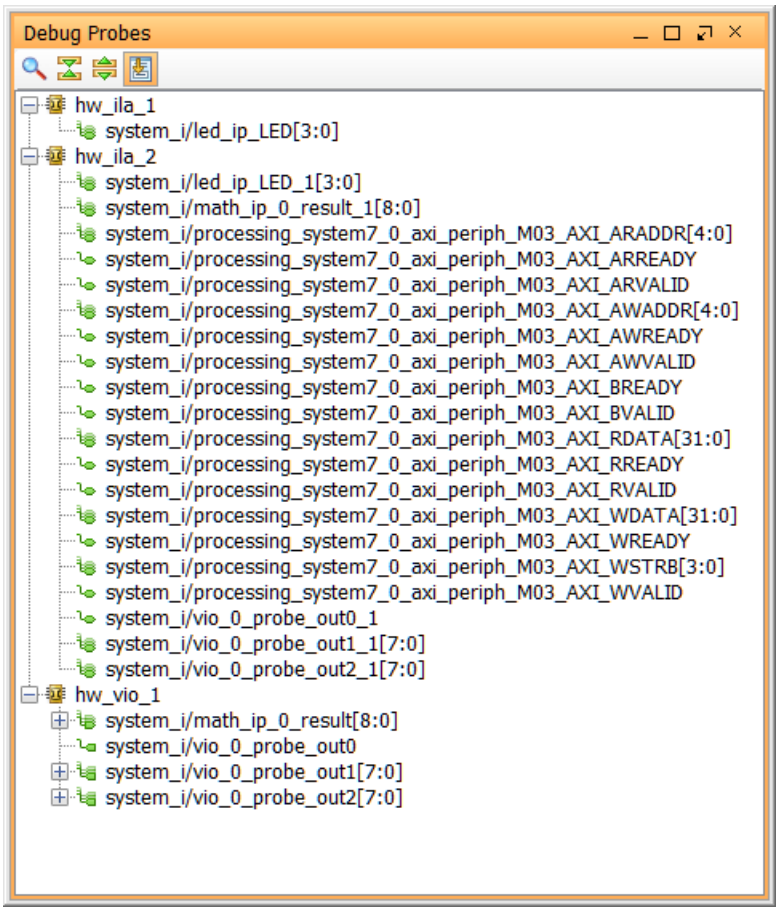



**Figure 8. Debug probes**

The hardware session status window also opens showing that the FPGA is programmed (we did it in SDK), there are three cores, and the two ila cores with the idle state.

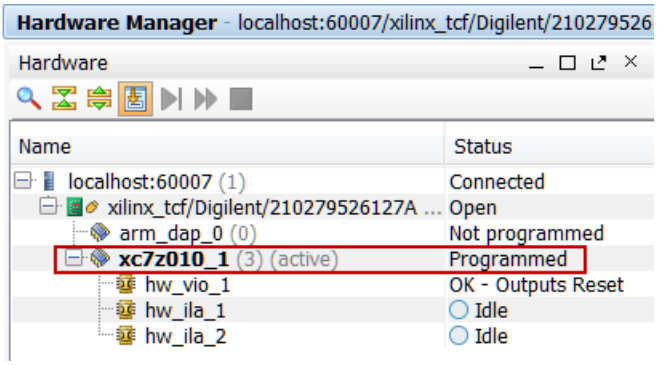



**Figure 9. Hardware session status**

**7-2-6.** Select **XC7Z010_1**, and click on the **Run Trigger Immediate** button to see the signals in the waveform window.

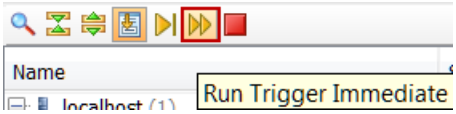



**Figure 10. Opening the waveform window**

Two waveform windows will be created, one for each ila; one ila (hw_ila_1) is of the instantiated ILA core and another (hw_ila_2) for the MARK DEBUG method.

## 7-3. Setup trigger conditions to trigger on a write transaction (WSTRB) when the valid address (AWADDR) is written with data (WVALID equal to 1).

**7-3-1.** In the **Debug Probes** window, select the *AWADDR* bus and drag it into the *ILA-hw_ila_2* window and release to add it to set a trigger condition on it (you can also add the signal by selecting it, right-clicking and selecting Add Probes to Basic Trigger Setup) .

**7-3-2.** Change the value from xx to 04 (the slave_reg2 address of the math_1 instance).

**7-3-3.** Similarly, add the **WSTRB** and **WVALID** signals, and change the condition from xxxx to xxx1 and to 1.

**7-3-4.** Set the trigger position of the *hw_ila_2* to **512**.

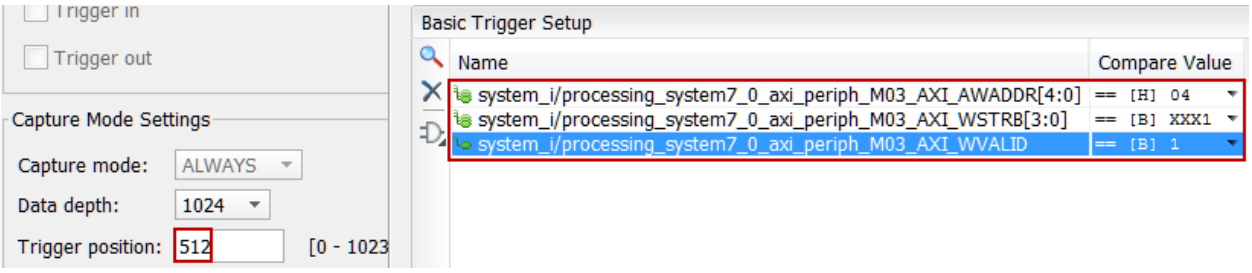



**Figure 11. Setting up the trigger position**

**7-3-5.** Similarly, set the trigger position of the *hw_ila_1* to **512**.

**7-3-6.** Select hw_ila_2 in the *Hardware* pane.

**7-3-7.** Click on the **Run Trigger** ( ) button and observe that the *hw_ila_2* core is armed and showing the status as **Waiting For Trigger**.

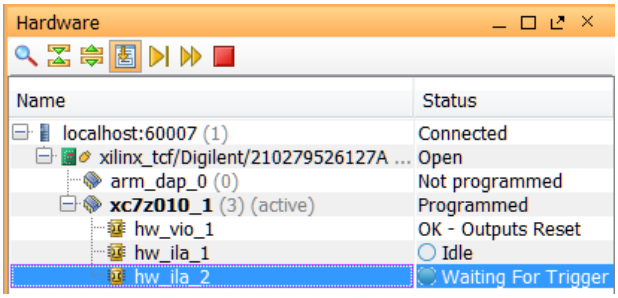



**Figure 12. Hardware analyzer running and in capture mode**

**7-3-8.** Switch to SDK.

**7-3-9.** Double click on the left border on the line where xil_printf statement is (before the while (1) statement) is defined in the lab5.c window to set a breakpoint.

```
Xil_Out32(XPAR_MATH_IP_0_BASEADDR, 0x12);
Xil_Out32(XPAR_MATH_IP_0_BASEADDR+4, 0x34);
i=Xil_In32(XPAR_MATH_IP_0_BASEADDR);
xil_printf("result=%x\r\n",i);
```

**Figure 13. Setting a breakpoint**

**7-3-10.** Click on the **Resume** ( ) button to execute the program and stop at the breakpoint.

**7-3-11.** In the Vivado program, notice that the **hw_ila_2** status changed from *capturing* to *Idle*, and the waveform window shows the triggered output.

**7-3-12.** Move the cursor to closer to the trigger point and then click on the button to zoom at the cursor. Click on the **Zoom In** button couple of times to see the activity near the trigger point.

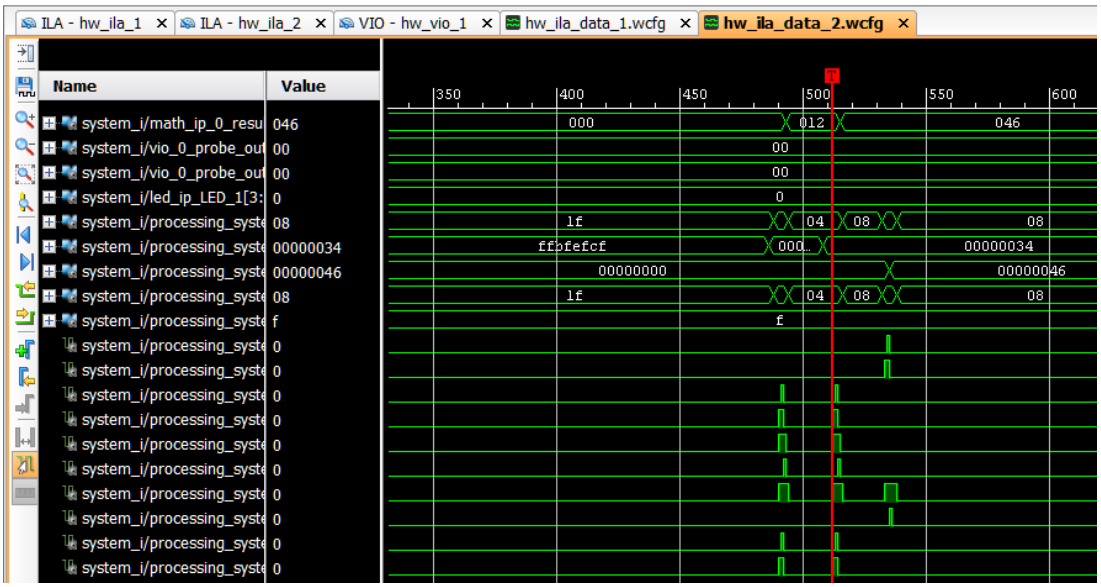



**Figure 14. Zoomed waveform view**

Observe the following:

Around 490th sample the RDATA value is 0x00, WDATA being written is 0x012 at offset 0 (AWADDR=0x0), WVALID is '1', WREADY '1' indicating the data is being written into the IP.

At 512th sample, WVALID is '1' WSTRB is 0xf, offset is 0x4 (AWADDR), and the data being written is 0x034.

At 536th sample, RREADY and RVALID are '1' indicating data (result=0x046) is being read from the IP at the offset 0x0 (ARADDR).

**7-3-13.** You also should see the following output in the SDK Terminal console.

```
-- Start of the Program --
-- Change slide switches to see corresponding output on LEDs --
```

**Figure 15. SDK Terminal Output**

## 7-4. In Vivado, select the VIO Core in Console, add the signals to the VIO-hw_vio_1 window, and set the vio_0_probe_out0 so math_ip's input can be controlled manually through the VIO core. Try entering various values for the two operands and observe the output on the math_ip_0_result port in the Console pane.

**7-4-1.** Select the all the signals of the **VIO Cores** in the Debug Probes, drag them into the *VIO-hw_vio_1* window.

**7-4-2.** Select **vio_0_probe_out0** and change its value to **1** so the math_ip core input can be controlled via the VIO core.

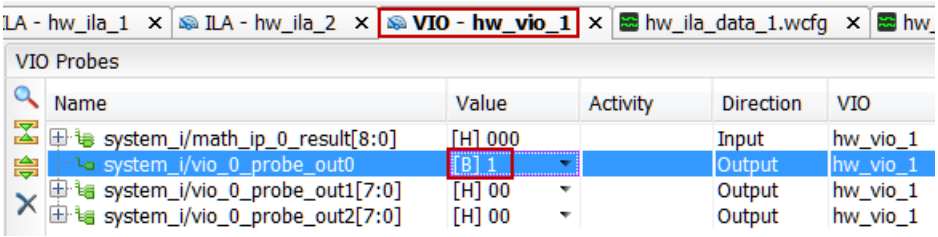



**Figure 16. VIO probes**

**7-4-3.** Click on the Value field of *vio_0_probe_out1* and change the value to **55** (in Hex). Similarly, click on the Value field of *vio_0_probe_out2* and change the value to **44** (in Hex). Notice that for a brief moment a blue-colored up-arrow will appear in the Activity column and the result value changes to **099** (in Hex).

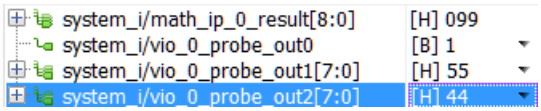



**Figure 17. Input stimuli through the VIO core's probes**

**7-4-4.** Try a few other inputs and observe the outputs.

**7-4-5.** Once done, set the *vio_0_probe_out0* to **0** to isolate the vio interactions with the math_ip core.

This is not required in this exercise but normally you would do.

## 7-5. Setup the ILA core (hw_ila_1) trigger condition to 0101 (x5). Make sure that the switches on the board are not set at x5. Set the trigger equation to be ==, and arm the trigger. Click on the Resume button in the SDK to continue executing th eprogram. Change the switches and observe that the hardware core triggers when the preset condition is met.

**7-5-1.** Select the **hw_ila_1** in the *ILA Cores* tab and the *led_ip_led* signals to the *ILA-hw_ila_1* window.

**7-5-2.** Set the trigger condition of the *hw_ila_1* to trigger at LED output value equal to **55**.

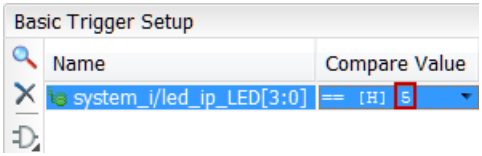



**Figure 18.Setting up Trigger for hw_ila_1**

**7-5-3.** Verify that the trigger position for the *hw_ila_1* is set to **512**. If not, then set it.

Make sure that the switches are not set to 0101

**7-5-4.** Right-click on the *hw_ila_1* in the *hardware* window, and arm the trigger by selecting **Run Trigger.**

The hardware analyzer should be waiting for the trigger condition to occur indicated by the *Capturing* status.

**7-5-5.** In the SDK window, click on the *Resume* button.

**7-5-6.** Change the slide switches and see the corresponding LED turning ON and OFF.

**7-5-7.** When the condition (0x5) is met, the waveform will be displayed.


**Figure 20. ILA waveform window after Trigger**

**7-5-8.** Click on the Disconnect button ( ) in the SDK to terminate the execution.

**7-5-9.** Close the SDK by selecting **File > Exit**.

**7-5-10.** In Vivado, close the hardware session by selecting **File > Close Hardware Manager**. Click **OK**.

**7-5-11.** Click **No** to not to save the waveform configuration.

**7-5-12.** Close Vivado program by selecting **File > Exit**. Click **OK**.

**7-5-13.** Turn OFF the power on the board.

## Conclusion

In this lab, you added a custom core with extra ports so you can debug the design using the VIO core. You instantiated the ILA and the VIO cores into the design. You used Mark Debug feature of Vivado to debug the AXI transactions on the custom peripheral. You then opened the hardware session from Vivado, setup various cores, and verified the design and core functionality using SDK and the Vivado hardware analyzer.

.

# Creating a Processor System Lab

## Introduction

This lab introduces a design flow of profiling an application, determine which function to port to hardware, generate an IP-XACT adapter from a design using Vivado HLS and use the generated IP-XACT adapter in a processor system using IP Integrator in Vivado.

## Objectives

After completing this lab, you will be able to:

- Profile a software application
- Understand the steps and directives involved in creating an IP-XACT adapter in Vivado HLS
- Create a processor system using IP Integrator in Vivado
- Integrate the generated IP-XACT adapter into the created processor system
- Profile an application having an hardware accelerator

## The Design

The design consists of a FIR filter to filter a 4 KHz tone added to CD quality (48 KHz) music. The characteristic of the filter is as follows:

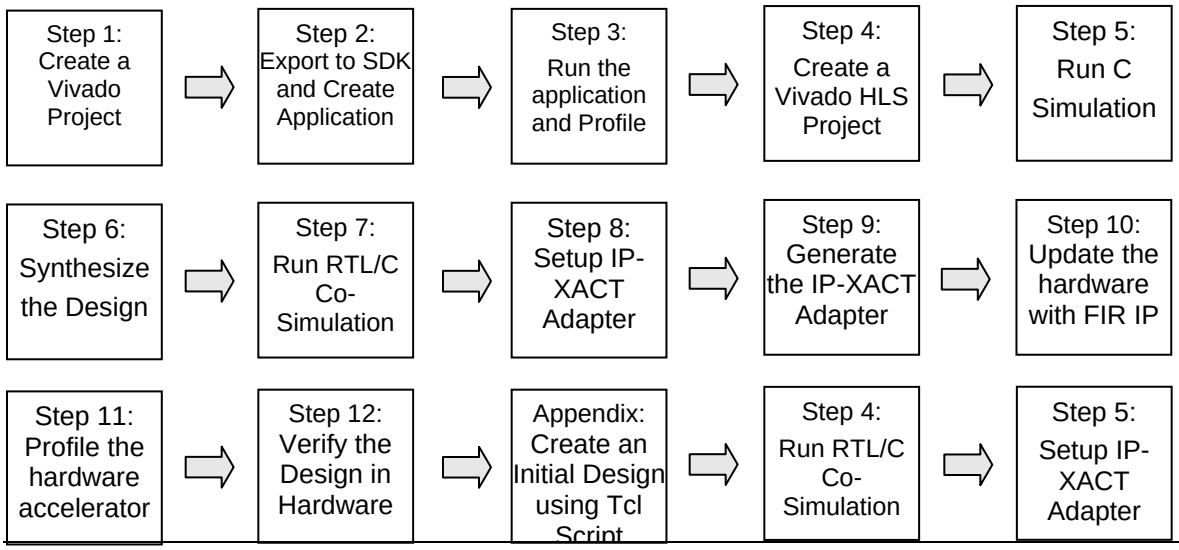
```
FS=48000 Hz
FPASS1=2000 Hz
PSTOP1=3800 Hz
FSTOP2=4200 Hz
FPASS2=6000 Hz
APASS1=APASS2=1 dB
ASTOP=60 dB
```

This lab requires you to develop a peripheral core of the designed filter that can be instantiated in a processor system.

## Procedure

This lab is separated into steps that consist of general overview statements that provide information on the detailed instructions that follow. Follow these detailed instructions to progress through the lab.

## General Flow for this Lab



## Create a Vivado Project Step 1

### 1-1. Launch Vivado Tcl Shell and run the provided tcl script to create an initial system targeting the ZYBO (having xc7z010clg400-1 device) board.

***If you want to create the system from scratch then follow the steps provided in Appendix and then continue from step 7-2 below.***

**1-1-1.** Open Vivado Tcl Shell by selecting **Start > All Programs > Xilinx Design Tools > Vivado 2013.4 > Vivado 2013.4 Tcl Shell**

**1-1-2.** In the shell window, change the directory to *c:/xup/sys_design/sources/lab6* using the **cd** account.

**1-1-3.** Run the provided script file to create an initial system having zybo_audio_ctrl and GPIO peripherals by typing the following command:

**source audio_project_create.tcl**

The script will be run and the initial system, shown below, will be created.

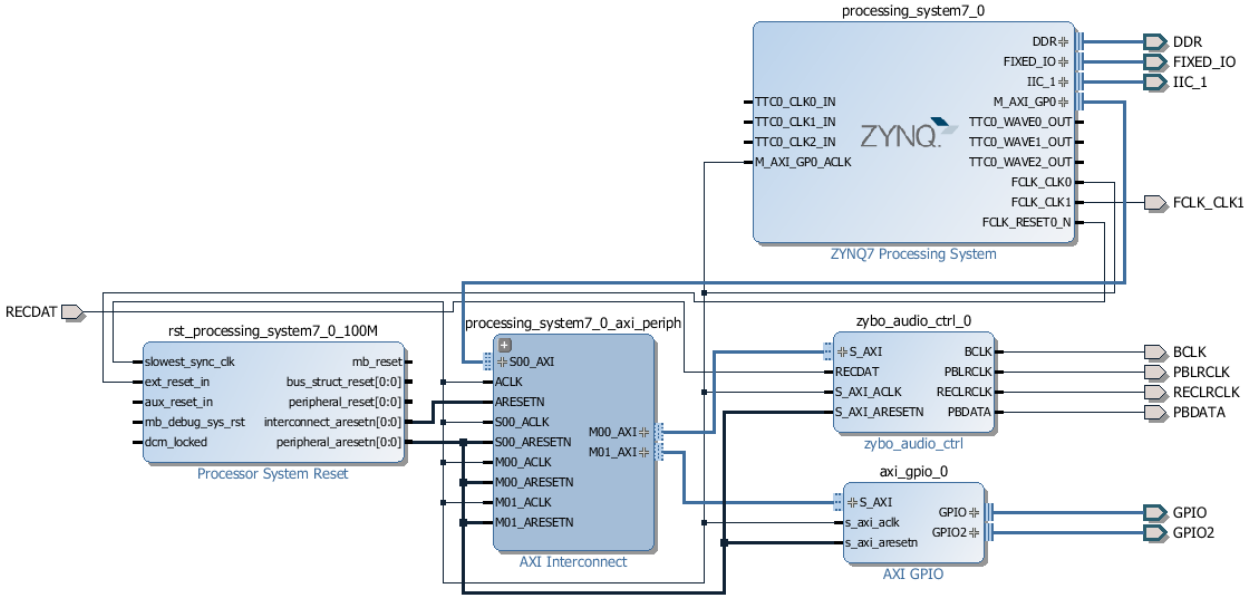



**Figure 12. Block design after I2C based zybo_audio_ctrl core added and connections made**

### 1-2. Verify addresses and validate the design. Generate the system_wrapper file, and add the provided Xilinx Design Constraints (XDC) file from the sources directory.

**1-2-1.** Click on the *Address Editor*, and expand the **processing_system7_0 > Data** if necessary.

The generated address map should look like as shown below.

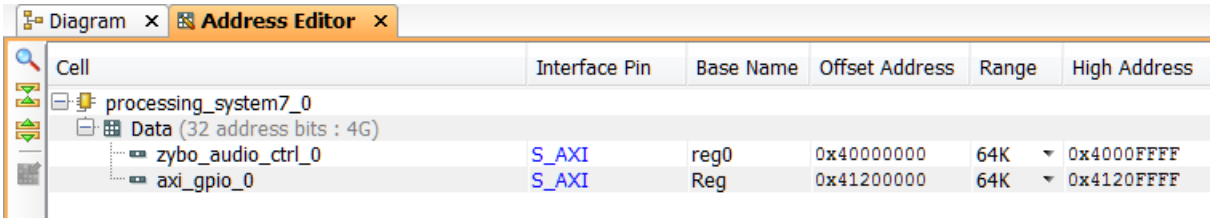



**Figure 2. Generated address map**

**1-2-2.** Run *Design Validation* (**Tools > Validate Design**) and verify there are no errors

**1-2-3.** In the *sources* view, right-click on the block diagram file, **system.bd**, and select **Create HDL Wrapper** to update the HDL wrapper file. When prompted, click **OK** with the *Let Vivado manage wrapper and auto-update* option.

**1-2-4.** Click **Add Sources** in the *Flow Navigator* pane, select **Add or Create Constraints**, and click **Next**.

**1-2-5.** Click the **Add Files** button, browse to the **c:\xup\sys_design\sources\lab6** folder, select **zybo_audio_constraints.xdc**

**1-2-6.** Click **Finish** to add the file.

**1-2-7.** Click on the **Generate Bitstream** in the Flow Navigator to run the synthesis, implementation, and bitstream generation processes.

**1-2-8.** Click **Save** and **Yes** if prompted.

**1-2-9.** When the bit generation is completed, a selection box will be displayed with the *Open Implemented Design* option selected. Click **OK**.

## Export to SDK and Create an Application Project Step 2

### 2-1. Export the hardware along with the generated bitstream to SDK. Create a Board Support Package enabling profiling.

To Export the hardware, the block diagram must be open and the Implemented design must be open.

**2-1-1.** If it is not already open, click **Open Block Design** (under IP Integrator in the Flow Navigator)

**2-1-2.** If it is not already open, click **Open Implemented Design** (under Implementation)

**2-1-3.** Start *SDK* by clicking **File > Export > Export Hardware for SDK**...

The export GUI will be displayed.

**2-1-4.** Check all three checkboxes, including the **Launch SDK** box and click **OK**.

**2-1-5.** In **SDK**, select **File** > **New** > **Board Support Package.**

**2-1-6.** Click **Finish** with the default settings (with standalone operating system).

This will open the Software Platform Settings form showing the OS and libraries selections.

**2-1-7.** Select the **Overview > standalone** entry in the left pane, click on the drop-down arrow of the *enable_sw_intrusive_profiling Value* field and select **true**.

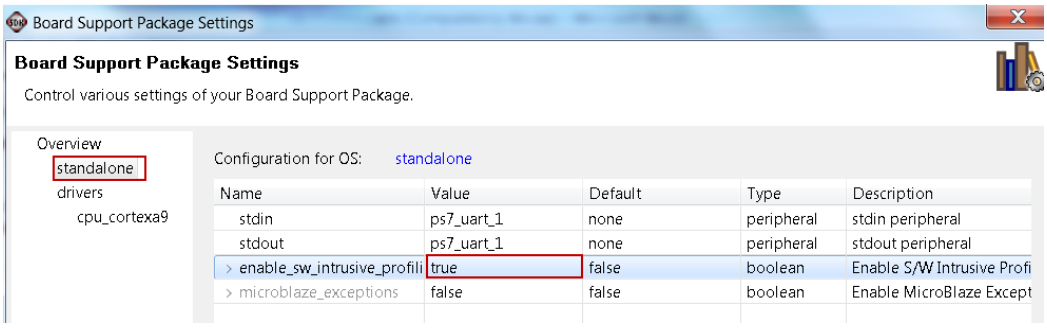



**Figure 3. Setting up for profiling**

**2-1-8.** Select the **Overview > drivers > cpu_cortexa9** and add **–pg** in addition to the –g in the *extra_compiler_flags Value* field.

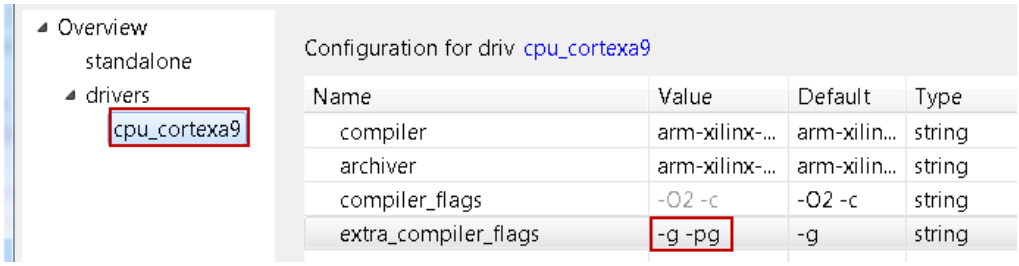



**Figure 4. Adding profiling switch**

**2-1-9.** Click **OK** to accept the settings and create the BSP.

## 2-2. Create an application for profiling.

**2-2-1.** Select **File** > **New** > **Application Project.**

**2-2-2.** Enter **lab6** as the *Project Name*, and for *Board Support Package*, choose **Use Existing** (*standalone_bsp_0* should be the only option).

**2-2-3.** Click **Next**, and select *Empty Application* and click **Finish.**

**2-2-4.** Select **lab6** in the project view, right-click the *src* folder, and select **Import.**

**2-2-5.** Expand **General** category and double-click on **File System.**

**2-2-6.** Browse to **c:\xup\sys_design\sources\lab6** folder and click **OK.**

**2-2-7.** Select **lab6.c, audio.h,** and **fir_coef.dat**, and click **Finish** to add the file to the project. (Ignore any errors/warnings for now).

## Run the Application and Profile Step 3

**3-1. Make sure that the JP7 is set to select USB power. Connect a micro-usb cable between a PC and the JTAG port of the board. Power ON the board. Program the PL section and run the application using the user defined SW_PROFILE symbol.**

**3-1-1.** Make sure that the JP7 is set to select USB power.

**3-1-2.** Make sure that a micro-USB cable is connected to the JTAG PROG connector (next to the power supply connector). Turn ON the power.

**3-1-3.** Power ON the board.

**3-1-4.** Select **Xilinx Tools > Program FPGA**.

**3-1-5.** Click on the **Program** button to download the bitstream and program the PL section.

**3-1-6.** Select the *lab6* application, right-click, and select **C/C++ Build Settings**.

**3-1-7.** Under the **ARM gcc compiler** group, select the **Symbols** sub-group**,** click on the **+** button to open the value entry form, enter **SW_PROFILE**, and click **OK**.

This will allow us to profile the software loop of the FIR application.

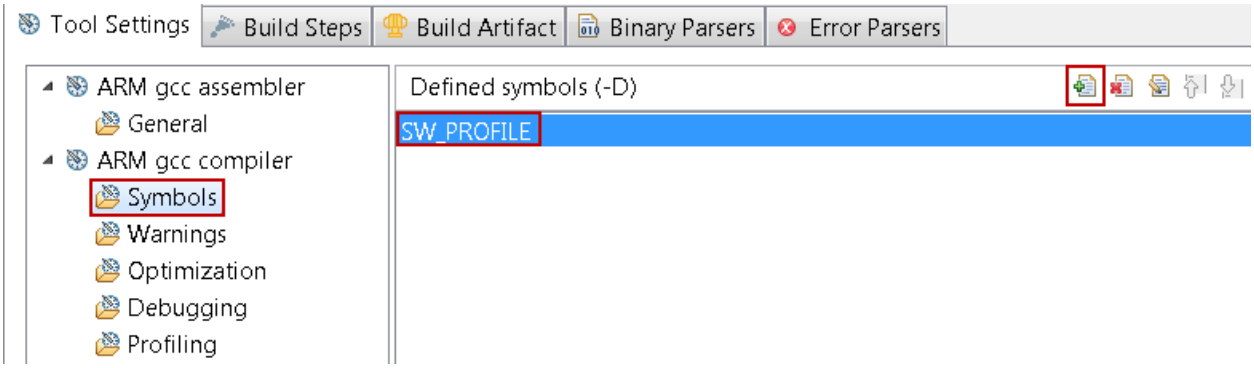



**Figure 5. Add a user-defined symbol**

**3-1-8.** Under the **ARM gcc compiler** group, select the **Profiling** sub-group, then check the **Enable Profiling** box, and click **OK**.

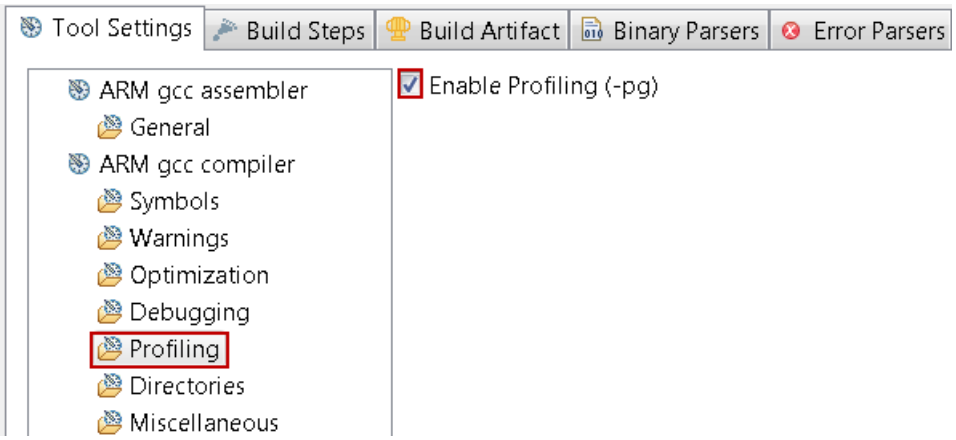



**Figure 6. Compiler setting for enabling profiling**

## 3-2. Set SW[1] to the up position. Create a Run Configuration, enabling the Profile option and setting up the profiling parameters. Run the profiler and gprof and analyze the results.

**3-2-1.** Make sure that the SW[1] is in the ON (up) position.

**3-2-2.** Select **Run > Run Configurations…** and double-click **Xilinx C/C++ Application** to create a new configuration.

**3-2-3.** Select the **Profile Options** tab. Click on the *Enable Profiling* check box, enter **1000000** (1 MHz) in the Sampling Frequency field, enter **0x10000000** in the scratch memory address field, and click **Apply**.

Use the high end address as the scratch memory address.

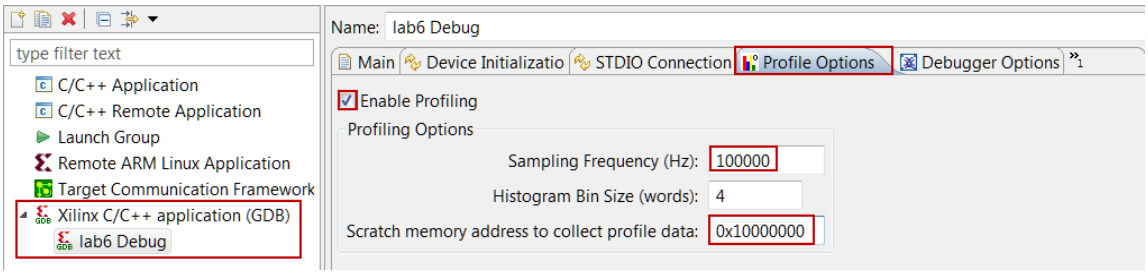



**Figure 7. Profiling options**

**3-2-4.** Click the **Apply** button followed by clicking the **Run** button to download the application and execute it.

When program execution has completed, a message will be displayed indicating that the profiling results are being saved in gmon.out file at the lab6\Debug directory.

**3-2-5.** Click **OK**.

**3-2-6.** Expand the *Debug* folder under the **lab6** project in the Project Explorer view, and double click on the **gmon.out** entry.

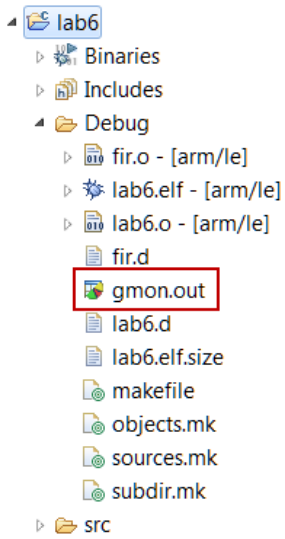



**Figure 8. Invoking gprof on gmon.out**

**3-2-7.** The Gmon File Viewer dialog box will appear showing *lab6.elf* as the corresponding binary file. Click **OK**.

**3-2-8.** Click on the **Sort samples per function** button (  ).

**3-2-9.** Click in the **%Time** column to sort in the descending order.

Note that the fir_software routine is called 68 times, 482 samples were taken during the profiling, and on an average of 7.88 microseconds were spent per call.

Problems | Tasks | Console | Properties | Terminal | gprof

gmon file: C:\xup\sys_design\labs\lab6\audio\audio.sdk\SDK\SDK_Export\lab6\Debug\gmon.out
program file: C:/xup/sys_design/labs/lab6/audio/audio.sdk/SDK/SDK_Export/lab6/Debug/lab6.elf
16 bytes per bucket, each sample counts as 1.0us

| Name (location) | Samples | Calls | Time/Call | %Time |
|---|---|---|---|---|
| Summary | 3479 | | | 100.0% |
| XIicPs_MasterSendPolled | 2873 | 11 | 261.181us | 82.58% |
| fir_software | 482 | 68 | 7.88us | 13.85% |
| __aeabi_uidiv | 35 | | | 1.01% |
| memcpy | 25 | | | 0.72% |
| mcount | 21 | | | 0.6% |

**Figure 9. Sorting results**

**3-2-10.** Select **File > Exit** to close the SDK.

**3-2-11.** In Vivado, select **File > Close Implemented Design**.

**3-2-12.** Power OFF the board.

# Create a Vivado HLS Project Step 4

## 4-1. Create a new project in Vivado HLS targeting xc7z010clg400-1.

**4-1-1.** Launch Vivado HLS: Select **Start > All Programs > Xilinx Design Tools > Vivado 2013.4 > Vivado HLS > Vivado HLS 2013.4**

A Getting Started GUI will appear.

**4-1-2.** In the Getting Started section, click on **Create New Project.** The New Vivado HLS Project wizard opens.

**4-1-3.** Click **Browse…** button of the **Location** field, browse to **c:\xup\sys_design\labs\lab6,** and then click **OK.**

**4-1-4.** For Project Name, type ***fir.prj***

**4-1-5.** Click **Next.**

**4-1-6.** In the *Add/Remove Files* for the source files, type **fir** as the function name (the provided source file contains the function, to be synthesized, called fir).

**4-1-7.** Click the **Add Files…** button, select *fir.c* and *fir_coef.dat* files from the **c:\xup\sys_design\sources\lab6** folder, and then click **Open**.

**4-1-8.** Click **Next**.

**4-1-9.** In the *Add/Remove Files* for the testbench, click the **Add Files…** button, select *fir_test.c* file from the **c:\xup\sys_design\sources\lab6** folder and click **Open.**

**4-1-10.** Click **Next.**

**4-1-11.** In the *Solution Configuration* page, leave *Solution* Name field as **solution1** and change the *clock period* to **8**. Leave *Uncertainty* field blank as it will take 0.1 as the default value.

**4-1-12.** Click on Part's Browse button, and select the following filters to select the **xc7z010clg400-1** part, and click **OK**:

Family: **Zynq**
Sub-Family: **Zynq**
Package: **clg400**
Speed Grade: **–1**

**4-1-13.** Click **Finish.**

You will see the created project in the Explorer view. Expand various sub-folders to see the entries under each sub-folder.

**4-1-14.** Double-click on the **fir.c** under the source folder to open its content in the information pane.

```
1 #include "fir.h"
2
3 void fir (
4   data_t *y,
5   data_t x
6   ) {
7   const coef_t c[N+1]={
8  #include "fir_coef.dat"
9     };
10
11
12   static data_t shift_reg[N];
13   acc_t acc;
14   int i;
15
16   acc=(acc_t)shift_reg[N-1]*(acc_t)c[N];
17   loop: for (i=N-1;i!=0;i--) {
18     acc+=(acc_t)shift_reg[i-1]*(acc_t)c[i];
19     shift_reg[i]=shift_reg[i-1];
20   }
21   acc+=(acc_t)x*(acc_t)c[0];
22   shift_reg[0]=x;
23   *y = acc >> 15;
24 }
```

**Figure 10. The design under consideration**

The FIR filter expects x as a sample input and pointer to the computed sample out. Both of them are defined of data type data_t. The coefficients are loaded in array c of type coef_t from the file called fir_coef.dat located in the current directory. The sequential algorithm is applied and accumulated value (sample out) is computed in variable acc of type acc_t.

**4-1-15.** Double-click on the **fir.h** in the *outline* tab to open its content in the information pane.

```
1 #ifndef _FIR_H_
2 #define _FIR_H_
3 #include "ap_cint.h"
4 #define N   58
5 #define SAMPLES N+10 // just few more samples then number of taps
6 typedef short   coef_t;
7 typedef short   data_t;
8 typedef int38   acc_t;
9 #endif
10
```

**Figure 11. The header file**

The header file includes ap_cint.h so user defined data width (of arbitrary precision) can be used. It also defines number of taps (N), number of samples to be generated (in the testbench), and data types coef_t, data_t, and acc_t. The coef_t and data_t are short (16 bits). Since the algorithm iterates (multiply and accumulate) over 59 taps, there is a possibility of bit growth of 6 bits and hence acc_t is defined as int38. Since the acc_t is bigger than sample and coefficient width, they have to cast before being used (like in lines 16, 18, and 21 of fir.c).

**4-1-16.** Double-click on the **fir_test.c** under the testbench folder to open its content in the information pane.

Notice that the testbench opens fir_impulse.dat in write mode, and sends an impulse (first sample being 0x8000.

## Run C Simulation — Step 5

### 5-1. Run C simulation to observe the expected output.

**5-1-1.** Select **Project > Run C Simulation** or click on  from the tools bar buttons, and Click **OK** in the *C Simulation Dialog* window.

The testbench will be compiled using apcc compiler and csim.exe file will be generated. The csim.exe will then be executed and the output will be displayed in the console view.

```
Starting C simulation ...
C:/Xilinx/Vivado_HLS/2013.4/bin/vivado_hls.bat C:/xup/sys_design/labs/lab6/fir.prj/solution1/
@I [LIC-101] Checked out feature [VIVADO_HLS]
@I [HLS-10] Running 'C:/Xilinx/Vivado_HLS/2013.4/bin/unwrapped/win64.o/vivado_hls.exe'
            for user 'parimalp' on host 'xsjparimalp30' (Windows NT_amd64 version 6.1) on Thu
            in directory 'C:/xup/sys_design/labs/lab6'
@I [HLS-10] Opening project 'C:/xup/sys_design/labs/lab6/fir.prj'.
@I [HLS-10] Opening solution 'C:/xup/sys_design/labs/lab6/fir.prj/solution1'.
@I [SYN-201] Setting up clock 'default' with a period of 8ns.
@I [HLS-10] Setting target device to 'xc7z010clg400-1'
   Compiling(apcc) ../../../../../../sources/lab6/fir_test.c in debug mode
@I [LIC-101] Checked out feature [VIVADO_HLS]
@I [HLS-10] Running 'c:/Xilinx/Vivado_HLS/2013.4/bin/unwrapped/win64.o/apcc.exe'
            for user 'parimalp' on host 'xsjparimalp30' (Windows NT_amd64 version 6.1) on Thu
            in directory 'C:/xup/sys_design/labs/lab6/fir.prj/solution1/csim/build'
clang: warning: c:/Xilinx/Vivado_HLS/2013.4/bin/unwrapped/win64.o/apcc.exe: 'linker' input ur
gcc.exe: warning: c:/Xilinx/Vivado_HLS/2013.4/bin/unwrapped/win64.o/apcc.exe: linker input fi
@I [APCC-3] Tmp directory is apcc_db
@I [APCC-1] APCC is done.
@I [LIC-101] Checked in feature [VIVADO_HLS]
   Compiling(apcc) ../../../../../../sources/lab6/fir.c in debug mode
@I [LIC-101] Checked out feature [VIVADO_HLS]
@I [HLS-10] Running 'c:/Xilinx/Vivado_HLS/2013.4/bin/unwrapped/win64.o/apcc.exe'
            for user 'parimalp' on host 'xsjparimalp30' (Windows NT_amd64 version 6.1) on Thu
            in directory 'C:/xup/sys_design/labs/lab6/fir.prj/solution1/csim/build'
clang: warning: c:/Xilinx/Vivado_HLS/2013.4/bin/unwrapped/win64.o/apcc.exe: 'linker' input ur
gcc.exe: warning: c:/Xilinx/Vivado_HLS/2013.4/bin/unwrapped/win64.o/apcc.exe: linker input fi
@I [APCC-3] Tmp directory is apcc_db
@I [APCC-1] APCC is done.
@I [LIC-101] Checked in feature [VIVADO_HLS]
   Generating csim.exe
0 -32768 378
1 0 73
2 0 -27
3 0 -170
```

**Figure 12. Initial part of the generated output in the Console view**

You should see the filter coefficients being computed.

## Synthesize the Design — Step 6

### 6-1. Synthesize the design with the defaults. View the synthesis results and answer the question listed in the detailed section of this step.

**6-1-1.** Select **Solution > Run C Synthesis > Active Solution** to start the synthesis process.

**6-1-2.** When synthesis is completed, several report files will become accessible and the Synthesis Results will be displayed in the information pane.

**6-1-3.** The Synthesis Report shows the performance and resource estimates as well as estimated latency in the design.

**6-1-4.** Using scroll bar on the right, scroll down into the report and answer the following question.

## Question 1

Estimated clock period: _____
Worst case latency: _____
Number of DSP48E used: _____
Number of BRAMs used: _____
Number of FFs used: _____
Number of LUTs used: _____

**6-1-5.** The report also shows the top-level interface signals generated by the tools.

**Interface**

**Summary**

| RTL Ports | Dir | Bits | Protocol | Source Object | C Type |
|---|---|---|---|---|---|
| ap_clk | in | 1 | ap_ctrl_hs | fir | return value |
| ap_rst | in | 1 | ap_ctrl_hs | fir | return value |
| ap_start | in | 1 | ap_ctrl_hs | fir | return value |
| ap_done | out | 1 | ap_ctrl_hs | fir | return value |
| ap_idle | out | 1 | ap_ctrl_hs | fir | return value |
| ap_ready | out | 1 | ap_ctrl_hs | fir | return value |
| y | out | 16 | ap_vld | y | pointer |
| y_ap_vld | out | 1 | ap_vld | y | pointer |
| x | in | 16 | ap_none | x | scalar |

**Figure 13. Generated interface signals**

You can see the design expects x input as 16-bit scalar and outputs y via pointer of the 16-bit data. It also has ap_vld signal to indicate when the result is valid.

## 6-2. Add PIPELINE directive to loop and re-synthesize the design. View the synthesis results.

**6-2-1.** Make sure that the **fir.c** is open in the information view.

**6-2-2.** Select the **Directive** tab, and apply the **PIPELINE** directive to the **loop**.

**6-2-3.** Select **Solution > Run C Synthesis > Active Solution** to start the synthesis process.

**6-2-4.** When synthesis is completed, the Synthesis Results will be displayed in the information pane.

**6-2-5.** Note that the latency has reduced to 64 clock cycles. The DSP48 and BRAM consumption remains same; however, LUT and FF consumptions have changed.

## Run RTL/C CoSimulation Step 7

### 7-1. Run the RTL/C Co-simulation, selecting SystemC and skipping VHDL and Verilog. Verify that the simulation passes.

**7-1-1.** Select **Solution > Run C/RTL Cosimulation** or click on the button to open the dialog box so the desired simulations can be run.

A C/RTL Co-simulation Dialog box will open.

**7-1-2.** Click **OK** to run the SystemC simulation.

The Co-simulation will run, generating and compiling several files, and then simulating the design. In the console window you can see the progress. When done the RTL Simulation Report shows that it was successful and the latency reported was 64.

## Setup IP-XACT Adapter Step 8

### 8-1. Add RESOURCE directives to create AXI4LiteS adapters so IP-XACT adapter can be generated during the RTL Export step.

**8-1-1.** Make sure that **fir.c** file is open and in focus in the information view.

**8-1-2.** Select the **Directive** tab.

**8-1-3.** Right-click **x**, and click on **Insert Directive…**.

**8-1-4.** In the **Vivado HLS Directive Editor** dialog box, select RESOURCE using the drop-down button.

**8-1-5.** Click on the button beside **core (required)**. Available cores list will pop-up. Select **AXI4LiteS [adapter]**, and click **OK**.

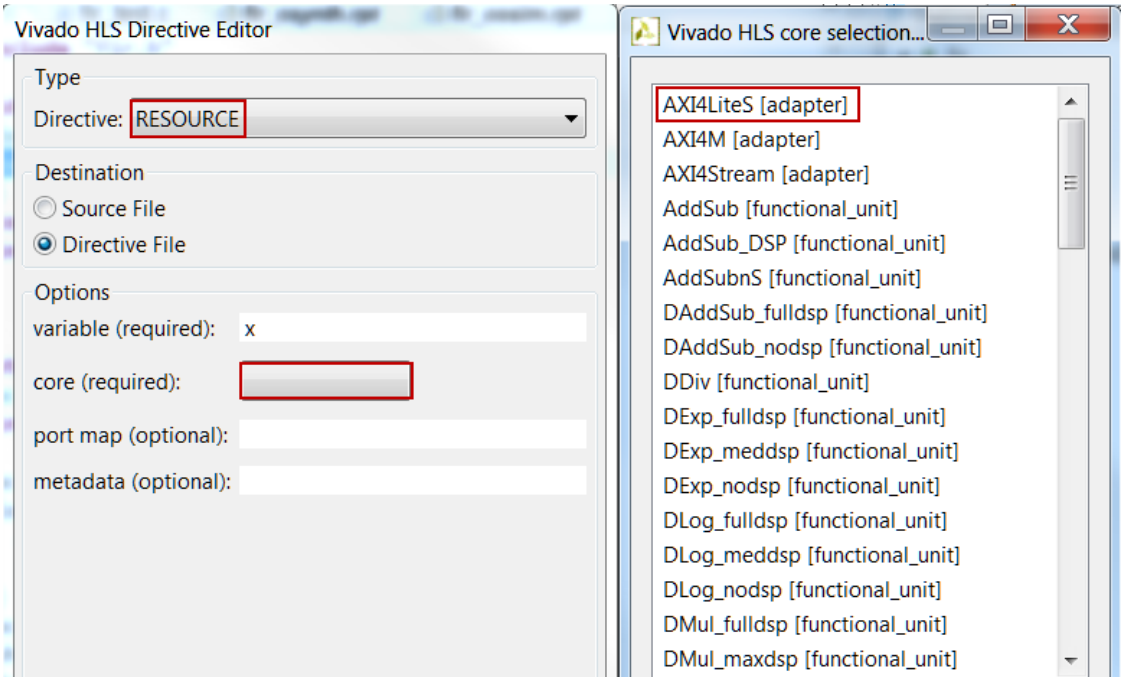



**Figure 14. Selecting the AXI4LiteS adapter**

**8-1-6.** In the **metadata (optional)** field, type **-bus_bundle fir_io** to associate input, output, and handshaking signals (done through next two steps) into one AXI4Lite adapter called **fir_io**. Click **OK**.

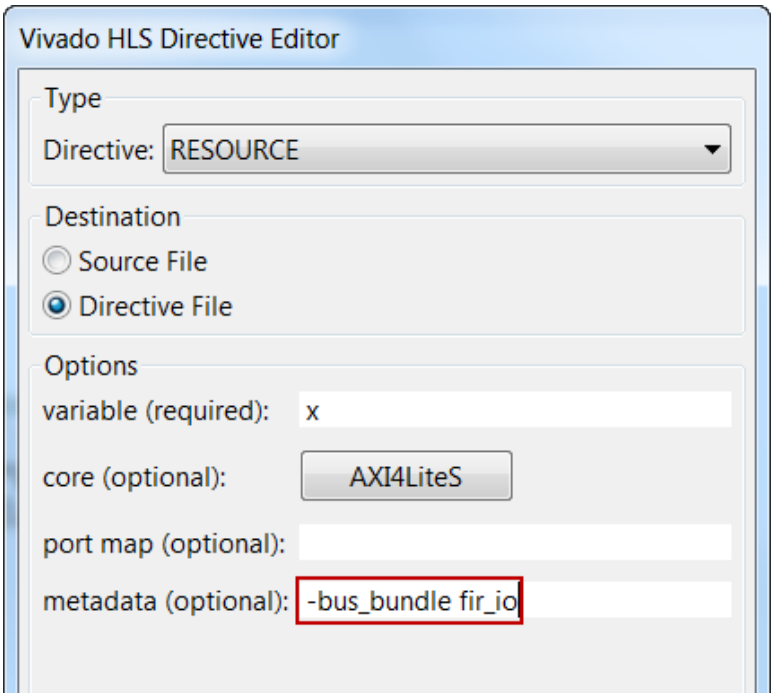



**Figure 15. Applying metadata to assign x input to AXI4Lite adapter**

**8-1-7.** Similarly, apply the **RESOURCE** directive (including metadata) to the **y** output.

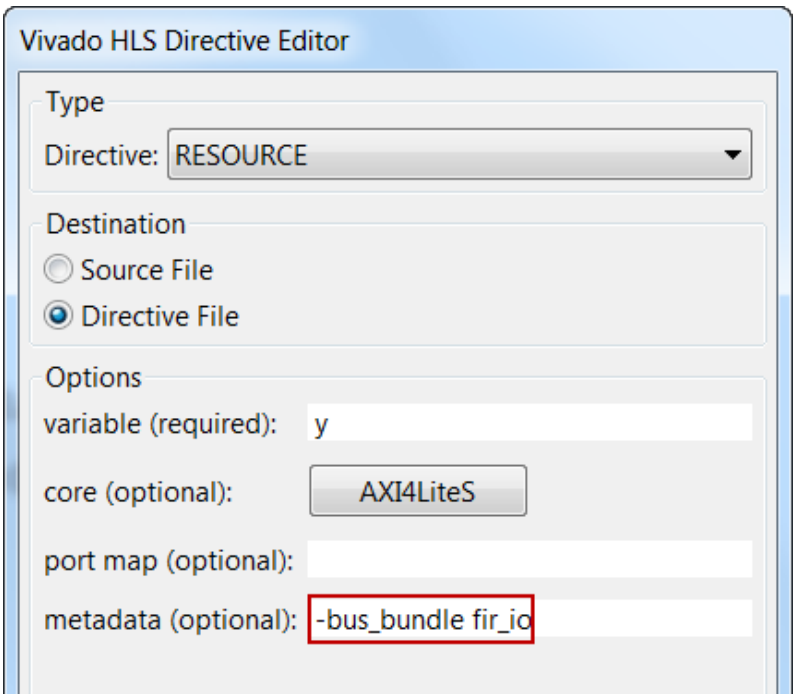



**Figure 16. Applying metadata to assign y output to AXI4Lite adapter**

**8-1-8.** Apply the **RESOURCE** directive to the top-level module **fir** to include ap_start, ap_done, and ap_idle signals as part of bus adapter (the variable name shown will be **return**). Include metadata information too.

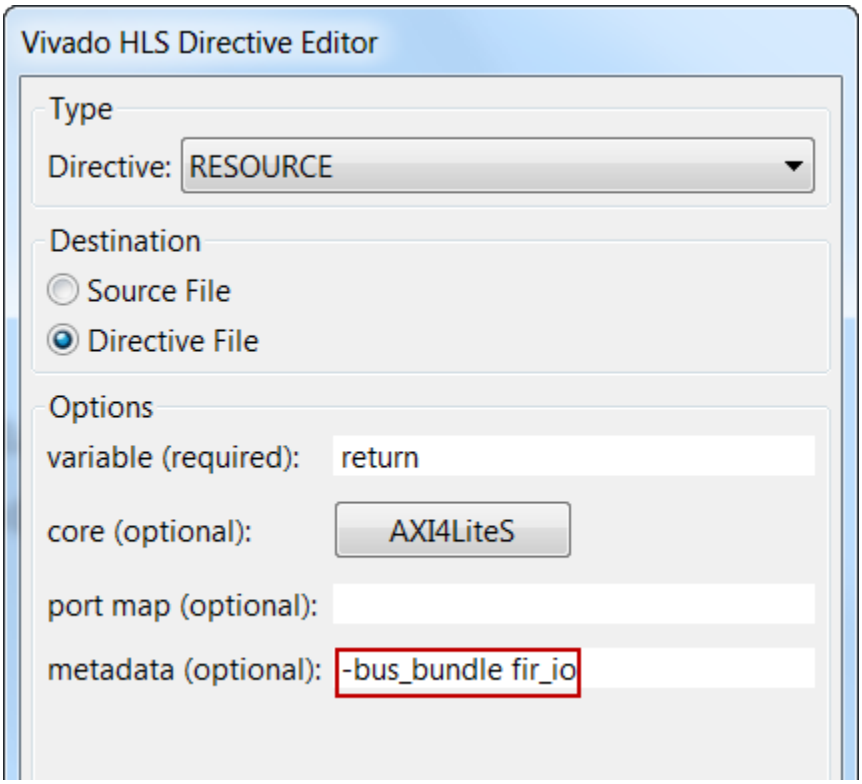



**Figure 17. Applying metadata to assign function control signals to AXI4Lite adapter**

Note that the above steps 5-1-3 through 5-1-7 will create address maps for x, y, ap_start ap_valid, ap_done, and ap_idle, which can be accessed via software. Alternately, ap_start, ap_valid, ap_done, ap_idle signals can be generated as separate ports on the core by not applying RESOURCE directive to the top-level module fir. These ports will then have to be connected in a processor system using available GPIO IP.

## Generate the IP-XACT Adapter Step 9

### 9-1. Re-synthesize the design as directives have been added. Run the RTL Export to generate the IP-XACT adapter.

**9-1-1.** Since the directives have been added, it is safe to re-synthesize the design. Select **Solution > Run C Synthesis > Active Solution.**

**9-1-2.** Once the design is synthesized, select **Solution > Export RTL** to open the dialog box so the desired IP can be generated.

An Export RTL Dialog box will open.

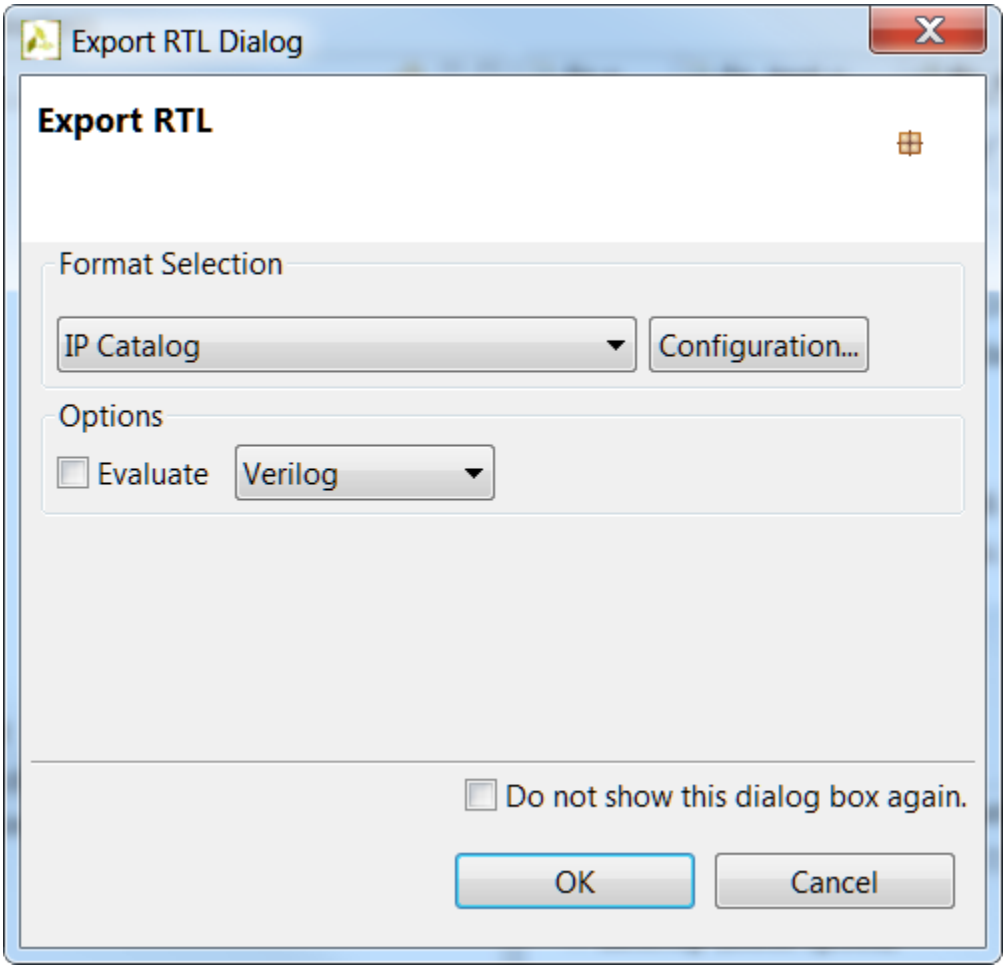



**Figure 18. Export RTL Dialog**

**9-1-3.** Click **OK** to generate the IP-XACT adapter.

**9-1-4.** When the run is completed, expand the **impl** folder in the *Explorer* view and observe various generated directories; drivers, ip, verilog and vhdl.

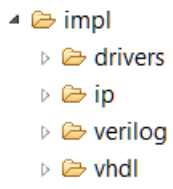



**Figure 19. IP-XACT adapter generated**

Expand the ip directory and observe several files and sub-directories. One of the sub-directory of interest is drivers directory which consists header, c, tcl, mdd, and makefile files. Another file of interest is the zip file, which is the ip repository file that can be imported in an IP Integrator design

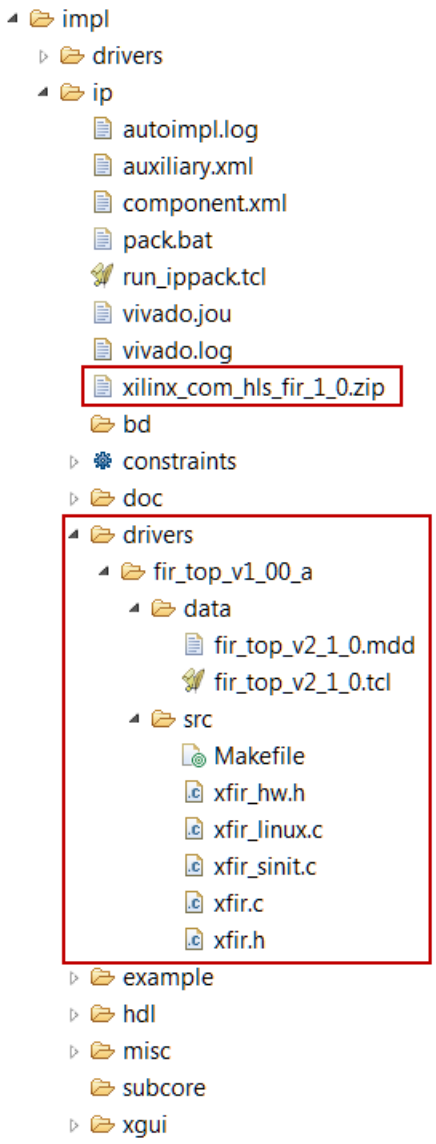



**Figure 20. Adapter's drivers directory**

**9-1-5.** Close Vivado HLS by selecting **File > Exit.**

# Update the Vivado Project with the FIR IP — Step 10

## 10-1. In Vivado, set the HLS IP to the IP Catalog settings.

**10-1-1.** In the *Flow Navigator* pane, click **Project Settings** under *Project Manager.*

**10-1-2.** Click the **IP** icon.

**10-1-3.** Click the **Add Repository…** button. Browse to **c:\xup\sys_design\labs\lab6\fir.prj\solution1\impl\ip** and click **Select**.

The directory will be scanned and added in the *IP Repositories* window, and *Fir* IP entry will be displayed in the *IP in Selected Repository* window.

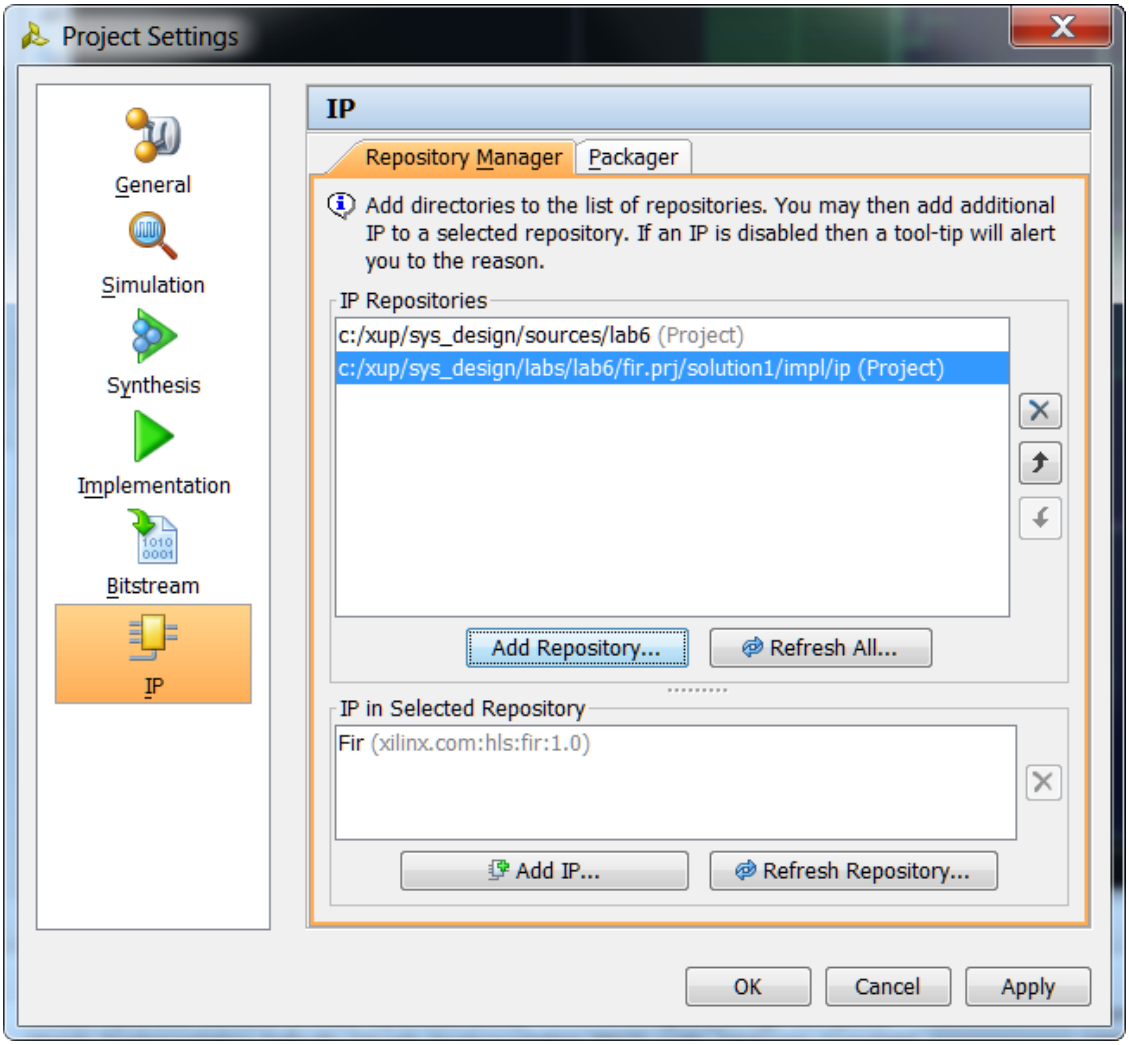



**Figure 21. Setting path to IP Repositories**

**10-1-4.** Click **OK** to accept the settings.

## 10-2. Open the block design. Instantiate fir_top core twice, one for each side channel, into the processing system naming the instances as fir_left and fir_right.

**10-2-1.** Select **Open Block Design > system.bd** from the *Flow Navigator* pane.

**10-2-2.** Click the Add IP icon and search for **FIR** in the catalog by typing **FIR** and double-click on the *FIR* entry to add an instance.

**10-2-3.** Click on the **Add IP to Block Design** button if presented.

Notice that the added IP has HLS logo in it indicating that this was created by Vivado HLS.

**10-2-4.** Select the added instance in the diagram, and change its instance name to **fir_left** by typing it in the *Name* field of the *Block Properties* form in the left.

**10-2-5.** Similarly, add another instance of the HLS IP, and name it **fir_right**.

**10-2-6.** Click on **Run Connection Automation**, and select **/fir_left/S_AXI_FIR_IO** and click **OK.**

**10-2-7.** Similarly, click on **Run Connection Automation** again, and select **/fir_right/S_AXI_FIR_IO** and click **OK.**

At this stage the design should look like shown below (you may have to click the regenerate button [ ]).

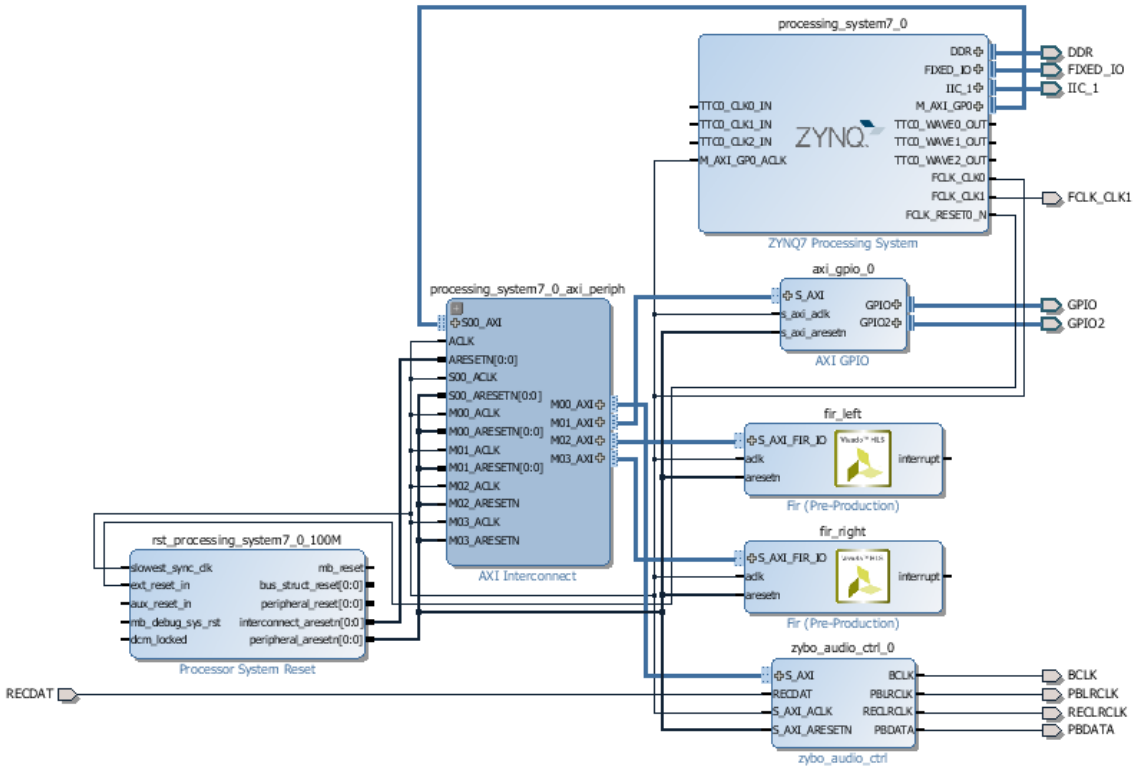



**Figure 22. The complete hardware design**

### 10-3. Verify addresses and validate the design. Generate the system_wrapper file.

**10-3-1.** Click on the *Address Editor*, and expand the **processing_system7_0 > Data** if necessary.

The generated address map should look like as shown below.

Diagram × | Address Editor ×

| Cell | Interface Pin | Base Name | Offset Address | Range | High Address |
|---|---|---|---|---|---|
| processing_system7_0 | | | | | |
| Data (32 address bits : 4G) | | | | | |
| zybo_audio_ctrl_0 | S_AXI | reg0 | 0x40000000 | 64K | 0x4000FFFF |
| axi_gpio_0 | S_AXI | Reg | 0x41200000 | 64K | 0x4120FFFF |
| fir_left | S_AXI_FIR_IO | Reg | 0x43C00000 | 64K | 0x43C0FFFF |
| fir_right | S_AXI_FIR_IO | Reg | 0x43C10000 | 64K | 0x43C1FFFF |

**Figure 23. Generated address map**

**10-3-2.** Run *Design Validation* (**Tools > Validate Design**) and verify there are no errors

**10-3-3.** In the *sources* view, right-click on the block diagram file, **system.bd**, and select **Create HDL Wrapper** to update the HDL wrapper file. When prompted, click **OK** with the *Let Vivado manage wrapper and auto-update* option.

**10-3-4.** Click on the **Generate Bitstream** in the Flow Navigator to run the synthesis, implementation, and bitstream generation processes.

**10-3-5.** Click **Save** to save the design and **Yes** to re-run the processes, if prompted.

**10-3-6.** When the bit generation is completed, click **OK** to open the implemented design.

## Export to SDK and Profile the Application with Hardware Step 11

### 11-1. Export the hardware along with the generated bitstream to SDK.

To Export the hardware, the block diagram must be open and the Implemented design must be open.

**11-1-1.** If it is not already open, click **Open Block Design** (under IP Integrator in the Flow Navigator)

**11-1-2.** If it is not already open, click **Open Implemented Design** (under Implementation)

**11-1-3.** Start *SDK* by clicking **File > Export > Export Hardware for SDK**...

The export GUI will be displayed.

**11-1-4.** Click Yes to overwrite the exported module file.

**11-1-5.** Right-click on the standalone_bsp_0 project in the Project Explorer and select Board Support Package Settings.

**11-1-6.** Select the *drivers* pane and see if fir_left and fir_right entries are present. If not, then close the settings form and close the SDK, and re-open the SDK again.

**11-1-7.** Make sure that the **generic** driver is assigned to both the instances. Click **OK**.

| Component | Component Type | Driver | Dr... |
|---|---|---|---|
| ps7_cortexa9_0 | ps7_cortexa9 | cpu_cortexa9 | 1.... |
| axi_gpio_0 | axi_gpio | gpio | 3.... |
| fir_left | fir | generic | 1.... |
| fir_right | fir | generic | |
| ps7_afi_0 | ps7_afi | generic | 1.... |

**Figure 24. Assigning the generic driver**

## 11-2. Remove the user defined SW_PROFILE symbol and add the HW_PROFILE symbol.

**11-2-1.** Select the *lab6* application, right-click, and select **C/C++ Build Settings**.

**11-2-2.** Under the **ARM gcc compiler** group, select the **Symbols** sub-group**,** select **SW_PROFILE**, and delete it by clicking on the delete button.

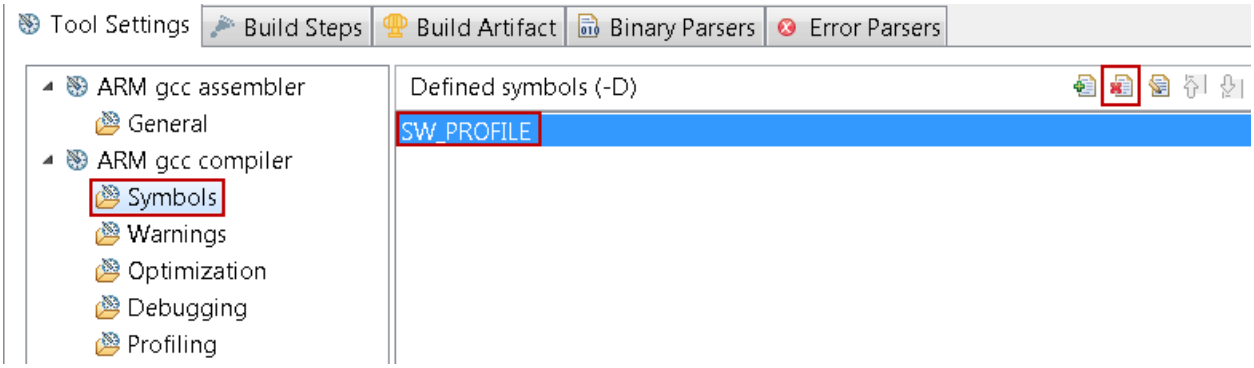



**Figure 25. Deleting the user-defined symbol**

**11-2-3.** Add the **HW_PROFILE** symbol.

This will allow us to profile the hardware IP of the FIR application.

**11-2-4.** Select **lab6** in the project view, right-click the *src* folder, and select **Import.**

**11-2-5.** Expand **General** category and double-click on **File System.**

**11-2-6.** Browse to **C:\xup\sys_design\labs\lab6\fir.prj\solution1\impl\drivers\fir_top_v1_00_a\src** folder and click **OK.**

**11-2-7.** Select **fir_hw.h** and click **Finish** to add the file to the project.

### 11-3. Make sure that the SW[1] is in ON position. Power ON the board. Program the FPGA. Profile the application using the hardware FIR.

**11-3-1.** Make sure that the SW[1] is in the ON (up) position.

**11-3-2.** Power ON the board.

**11-3-3.** Select **Xilinx Tools > Program FPGA**

**11-3-4.** Click the **Program** button.

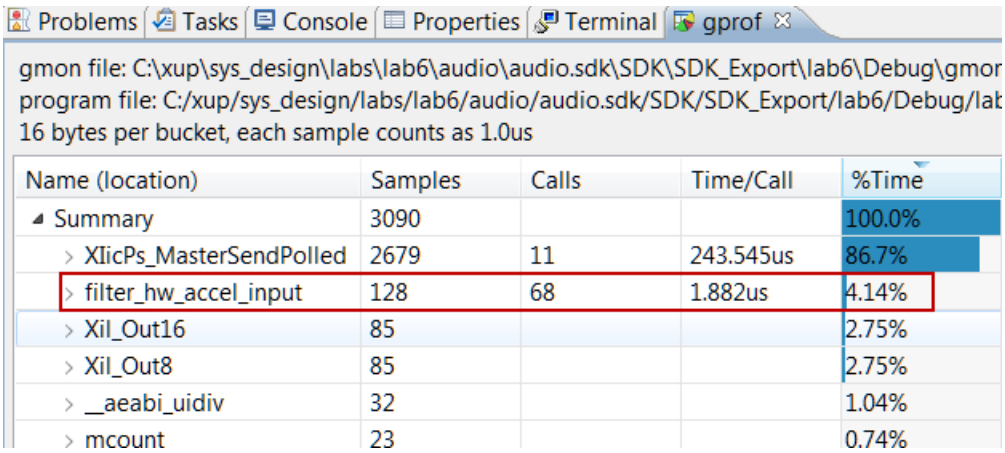
**11-3-5.** Select **Run > Run Configurations** and click the **Run** button to profile the application.

**11-3-6.** Click **OK** when prompted.

**11-3-7.** Invoke *gprof* by double-clicking gmon.out entry under *lab6* > *Debug* folder in the Project Explorer view of SDK, select the **Sorts samples per function** output, and sort the %Time column.

Notice that the output now shows filter_hw_accel_input function call instead of the fir_software function call. Note that the number of calls to the filter function has not changed but the average time spent per call is 1.882 us as the filtering is done in the hardware instead of the software.

Also notice that the amount of time spent in the filtering function reduced from about 13.85% to 4.14%.




| Name (location) | Samples | Calls | Time/Call | %Time |
|---|---|---|---|---|
| Summary | 3090 | | | 100.0% |
| XIicPs_MasterSendPolled | 2679 | 11 | 243.545us | 86.7% |
| filter_hw_accel_input | 128 | 68 | 1.882us | 4.14% |
| Xil_Out16 | 85 | | | 2.75% |
| Xil_Out8 | 85 | | | 2.75% |
| __aeabi_uidiv | 32 | | | 1.04% |
| mcount | 23 | | | 0.74% |

**Figure 26. Profiling the application with the hardware IP**

## Verify the Design in Hardware — Step 12

### 12-1. Turn-OFF the profiling in the Board Support Package setting, lab6's C/C++ Build Settings, and Run Configuration settings.

**12-1-1.** Right-click on the **standalone_bsp_0** entry in the Explorer view and select *Board Support Package Settings*.

**12-1-2.** Select the **Overview > standalone** entry in the left pane, click on the drop-down arrow of the *enable_sw_intrusive_profiling Value* field and select **false**.

**12-1-3.** Select the **Overview > drivers > cpu_cortexa9** and remove **–pg** from the *extra_compiler_flags Value* field.

**12-1-4.** Click **OK** to accept the settings and update the BSP.

**12-1-5.** Right-click on the *lab6* project and select *C/C++ Build Settings*. Select the *Profiling* settings and **uncheck** the *Enable Profiling* check box.

**12-1-6.** Select *Run* > *Run Configurations*. Select the *Profiling* tab, and **uncheck** the *Enable Profiling* option.

**12-1-7.** Click **Apply** and **Close**.

**12-1-8.** Select *lab6* > *Clean Project* to recompile everything.

### 12-2. Connect an audio patch cable between the Line In jack of the board and the Speaker (header) out jack of a PC. Connect a headphone to the Line Out jack of the board. Set the SW[1] in the OFF position. Play the provided corrupted_music_4kHz.wav file.

**12-2-1.** Connect an audio patch cable between the Line In jack of the board and the Speaker (header) out jack of a PC .

**12-2-2.** Connect a headphone to the Line Out jack on the board.

**12-2-3.** Set the SW[1] in the OFF position.

**12-2-4.** Double-click **corrupted_music_4KHz.wav** or some other wave file of interest to play it using the installed media player. Place it in the continuous play mode.

**12-2-5.** Right-click on the *lab6* in the Project Explorer pane and select **Run As > Launch On Hardware(GDB)**.

The program will be downloaded and run. Set the PC volume to about 25%. If you want to listen to corrupted signal then set the SW0 OFF. To listened the filtered signal set the SW0 ON.

**12-2-6.** When done, power OFF the board, and exit SDK and Vivado using **File > Exit**.

## Conclusion

In this lab, you profiled a software application after creating a processor system using IP Integrator. Then you created a Vivado HLS project and added RESOURCE directive to create an IP-XACT adapter. You generated the IP-XACT adapter during the export phase. You then updated the processor system using the generated IP-XACT adapter, and profiled the system with the provided application.

## Answers

1. Answer the following questions:

| | |
|---|---|
| Estimated clock period: | 6.38 ns |
| Worst case latency: | 175 clock cycles |
| Number of DSP48E used: | 3 |
| Number of BRAMs used: | 2 |
| Number of FFs used: | 115 |
| Number of LUTs used: | 77 |

# Appendix

## Create a Project using Vivado GUI — Step 13

### 13-1. Launch Vivado and create an empty project targeting the ZYBO (having xc7z010clg400-1 device) board and using the Verilog language.

**13-1-1.** Open Vivado by selecting **Start > All Programs > Xilinx Design Tools > Vivado 2013.4 > Vivado 2013.4**

**13-1-2.** Click **Create New Project** to start the wizard. You will see the *Create a New Vivado Project* dialog box. Click **Next**.

**13-1-3.** Click the Browse button of the *Project Location* field of the **New Project** form, browse to **c:\xup\sys_design\labs**, and click **Select**.

**13-1-4.** Append **lab6** in the *Project Location* field.

**13-1-5.** Enter **audio** in the *Project Name* field. Make sure that the *Create Project Subdirectory* box is checked. Click **Next**.

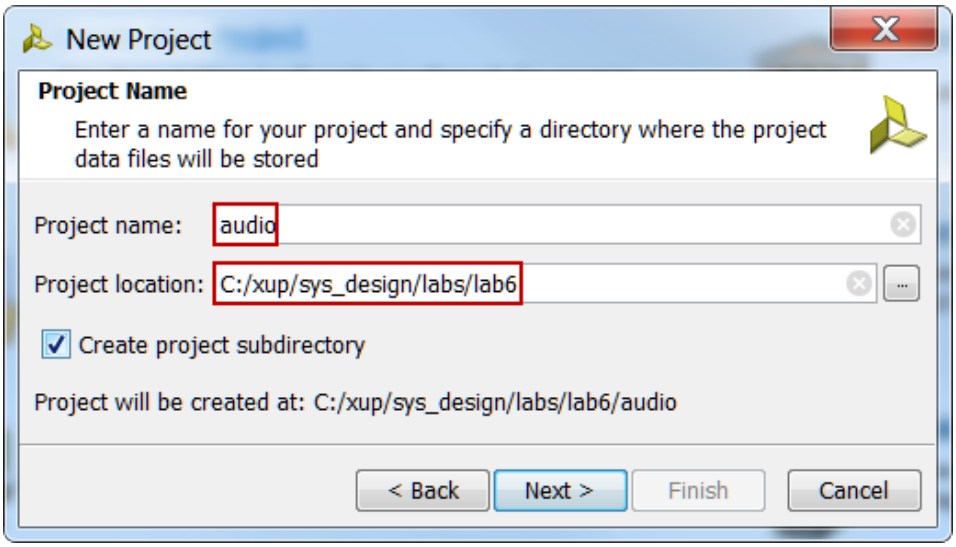



**Figure A-1. Project Name entry**

**13-1-6.** Select **RTL Project** in the *Project Type* form, and click **Next**.

**13-1-7.** Select **Verilog** as the *Target language* and *Simulator Language* in the *Add Sources* form, and click **Next**.

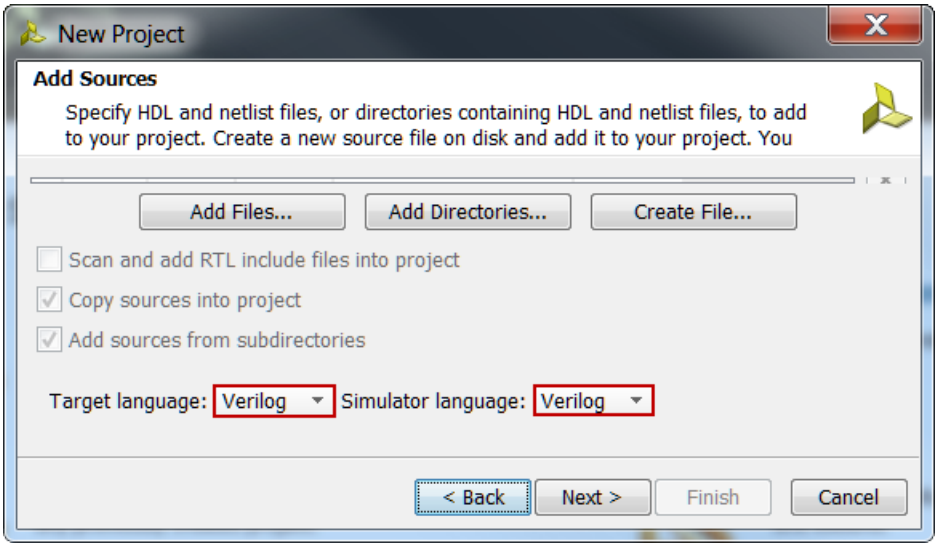



**Figure A-2. Add sources to new project**

**13-1-8.** Click **Next** two times to skip *Adding Existing IP* and *Add Constraints* dialog boxes

**13-1-9.** In the *Default Part* form, select *Parts*, select various filters as shown in the below figure, and select **xc7z010clg400-1**. Click **Next**.

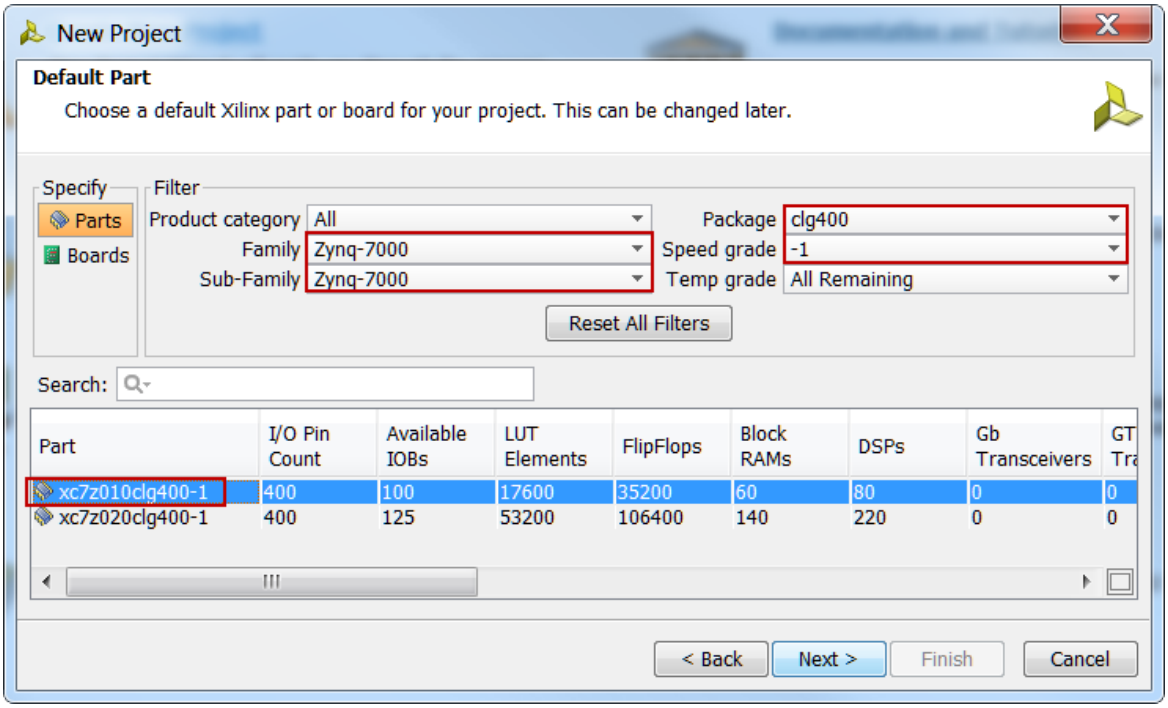



**Figure A-3. Part selection**

**13-1-10.** Check the *Project Summary* and click **Finish** to create an empty Vivado project.

## Creating the System Using the IP Integrator Step 14

### 14-1. Use the IP Integrator to create a new Block Design, add the ARM Cortex-A9 processor IP, and import the settings for the processor on the target board (since the board was not selected during the project creation).

**14-1-1.** In the Flow Navigator, click **Create Block Design** under IP Integrator

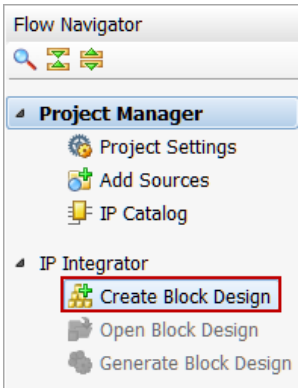



**Figure A-4. Create IP Integrator Block Diagram**

**14-1-2.** Enter **system** for the design name and click **OK**

**14-1-3.** IP from the catalog can be added in different ways. Click on *Add IP* in the message at the top of the *Diagram* panel, or click the *Add IP icon* in the block diagram side bar, press Ctrl + I, or right-click anywhere in the Diagram workspace and select Add IP.

**14-1-4.** Once the IP Catalog is open, type "zy" into the Search bar, find and double click on **ZYNQ7 Processing System** entry, or click on the entry and hit the Enter key to add it to the design.

The Zynq block will be added.

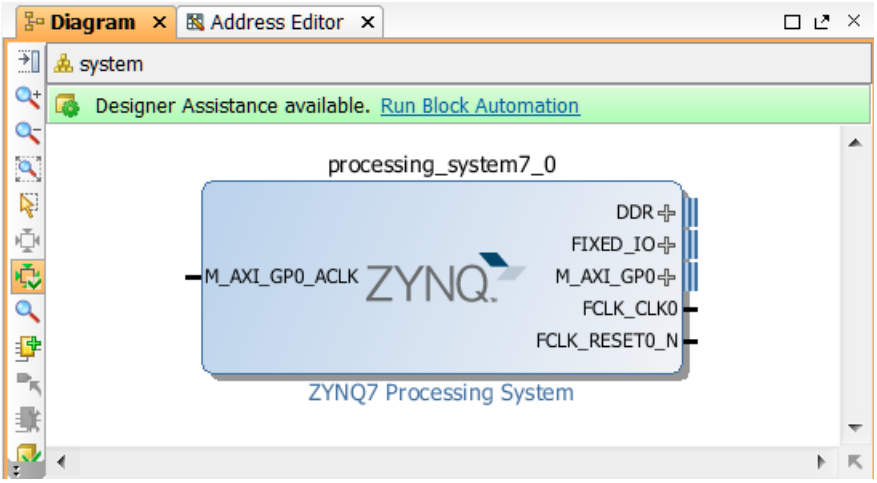



**Figure A-5. The Zynq IP Block**

**14-1-5.** Double-click on the added block to open its *Customization* window.

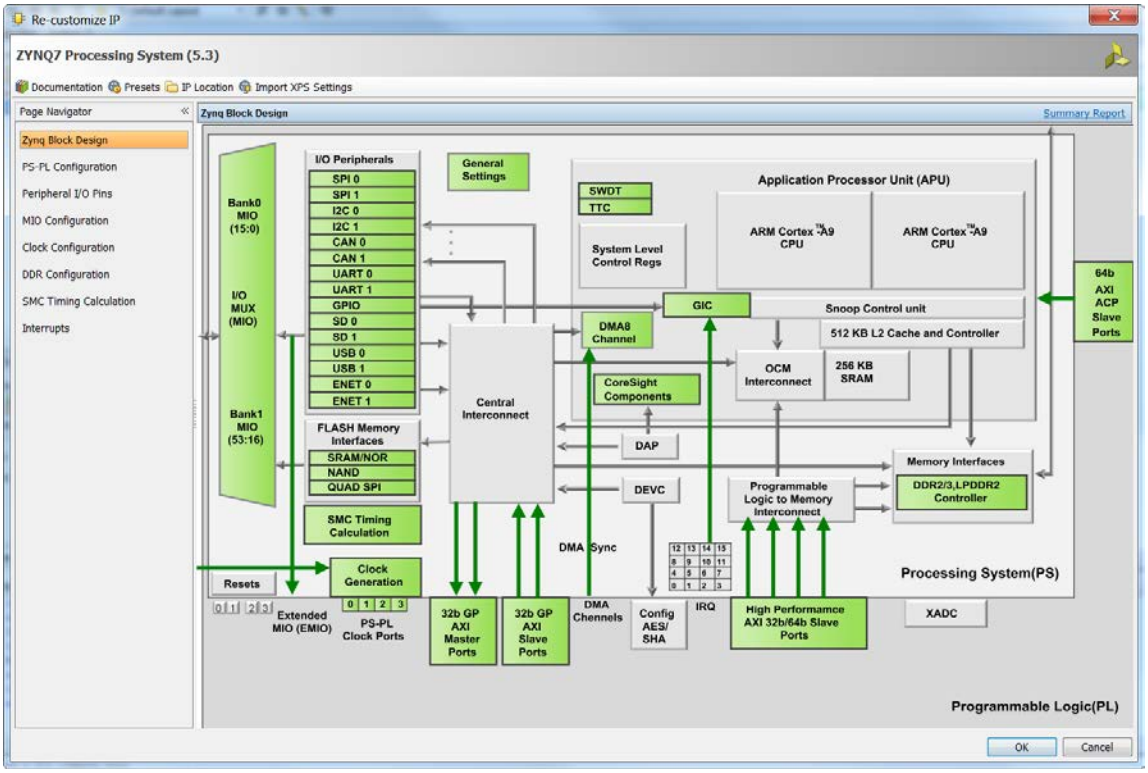



**Figure A-6. Default configuration form with no peripheral selected**

**14-1-6.** Click on the **Import XPS Settings** button on the top tools bar to import the setting for the ZYBO board using the provided xml file.

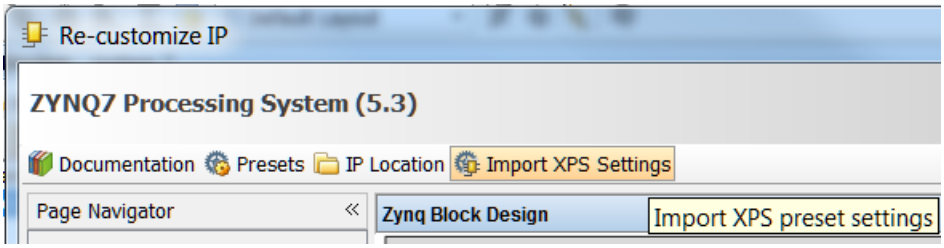



**Figure A-7. Importing xml file for the ZYBO board**

**14-1-7.** Click on the browse button, browse to c:\xup\sys_design\sources\lab6, select the provided *ps7_system_prj.xml* file, and click **OK**.

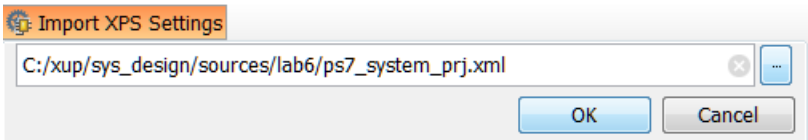



**Figure A-8. Selecting the xml file**

**14-1-8.** Click **OK**.

Notice now the *Customization* window shows selected peripherals (with tick marks).

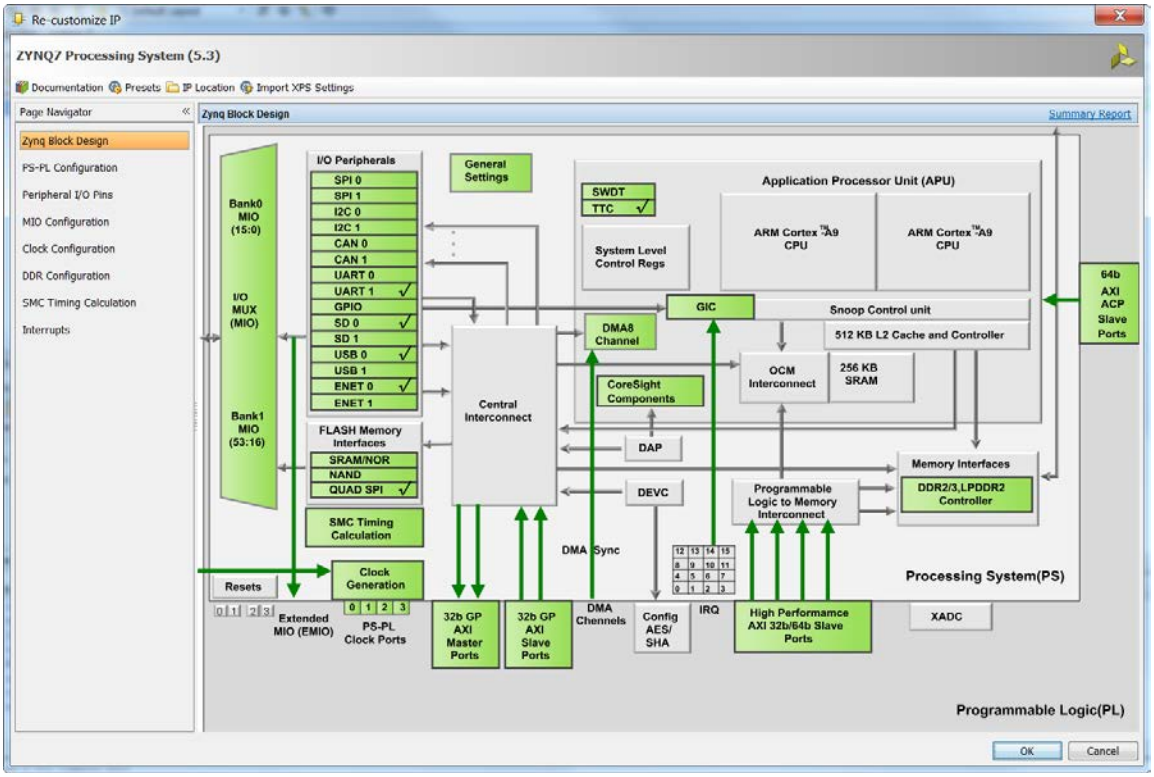



**Figure A-9. Imported peripherals settings**

**14-1-9.** Click **OK** to close the *Customization* window for now.

## 14-2. Run Block Automation process on the ZYNQ block instance. Customize the ARM Cortex-A9 processor based hardware system. Configure I/O Peripherals block to use I2C 1 peripheral. Enable FCLK_CLK1, the PL fabric clock and set its frequency to 10.000000 MHz.

**14-2-1.** Notice the message at the top of the Diagram window that Designer Assistance available. Click on **Run Block Automation** and select **/processing_system7_0**

**14-2-2.** Click **OK** when prompted to run automation.

Notice that external ports have been automatically added for the DDR and Fixed IO once Block Automation has been complete, Some of the other default ports are also added to the block.

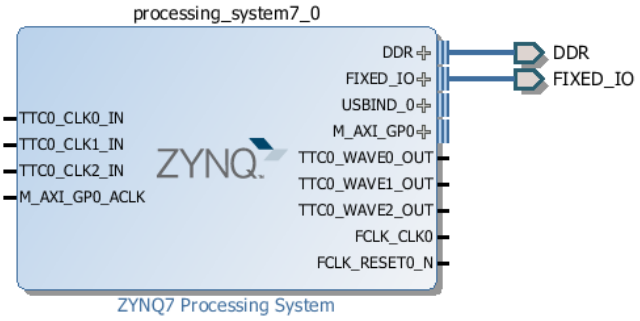



**Figure A-10. Zynq Block with DDR and Fixed IO ports**

**14-2-3.** In the block diagram, double click on the *Zynq* block to open the *Customization* window for the Zynq processing system.

**14-2-4.** Select the *MIO Configuration* tab on the left to open the configuration form and expand *I/O Peripheral* in the right pane. Uncheck the peripherals we don't need, and make sure that UART 1 and I2C 1 peripherals are checked.

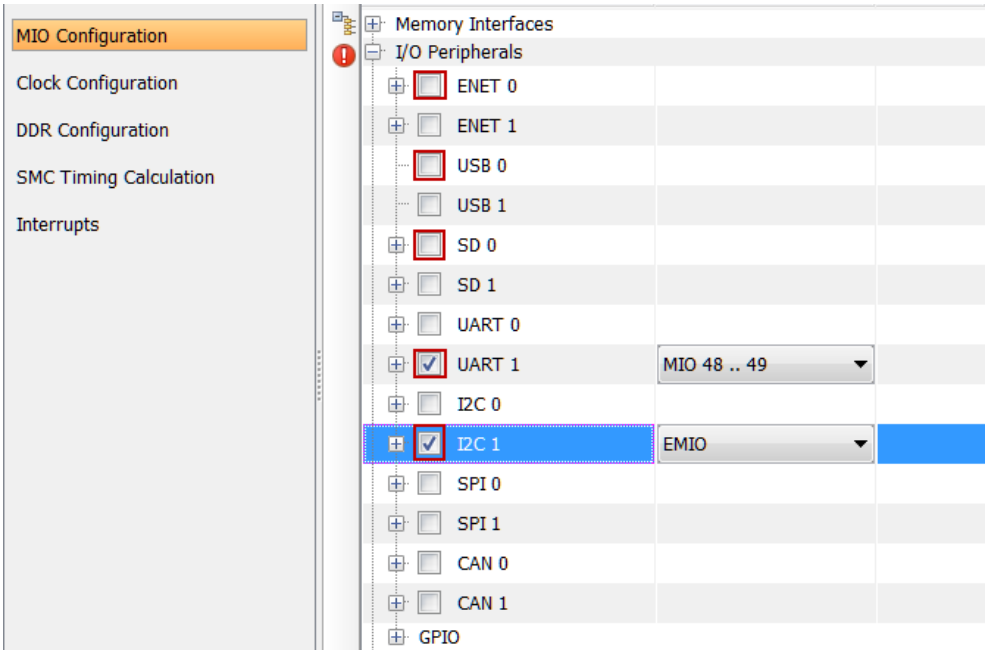



**Figure A-11. Selecting I2C 1**

**14-2-5.** Select the *Clock Configuration* in the left pane, expand *the PL Fabric Clocks* entry in the right, and click the check-box of *FCLK_CLK1*.

**14-2-6.** Change the *Requested Frequency* value of *FCLK_CLK1* to **12.288** MHz.

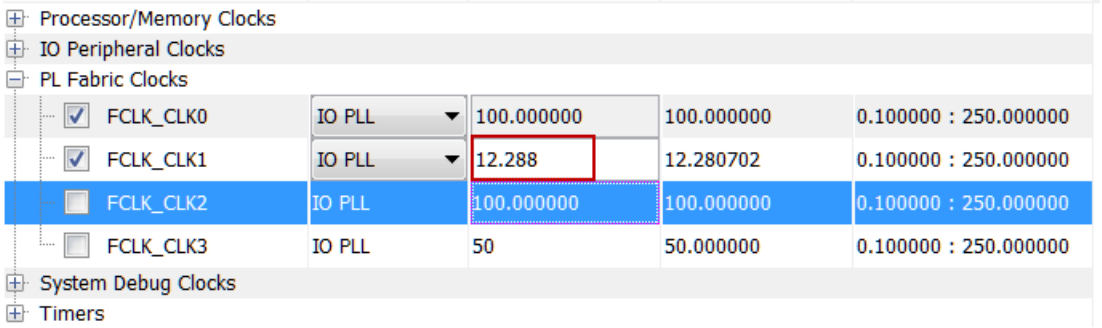



**Figure A-12. Enabling and setting the frequency of FCLK_CLK1**

**14-2-7.** Click **OK**.

Notice that the Zynq block only shows the necessary ports.

## 14-3. Add the provided I2C-based zybo_audio_ctrl IP to the IP Catalog

**14-3-1.** In the *Flow Navigator* pane, click **IP Catalog** under *Project Manager.*

The IP Catalog will open.

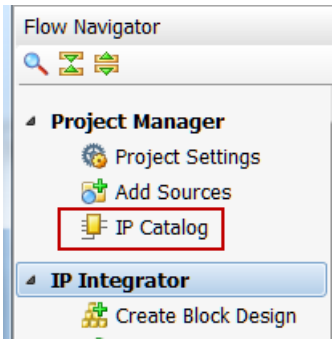



**Figure A-13. Invoking IP Catalog**

**14-3-2.** Click on the *IP Settings* button in the IP Catalog.

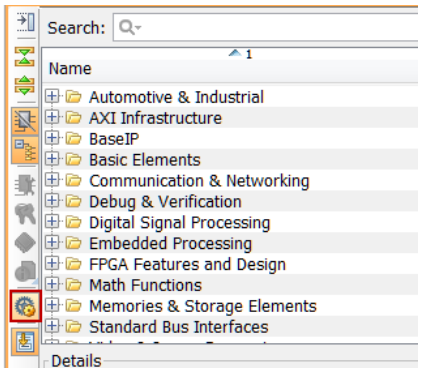



**Figure A-14. Invoking IP Settings**

**14-3-3.** Click on the **Add Repository…** button. Browse to **c:\xup\sys_design\sources\lab6** directory, and click **Select**.

Notice that *zybo_audio_ctrl* entry is displayed in the *IP in Selected Repository* field.

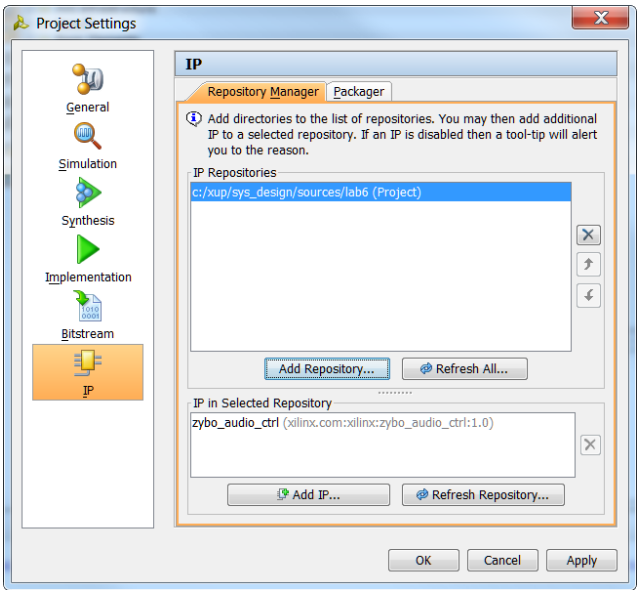



**Figure A-15. Adding IP repository for the provided I2C based zybo_audio_ctrl core**

**14-3-4.** Click **OK** to accept the settings.

## 14-4. Instantiate zybo_audio_ctrl and GPIO with width of 1 bit output only on channel 1 and width of 1 bit input only on channel 2. Run connection automation to connect them.

**14-4-1.** In the Diagram tab, click the Add IP button if the IP Catalog is not open and search for **AXI GPIO** in the catalog by typing **gpi** and double-click on the AXI GPIO entry to add an instance.

**14-4-2.** Click on the **Add IP to Block Design** button.

**14-4-3.** Double-click on the added instance and the **Re-Customize IP** GUI will be displayed.

**14-4-4.** Change the *Channel 1* width to 1. Check *All Outputs* box.

**14-4-5.** Check the *Enable Dual Channel* box, set the width to **2**, check *All Input* box, and click **OK**.

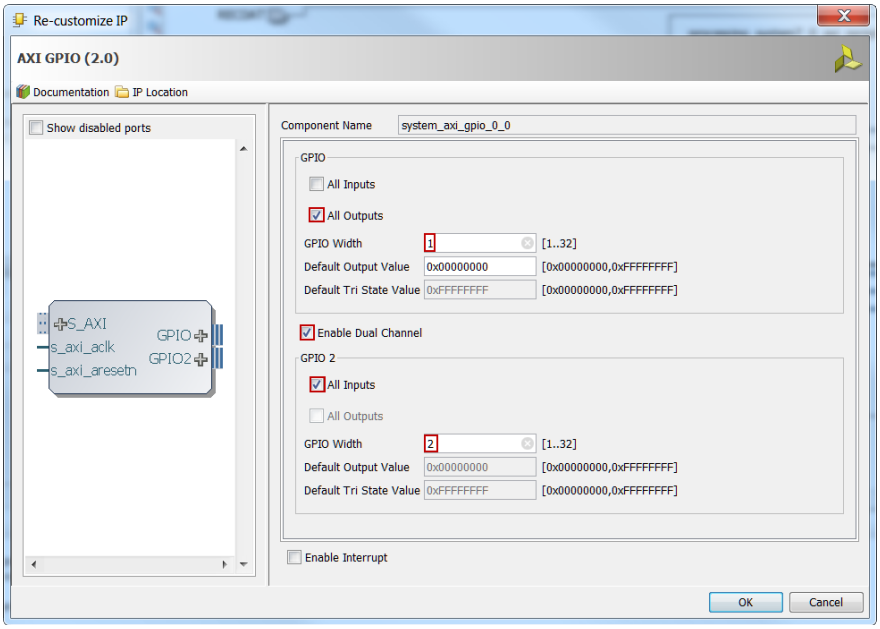



**Figure A-15. Configuring GPIO for two channels**

**14-4-6.** Similarly add an instance of the *zybo_audio_ctrl* IP, by typing **zyb** in the filter field, double-clicking the entry, and clicking the **Add IP to Block Design** button.

**14-4-7.** Notice that *Design assistance* is available. Click on **Run Connection Automation**, and select **/zybo_audio_ctrl_0/S_AXI**

**14-4-8.** Click **OK** to connect it to the M_AXI_GP0 interface.

Notice two additional blocks, *Processor System Reset*, and *AXI Interconnect* have automatically been added to the design.

**14-4-9.** Similarly, click on **Run Connection Automation**, and select **/axi_gpio_0/S_AXI** and click **OK**.

## 14-5. Make IIC_1, GPIO, FCLK_CLK1, and zybo_audio_ctrl ports external.

**14-5-1.** Select the **GPIO** interface of the *axi_gpio_0* instance, right-click on it and select **Make External** to create an external port. This will create the external port named *GPIO* and connect it to the peripheral.

**14-5-2.** Select the **GPIO2** interface of the *axi_gpio_0* instance, right-click on it and select **Make External** to create the external port.

**14-5-3.** Similarly, selecting one port at a time of the *zybo_audio_ctrl_0* instance, and make **RECDAT**, **PBDATA**, **BCLK**, **PBLRCLK,** and **RECLRCLK** ports external.

**14-5-4.** Similarly, make the **IIC_1** interface and **FCLK_CLK1** port of the *processing_system7_0* instance external.

**14-5-5.** Make the **FCLK_CLK1** port of the *processing_system7_0* instance external.

At this stage the design should look like shown below (you may have to click the regenerate [ ] button).

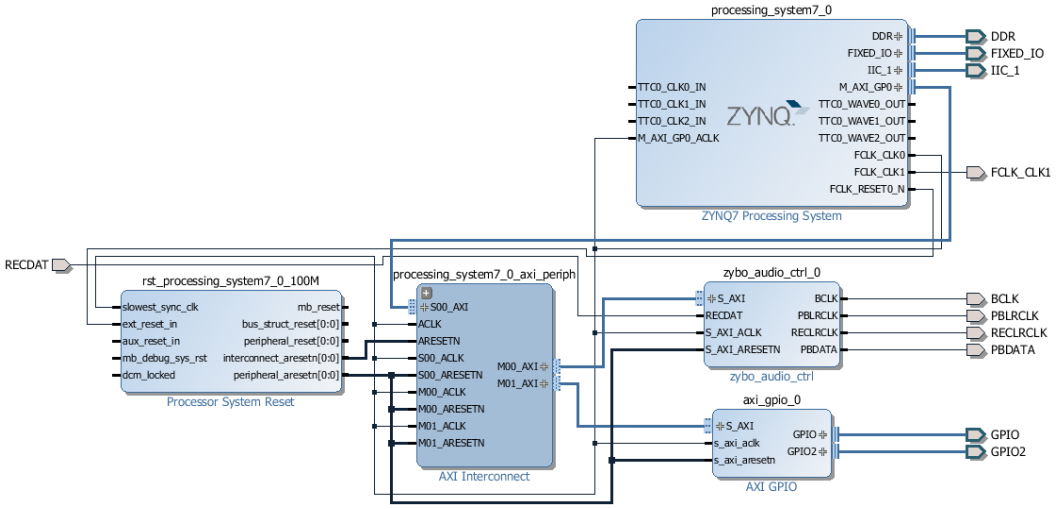



**Figure A-16. Block design after I2C based zybo_audio_ctrl core added and connections made**

## 14-6. Assign 64K address space to the zybo_audio_ctrl_0 instance

**14-6-1.** Select the *Address Editor* tab.

**14-6-2.** Expand **processing_system7_0 > Data.**

**14-6-3.** Click on the drop-down button of the size column corresponding to the *zybo_audio_ctrl_0* instance and change the size from **1G** to **64K.**

**14-6-4.** Select **Tools > Validate Design** and make sure that there are no errors.

**14-6-5.** Click **OK.**