Sri Durga Raju

Tutorial on how to implement and test an adder using PYNQ-Z2 board

Before we start with the tutorial on implementing an adder using a PYNQ board, let's briefly recap FPGA boards.

FPGA, which stands for **Field Programmable Gate Array**, is an integrated circuit that can be configured by the designer based on the requirements after manufacturing. In other words, it is reprogrammable; hence, it is called programmable.

They have two main elements – **Processing System (PS)** and **Programmable Logic (PL)**.

PS is a fixed-function processor subsystem that runs software (OS); PL is the reconfigurable set of logic blocks that provide custom hardware acceleration.

When working with FPGA, you can interact with it using either of the above-mentioned elements or a hybrid approach of using both. The PL way is best for real-time, high-speed hardware acceleration. The PS way is best for software-based tasks and embedded control.

In the case of PYNQ-Z2, the PS is the ARM Cortex-A9 Processors as the board is based on the Xilinx ZYNQ-7000 series SoC, and the PL is the FPGA fabric consisting of the logic cells, DSP slices, Block RAM, and configurable I/Os. Both the PS and PL interact with each other using the AXI Interconnect Interface, leveraging the power of Linux on ARM processors while accelerating algorithms on the PL.

In this tutorial, we will use the hybrid approach to implement and test the adder on the PYNQ-Z2 FPGA board.

STEP 1: Initial set up of the PYNQ Z2 board.

The following link provides the necessary steps required to set up the PYNQ Z2 board.

https://pynq.readthedocs.io/en/v2.5/getting_started/pynq_z2_setup.html

STEP 2: Add the PYNQ-Z2 board files to Vivado.

The following link provides the necessary PYNQ-Z2 board files required to work on a project using it in Vivado.

https://github.com/Xilinx/XilinxBoardStore/tree/master/boards/TUL/pynq-z2/A.0

Download all the files provided in the link, copy, and paste them into a new folder – 'pynq-z2' in the following location: **\Xilinx\Vivado\your_version\data\boards**.

STEP 3: Create a new Vivado project.

1. Open Vivado.
2. Click on **Create Project -> Next**.
3. Enter the **Project name – adder** and set a location, then click on Next.
4. Choose **RTL Project -> Next**.
5. Select **Do not specify sources at this time -> Next**.
6. In Part selection, select **PYNQ-Z2 board**.

STEP 4: Create an adder module in Verilog.

1. In Vivado, go to the **Flow Navigator -> Add Sources -> Add or Create design sources -> Create File**.
2. Select the **File type:** Verilog, provide the **File name:** adder.v , the **File location:** default option - **< Local to Project>** and click **Finish**.
3. Write the Verilog code for a simple 4-bit adder, which has two 4-bit input and a 5-bit output. **Fig 1,** provides the code.

```
1  `timescale 1ns / 1ps
2
3  module adder(
4  input [3:0] a,b,
5  output [4:0] sum
6      );
7
8      assign sum = a + b;
9  endmodule
10
```

Fig 1. Verilog file of the adder module

STEP 5: Create a testbench (Optional)

1. For simulation purposes, you can create a testbench to verify the adder. Go to the **Flow Navigator -> Add Sources -> Add or Create Simulation Sources -> Create File**.
2. Select the **File type:** Verilog, provide the **File name:** adder_tb.v, **File location:** default opion - **<Local to Project>** and click **Finish**.
3. Write the Verilog testbench code for a simple 4-bit adder. **Fig 2** provides the code.

```
1  `timescale 1ns / 1ps
2
3  module adder_tb;
4      reg [3:0] a, b;
5      wire [4:0] sum;
6
7      adder dut (
8          .a(a),
9          .b(b),
10         .sum(sum)
11     );
12
13     initial begin
14         a = 4'b0011; b = 4'b0101; #10; // 3 + 5 = 8
15         a = 4'b1111; b = 4'b0001; #10; // 15 + 1 = 16
16         a = 4'b1010; b = 4'b1010; #10; // 10 + 10 = 20
17         $stop;
18     end
19 endmodule
20
21
```

Fig 2. Verilog testbench file of the adder

STEP 6: Create a Block Design

1. Go to the **Flow Navigator -> IP Integrator** and click on **Create Block Design**, name the block design as **adder_design**, and click **OK**.
2. Add the following blocks:
    - **Adder module:** Right-click on the **Block Design Window** and choose **Add Module**.
    - **AXI GPIO Blocks (3 instances):** Click on **ADD IP** and search for **AXI GPIO**. Use one AXI GPIO block for each input/output port (a, b, and sum).
    - **ZYNQ7 Processing System:** Click on **ADD IP** and search for **ZYNQ7 Processing System**.
3. Connect components:
    - Configure the ZYNQ Processing System to enable AXI GPIO interfaces. Click on **Run Block Automation**. This will automatically connect the clocks and resets to the ZYNQ PS.
    - Configure the AXI GPIO blocks by setting the **GPIO Width** to 4 for a and b and 5 for the sum.
    - Connect the output of ‘a’ GPIO block to ‘a’ input port of the adder module. Similarly, connect the output of the ‘b’ GPIO block to the ‘b’ input port of the adder module.
    - Connect the input of the ‘sum’ GPIO block to the 'sum’ port of the adder module.
    - Connect each **S_AXI** port of an AXI GPIO block (e.g. for a) to the **M_AXI_GP0** (or similar) port on the ZYNQ PS.

STEP 7: Validate, generate the Block Design, and create HDL wrapper.

1. Once all connections are made, click **Tools -> Validate Design** to ensure there are no errors in the block design.
2. If validation is successful, save the block design.
3. Right-click ‘adder’ in your source tab and then select **Create HDL Wrapper -> Let Vivado manage wrapper and auto-update**. Click **OK**.

This setup creates a system where:

- The ARM processor can write to the ‘a’ and ‘b’ AXI GPIO blocks to control the inputs.
- The result (sum) can be read through the ‘sum’ AXI GPIO block.

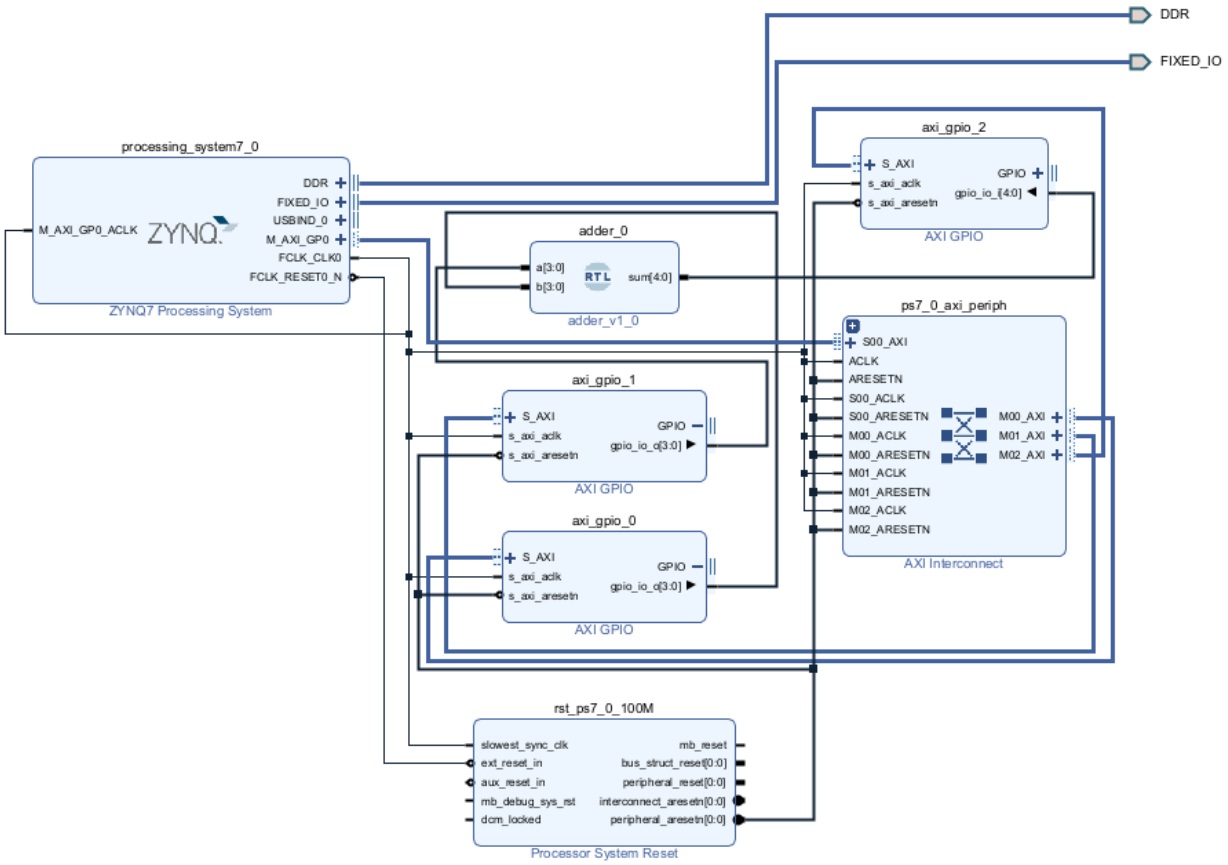

Fig.3 Block Design

STEP 8: Generate a Bitstream

1. Synthesize the design -> click on **Run Synthesis -> OK.**
2. Implement the design -> click on **Run Implementation -> OK**.
3. Generate the bitstream -> click on **Generate Bitstream -> OK**.

As of now, we are done with working on the board using the PL way – generating the overlay files by Vivado in the form of a bitstream file. Overlay files are the essential hardware libraries that will leverage the power of FPGA.

STEP 9: Testing the adder using PYNQ Jupyter Notebook

Now, we will start working on the board using the PS way by running a Python script to send data to the adder's input and read back the output using the AXI GPIO interface. Juypter Notebook is a server that runs on Linux. By using your PC, you can connect to the device using the Juypter Notebook server. In this way, you can run commands on the PYNQ processor remotely.

1. Connect the PYNQ-Z2 board to your PC via Micro-USB.
2. Use the following link to connect to the Juypter Notebook on the board. https://pynq.readthedocs.io/en/v2.5/getting_started.html#connecting-to-jupyter-notebook

3. Create a new folder by pressing **New -> Folder**. A folder called '**Untitled Folder**' will appear. Select the empty checkbox and click **Rename**. You can call this folder whatever you want, e.g. **'adder'**. After Renaming, Click on **'adder'**.
4. Export the design as an overlay to the Jupyter Notebook: This can be done by uploading the bitstream files of the adder to the same folder in the Jupyter Notebook where you will create the Python file. Remember that bitstreams require an associated **.tcl** or **.hwh** file that provides metadata about the hardware design (pins, interfaces, memory mappings) in addition to the **.bit** file. Make sure the **.hwh** file with the same base name as your bitstream file is in the same directory. For instance, if your bitstream is **adder.bit**, you should also have **adder.hwh** in the same folder. You can find **.bit** and **.tcl** files in the location where you had created your Vivado project: **\project_name\project_name.runs** and the **.hwh** file: **\project_name\project_name.gen\sources_1\bd\project_design.** Shows the bitstream files stored in the same folder as the python file.

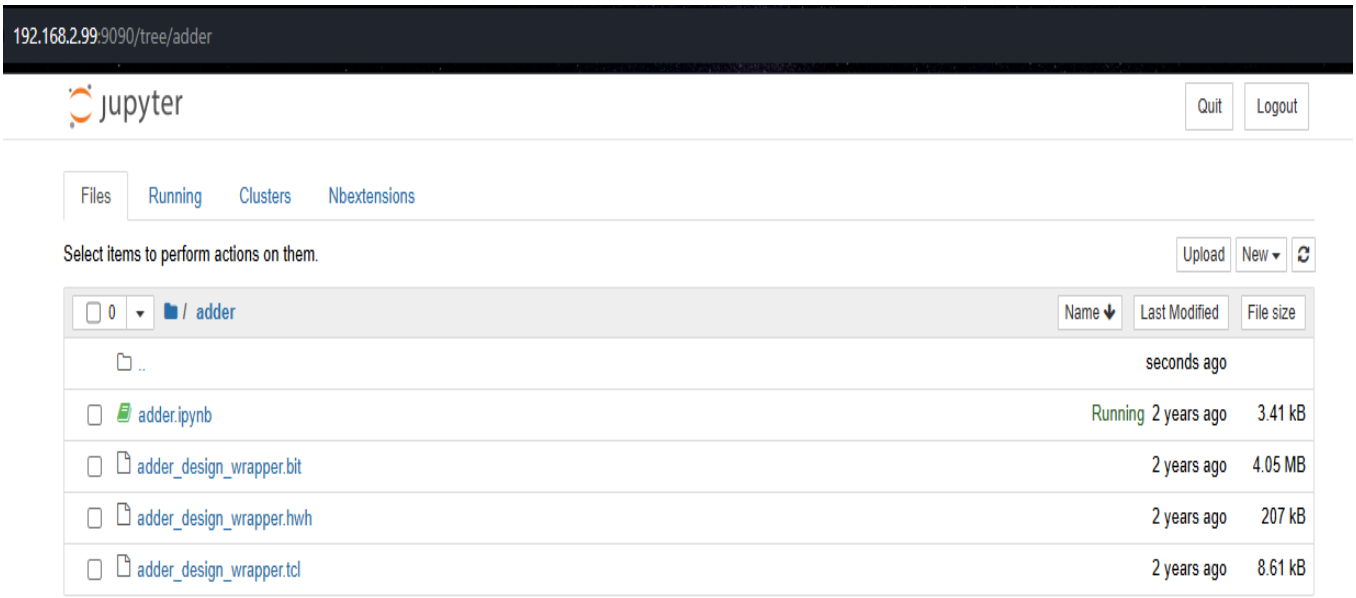

Fig 4. Uploaded bitstream files in the Jupyter Notebook

5. Now, we will create the Python file to test the adder. Click on **New -> Python**. Rename the notebook by clicking the **'Untitled'** label and call it **'adder'**. After Renaming, Click on **'adder'**.
6. You can write a part of the Python code and just execute that code. Click on **Insert** on the menu and choose **Insert Cell Above or Below**, depending upon your need. You can execute the code on each cell by pressing **Shift + Enter.**
7. **Fig. 5** provides the Python code to test the 4-bit adder design. It can be drawn from the figure that when providing different inputs, the desired outputs are displayed. Hence, we can conclude that we were able to test a 4-bit adder using the PYNQ-Z2 board.

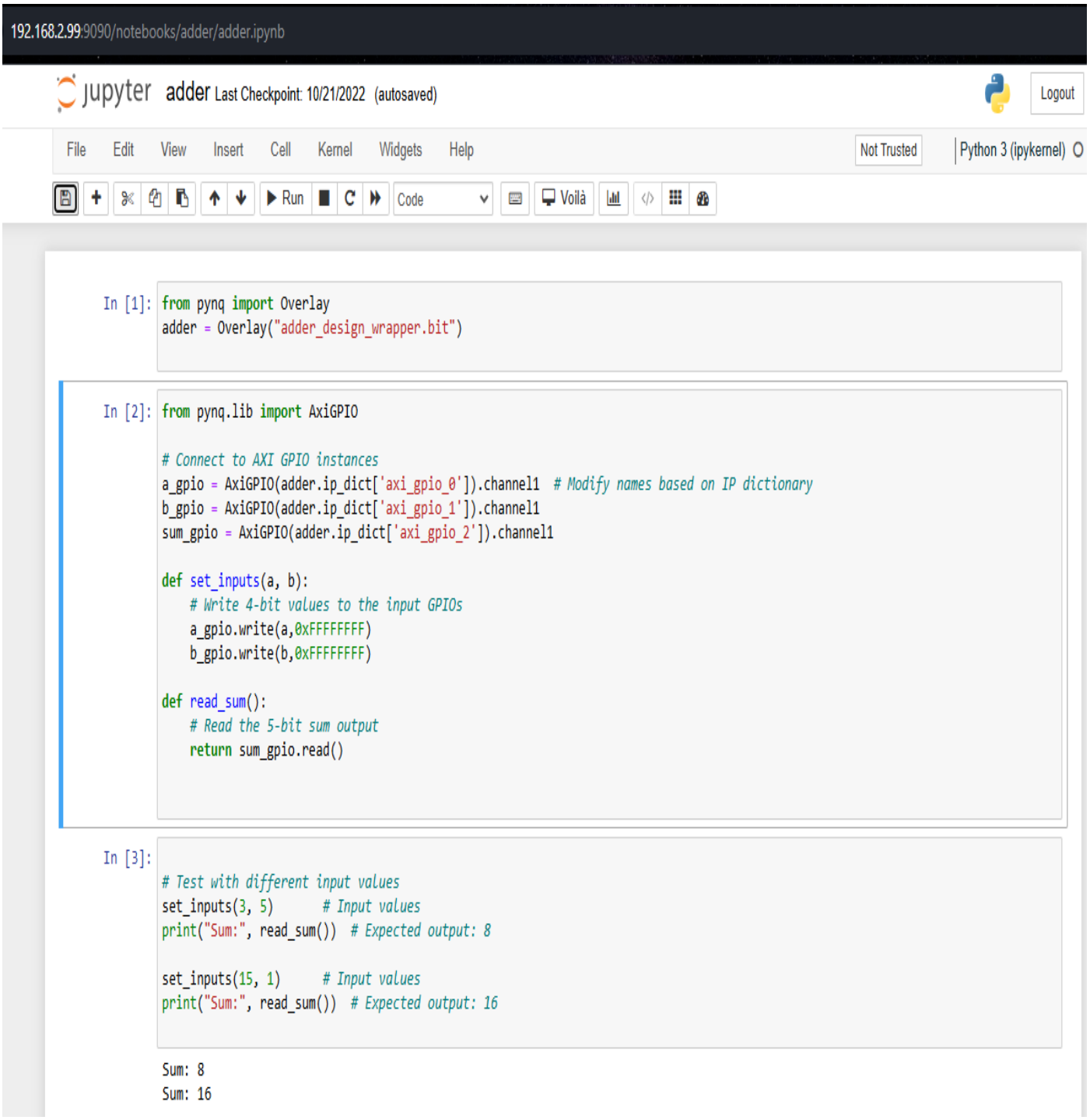

Fig. 5 Adder Python code

You can also try to project the outcome of the adder by utilizing 7-segment displays. This can be done by connecting the 7-segment displays to the Pmod ports of the PYNQ-Z2 board. PYNQ-Z2 has two PMOD ports – A and B available for general-purpose I/O. The procedure to implement this is similar to how we designed and tested the adder using the PYNQ-Z2 board with a few modifications.

The following provides the necessary steps required,

STEP 1: Create a new Vivado project – adder_7seg_display

STEP 2: Create the required Verilog modules.

1. We will be utilizing two 7-segment displays to project the outcome of the adder; one will display the ten's value of the sum, and the other will display the one's value of the sum. Therefore, we need to create a binary to BCD converter Verilog module. **Fig. 6** Provides the code.

```
`timescale 1ns / 1ps

module binary_to_bcd (
    input [4:0] binary,  // 5-bit binary input
    output reg [3:0] tens, // Tens digit (BCD)
    output reg [3:0] ones  // Ones digit (BCD)
);
    always @(*) begin
        tens = binary / 10;  // Integer division for tens place
        ones = binary % 10;  // Remainder for ones place
    end
endmodule
```

Fig. 6 Verilog code of the binary to BCD converter

2. To connect to the 7-segment display, a seven-segment decoder Verilog module is also required. **Fig. 7** Provides the code.

```
module seven_segment_decoder (
    input [3:0] digit,   // 4-bit BCD digit (0-9)
    output reg [6:0] seg // 7-segment display outputs: a, b, c, d, e, f, g
);
    always @(*) begin
        case (digit)
            4'd0: seg = 7'b0111111;  // Display 0
            4'd1: seg = 7'b0000110;  // Display 1
            4'd2: seg = 7'b1011011;  // Display 2
            4'd3: seg = 7'b1001111;  // Display 3
            4'd4: seg = 7'b1100110;  // Display 4
            4'd5: seg = 7'b1101101;  // Display 5
            4'd6: seg = 7'b1111101;  // Display 6
            4'd7: seg = 7'b0000111;  // Display 7
            4'd8: seg = 7'b1111111;  // Display 8
            4'd9: seg = 7'b1101111;  // Display 9
            default: seg = 7'b0000000; // Blank for invalid input
        endcase
    end
endmodule
```

Fig .7 Verilog code of the seven-segment decoder

3. Add the previously created adder module to the Design sources.

STEP 3: Create a Block Design

1. Go to the **Flow Navigator -> IP Integrator** and click on **Create Block Design**, name the block design as **adder_7seg_design**, and click **OK**.
2. Add the following blocks:
    - **Adder module:** Right-click on the **Block Design Window** and choose **Add Module**.
    - **Binary-to-BCD module:** Right-click on the **Block Design Window** and choose **Add Module.**
    - **Seven-segment decoder module (2 instances):** Right-click on the **Block Design Window** and choose **Add Module.** Use one seven-segment decoder block for each one's and ten's digits.
    - **AXI GPIO Blocks (2 instances):** Click on **ADD IP** and search for **AXI GPIO**. Use one AXI GPIO block for each input port (a, b).
    - **ZYNQ7 Processing System:** Click on **ADD IP** and search for **ZYNQ7 Processing System**.
3. Connect components:
    - Configure the ZYNQ Processing System to enable AXI GPIO interfaces. Click on **Run Block Automation**. This will automatically connect the clocks and resets to the ZYNQ PS.
    - Configure the AXI GPIO blocks by setting the **GPIO Width** to 4 for a and b.
    - Connect each **S_AXI** port of an AXI GPIO block (e.g. for a) to the **M_AXI_GP0** (or similar) port on the ZYNQ PS.
    - **AXI GPIO**: Connect the AXI GPIO blocks to the ZYNQ Processing System through the AXI interface. Use the GPIO ports to drive the a and b inputs of the adder.
    - **Adder Module**: Connect the 'a' and 'b' inputs of the adder to the GPIO outputs. Connect the 'sum' output to the input of the binary-to-BCD converter.
    - **Seven-Segment Decoders**: Connect the ten's and one's outputs from the binary-to-BCD converter to the two decoder blocks. Export the 'seg_tens' and 'seg_ones' outputs to PMOD pins. To do so, right-click on the outputs of the binary-to-BCD converter and choose **Make External**.

STEP 4: Validate, generate the block design, and create HDL wrapper.

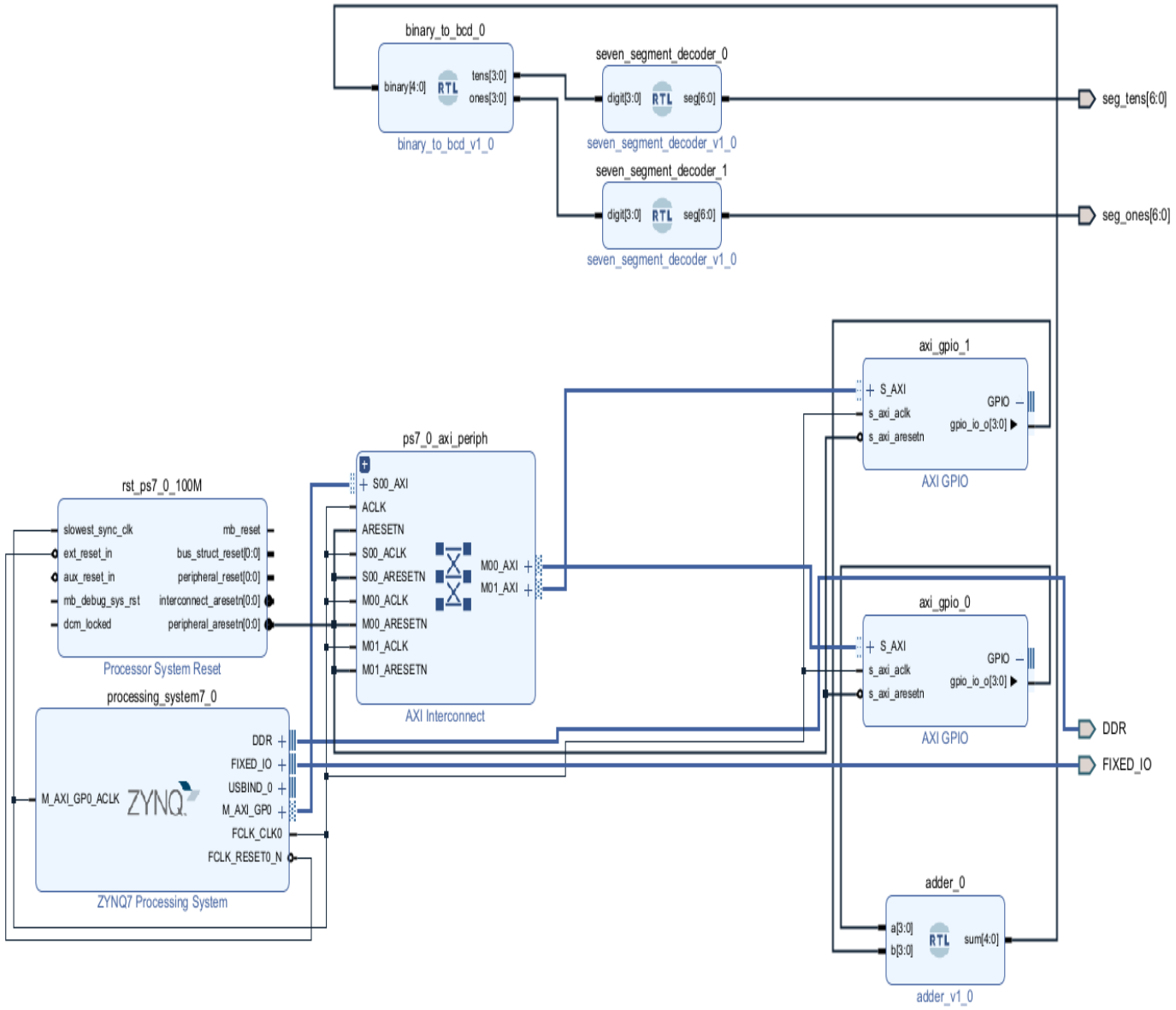

Fig.8 Block Design

STEP 5: Create the constraint file.

1. A constraint will have the port name from your design and the FPGA pin name.
2. Each 7-segment display requires seven pins, which are provided by the output of the seven-segment decoder. Assign these signals to PMOD pins using the constraint file (.xdc file).
3. Go to the **Flow Navigator -> Add Sources -> Add or create constraints -> Create File**. Provide **File name:** adder_top; **File type** and **File location** are the default options – **XDC** and **<Local to Project>.**
4. You can refer to the following link for the reference XDC file of the PYNQ-Z2 board to get the pin configuration of the Pmod ports.

https://github.com/Xilinx/PYNQ/blob/master/boards/Pynq-Z2/base/vivado/constraints/base.xdc

```
# PMOD A for tens digit
set_property PACKAGE_PIN Y18 [get_ports {seg_tens[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_tens[0]}]
set_property PACKAGE_PIN Y19 [get_ports {seg_tens[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_tens[1]}]
set_property PACKAGE_PIN Y16 [get_ports {seg_tens[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_tens[2]}]
set_property PACKAGE_PIN Y17 [get_ports {seg_tens[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_tens[3]}]
set_property PACKAGE_PIN U18 [get_ports {seg_tens[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_tens[4]}]
set_property PACKAGE_PIN U19 [get_ports {seg_tens[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_tens[5]}]
set_property PACKAGE_PIN W18 [get_ports {seg_tens[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_tens[6]}]

# PMOD B for ones digit
set_property PACKAGE_PIN W14 [get_ports {seg_ones[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_ones[0]}]
set_property PACKAGE_PIN Y14 [get_ports {seg_ones[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_ones[1]}]
set_property PACKAGE_PIN T11 [get_ports {seg_ones[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_ones[2]}]
set_property PACKAGE_PIN T10 [get_ports {seg_ones[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_ones[3]}]
set_property PACKAGE_PIN V16 [get_ports {seg_ones[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_ones[4]}]
set_property PACKAGE_PIN W16 [get_ports {seg_ones[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_ones[5]}]
set_property PACKAGE_PIN V12 [get_ports {seg_ones[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg_ones[6]}]

```

Fig. 9 Constraint file

STEP 6: Run Synthesis, Run Implementation, and Generate Bitstream.

STEP 7: Testing using PYNQ Jupyter Notebook

1. Connect the PYNQ-Z2 board to your PC via Micro-USB.
2. Connect the Pmod ports of the PYNQ-Z2 board to the inputs of the 7-segment display.
3. Create a new folder – **adder_7seg**. Export the design as an overlay to the adder_7seg folder of the Jupyter Notebook.
4. Now, create the Python file to test the adder. **Fig. 10** provides the Python code to test the 4-bit adder design.

192.168.2.99:9090/notebooks/adder_7seg/adder_7seg.ipynb

jupyter adder_7seg Last Checkpoint: 10/21/2022 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)

Run Code Voilà

In [1]:
```
from pynq import Overlay
from pynq.lib import AxiGPIO

# Load the bitstream
adder7seg = Overlay("adder_7seg_design_wrapper.bit")

# Connect to AXI GPIO instances
a_gpio = AxiGPIO(adder7seg.ip_dict['axi_gpio_0']).channel1  # Modify names based on IP dictionary
b_gpio = AxiGPIO(adder7seg.ip_dict['axi_gpio_1']).channel1

def set_inputs(a, b):
    # Write 4-bit values to the input GPIOs
    a_gpio.write(a,0xFFFFFFFF)
    b_gpio.write(b,0xFFFFFFFF)
```

In [2]:
```
# Test with different input values
set_inputs(15, 5)       # Input values

print(f"Value of a: {bin(a_gpio.read())}")
print(f"Value of b: {bin(b_gpio.read())}")
```

```
Value of a: 0b1111
Value of b: 0b101
```

Fig. 10 Python code

Outcome:

On running the Python code, the desired output of the 4-bit adder will be displayed in the seven-segment display. From **Fig. 10** and **Fig. 11**, it can be inferred that on providing inputs **'a': 15** and **'b': 5**, the desired **output: 20** is displayed; **2** - the ten's value of the sum is displayed on the **top** 7-segment display and **0** - the one's value of the sum is displayed on the **bottom** 7-segment display. Hence, we can conclude that we were able to test a 4-bit adder using the PYNQ-Z2 board and project the outcome to 7-segment displays.

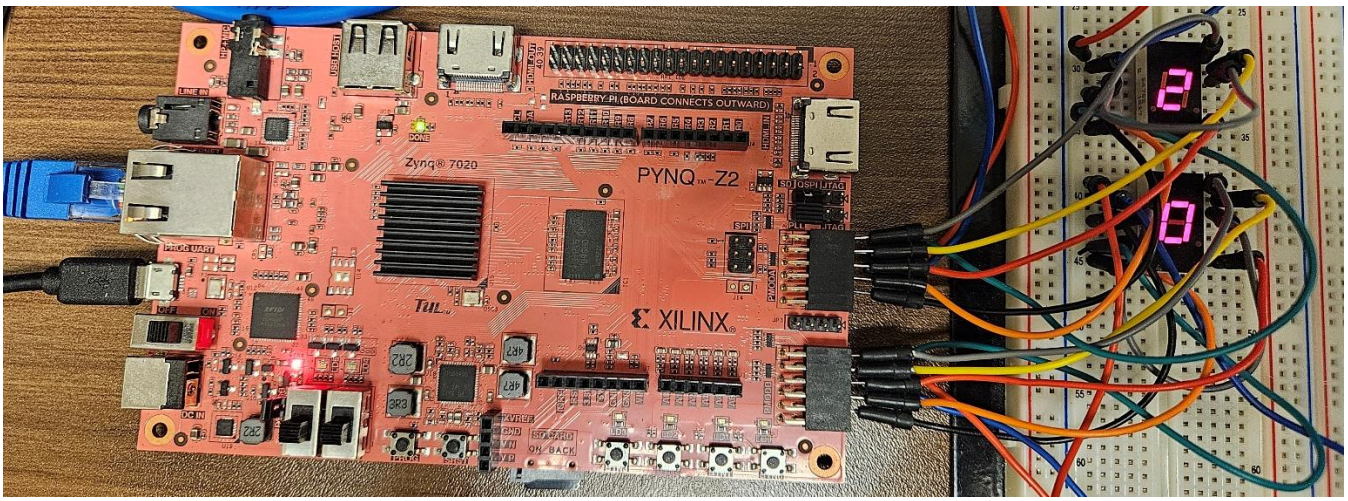

Fig. 11 Output of the 4-bit adder