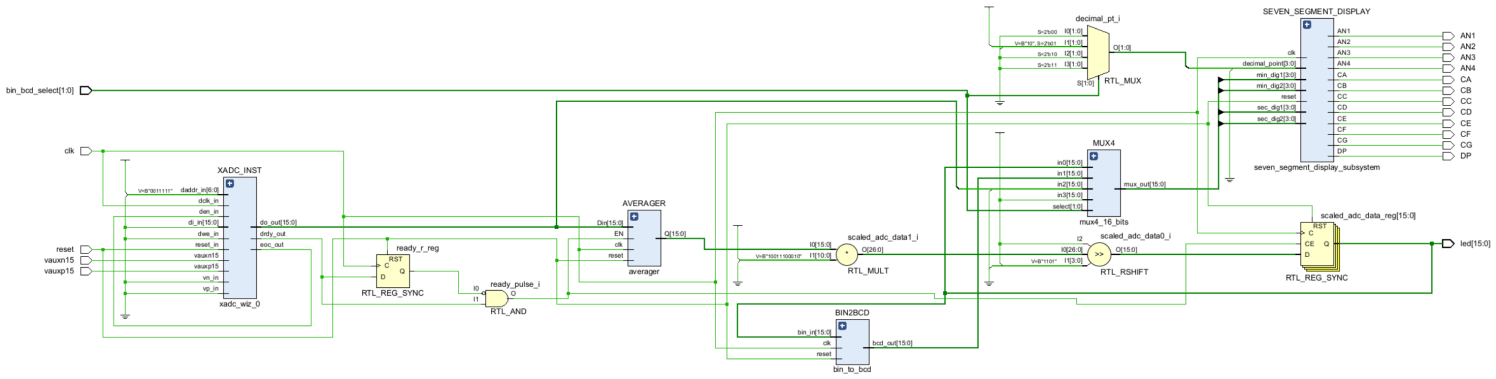
Lab 5 – XADC and Modularity

Introduction

In Lab 5, we will use the Xilinx Vivado IP Catalog to enable the Analog to Digital Converter (ADC) that is built into the Artix 7 FPGA. We will read an analog voltage between 0 V and 1 V, and display the 16-bit ADC value in hexadecimal. We will scale the 16-bit hexadecimal value and display it as an accurate voltage reading, in millivolts. You will use a parameterizable averager, which creates a moving average of the ADC samples, to smooth the reading. In fact, this averager actually increases the effective resolution of the 12-bit ADC, up to 16-bits (or more).

Since the 7-segment displays have 4-digits, we will need to place the decimal point accordingly so that a reading of, say, 999.9 mV is presented correctly. This means modifying the 7-segment display subsystem to provide control of the decimal point. This has been completed and is provided to you.

You will be provided the instructor's solution and will recreate it using the Xilinx IP Catalogue. You will then package the XADC and supporting logic into an ADC subsystem, to demonstrate modular design. You will calibrate the ADC, to give more accurate readings and use a DMM as the reference



Recall the guidance you were given in previous lab projects.

- Be sure to use your top level module only to instantiate and connect lower level modules. The example top level module code provided by the instructor ***violates*** this guidance. Please incorporate that top level logic into a reasonable module, or make it its own module.
- Consider when it's appropriate to combine related modules into a "mini top level," to create a higher level of abstraction and simplify your design.
- We will also follow synchronous design principles, so whenever we have a `posedge` in an `always` block, it shall be `posedge clk`, and not `posedge someOtherSignal`. Only one clock for the entire system. Use the concept of an enable signal, to avoid driving other signals as clock signals.

Caution: The synthesis times will be very long, so be careful on how you manage your time in this project.

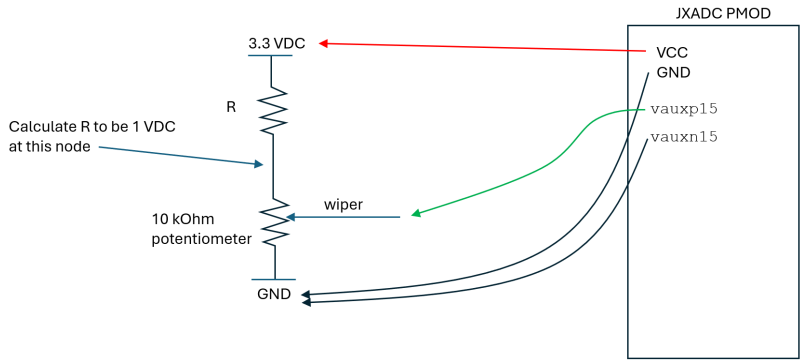
You will submit your Design Record to the D2L Dropbox prior to your TA evaluation.

Procedure

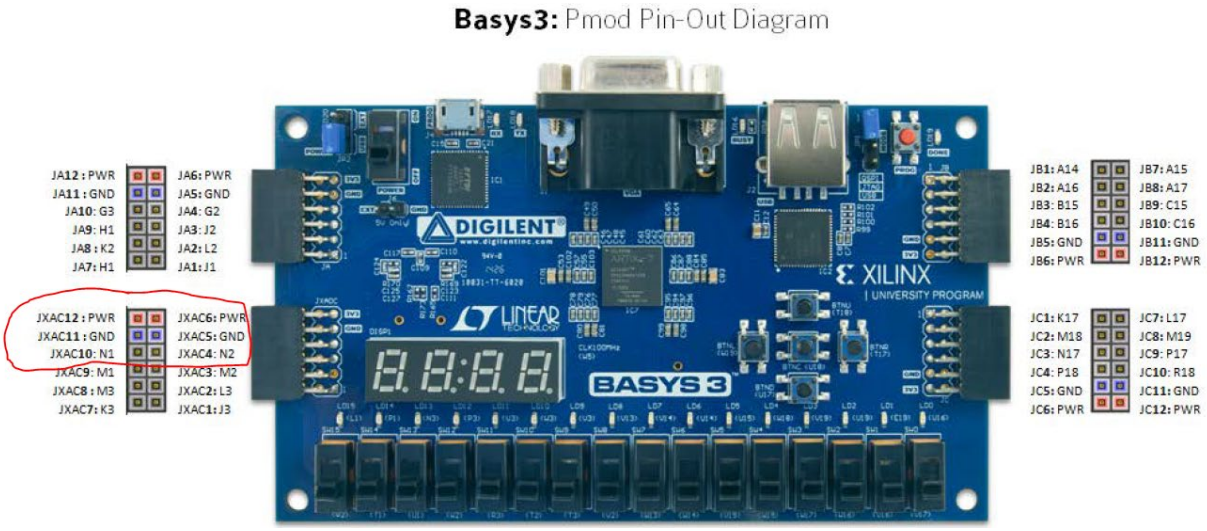
1. Prior to the lab, watch the all the lecture videos for this week and the previous week.
2. Create a Lab 5 project with the provided files from D2L. After the project has been setup and the .SV and .XDC files brought into the project, you will need to add the XADC block.
3. Add the XADC IP block from the IP catalogue, to the design. Detailed instructions are given in the Appendix of this document, please follow them carefully. After you complete generating the XADC block, you will want run Synthesis and Implementation, but that will take a long time. Therefore, it is recommended that you do a quick RTL Schematic, to at least check whether your design looks reasonable, i.e. matches the instructor's RTL Schematic on the previous page. If everything looks fine, generate the Bitstream and download to the Basys3. This will run Synthesis and Implementation, prior to generating the Bitstream.

The rest of this page is deliberately left blank.

4. Unplug the Basys3 to be safe, before making the following connections. You will generate a variable voltage with a 10 kΩ potentiometer and the 3.3 VDC available from the Basys3.
    - First, calculate a voltage divider, with a resister R connected to 3.3 VDC, and the 10 kΩ potentiometer connected to ground. Ensure by an appropriate selection of R, that the voltage at the node between the resistor R and the 10 kΩ potentiometer, is 1 V (see image below). This will ensure that the 10 kΩ potentiometer's wiper will be between 0 V and 1 V, as it turns. Connect resistor R and the 10 kΩ potentiometer on a breadboard. Note that the resistor R serves two practical purposes: it drops the potentiometer voltage to 1 V, and it also limits the current that can flow into the XADC, in case of overvoltage or undervoltage conditions. The Xilinx documentation states two numbers, 10 mA or 1 mA (it's ambiguous), as the maximum current that can safely flow into the XADC due to under/overvoltage conditions, so we should be safe with the R value you calculate, for either case.



- Use the JXADC PMOD header on the Basys3 (see red circle below), to connect 3.3 VDC and ground to the resistor/potentiometer circuit.
- Connect `vauxn15` (labelled JXAC10:N1) to ground (also see the second image below). This is the ground of the ADC.
- Connect `vauxp15` (labelled JXAC4:N2) to the 10 kΩ potentiometer wiper (also see the second image below). This is the positive of the ADC.



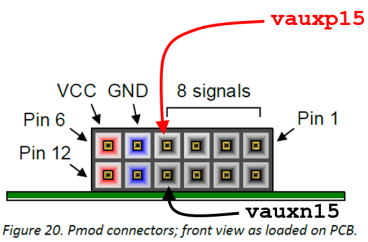


Figure 20. Pmod connectors; front view as loaded on PCB.

5. After making the connections to the breadboard, reconnect your Basys3 to USB and download the Bitstream again. You should see the decimal and hexadecimal values and the LEDs changing in response to the 10 kΩ potentiometer knob being turned. At one end, it should go to full scale, or close to it. There are 4 readings, depending on the positions of the slide switches SW0 and SW1 (the rightmost ones). The readings include:
    - The raw 12-bit value from the XADC, presented in hexadecimal. This will be fluctuating rapidly. The full scale reading will be 0FFFh.
    - The averaged XADC value, which is generated from the 4096-sample moving average. This will be fairly stable, and an extra 4-bits will be generated due to the averaging, so it will be a 16-bit hexadecimal value. The full scale reading will be FFFFh. If you wish to understand how the extra ADC resolution bits are obtained, please read the paper in D2L, AN118 Improving ADC Resolution by Oversampling and Averaging.
    - The scaled hexadecimal value of the moving average (previous bullet), to represent the actual voltage in hexadecimal. The full scale reading will be 270Fh. Read the comments starting on line 88 of the top level, to understand how the scaling is calculated.
    - The BCD value of the scaled hexadecimal value, to give a reading in millivolts. The decimal point is in the correct place, so the full scale reading will be 999.9 (mV). Note, the 7-segment display subsystem has been modified to provide access to the decimal points.
6. Measure the wiper voltage with a DMM and compare to the values you are seeing on the Basys3, at full scale, at zero and selected values in between.
    - **DO: copy and paste a snip of your top level RTL Schematic, into your Design Record.**
    - **DO: In the Design Record, briefly comment on the values you see on the 7-segment displays and LEDs, in comparison to the wiper voltage. Record several samples of measured voltage with the DMM, and what was reported by the Basys3. Briefly discuss the possible reasons for the discrepancies.**
7. Push and release reset. Observe that the reading on the Basys3 counts up from zero until it reaches a stable value.
    - **DO: IN the Design Record, briefly explain the counting up behavior and what causes it.**
8. Create an ADC subsystem module. Incorporate the XADC, the averager, and the scaling calculations into the module. We want our top level module to be clean and just have instantiations of lower level modules and the connections between them. Since Synthesis takes a very long time, do a quick RTL Schematic to ensure that at least your connections and basic logic look correct. Download the Bitstream to the Basys3 and verify that your modified system works as before (recall this is regression testing).
    - **DO: copy and paste a snip of your top level RTL Schematic, into your Design Record.**
    - **DO: take a snip of the Timing window and paste it into your Design Record. Show the calculations for the maximum clock frequency for your design, based on the WNS.**

9. Now, we want to improve the accuracy of our Basys3. Study the calculations that begin on line 88 below:

```
82      always_ff @(posedge clk) begin
83          if (reset) begin
84              scaled_adc_data <= 0;
85              scaled_adc_data_temp <= 0;
86          end
87          else if (ready_pulse) begin
88              // Calculation: This scales FFFFh to 270Fh (i.e. 9999d)
89              //    mVolts = ave_data/(2^16 - 1) * 9999 = ave_data * 0.152575
90              //    mVolts ~ ave_data * 1250/2^13 = (ave_data) * 1250 >> 13
91              // NOTE: The 7-seg display will display in millivolts,
92              //       i.e. 9999 is 0.9999 V or 999.9 mV
93              //       place the decimal point in the correct place!
94              scaled_adc_data <= (ave_data*1250) >> 13; // was scaled_adc_data_temp
95              //scaled_adc_data <= scaled_adc_data_temp; // additional register facilitates pipelining
96          end                                          // for higher clock frequencies
97      end
```

The basic idea is to convert a reading of FFFFh to the decimal number 9999d (or 270Fh), where 9999d represents 999.9 mV. The challenge is scaling properly, because a simple calculation of taking the reading of the XADC (call it ADC_value), and converting it to the voltage it represents (call it voltage_value) is

ADC_value/FFFFh*9999d = voltage_value

Which results in a multiplication by a fraction, i.e. voltage_value = ADC_value * 0.152575.

This fractional multiplication will not synthesize, so we need to find another way. Multiplication by integers will work, then we can scale the result by powers of 2 (i.e. right shifting).

For example: if our fraction was 0.37721 (using a different fraction), we could represent it as approximately $0.37721 * \frac{2^9}{2^9} \approx \frac{193}{2^9}$

Then, if our calculation was voltage_value = ADC_value * 0.37721, we could replace that difficult calculation with $voltage\_value = ADC\_value \times 0.37721 = ADC\_value \times \frac{2^9}{2^9} \approx ADC\_value \times \frac{193}{2^9}$

Which in SystemVerilog in a synchronous `always_ff` simply becomes:

```
voltage_value <= (ADC_value * 193) >> 9;
```

Note that the term $\frac{1}{2^9}$ became a right shift by 9 (i.e >> 9). The above calculation will synthesize fine and not result in timing errors (i.e. with the WNS). 5.

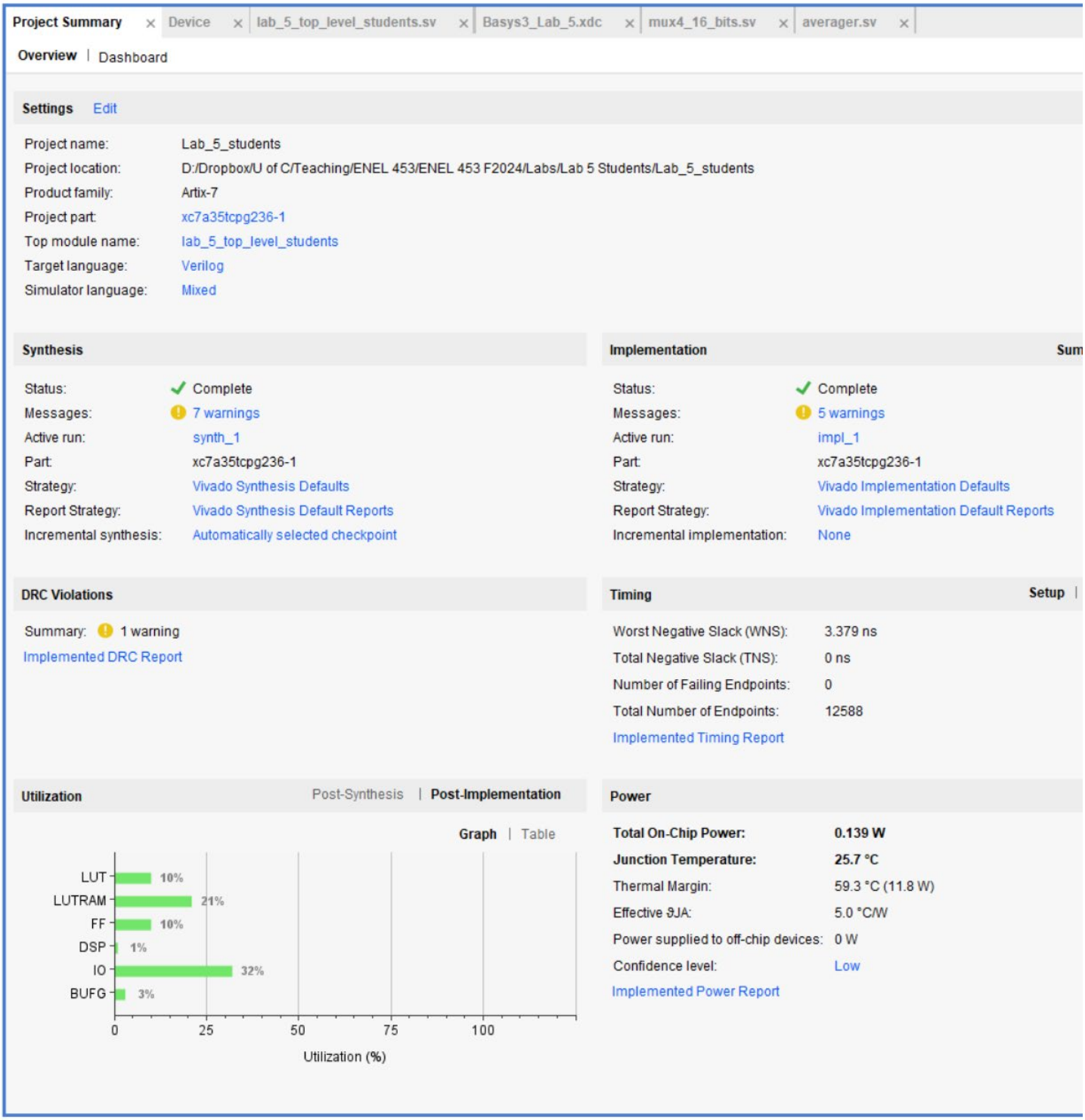
So, to improve the accuracy of the Basys3, you'll need to calibrate the scaling factor (e.g. 1250/2^13) to a more accurate scaling factor, based on a measurement with the DMM. The general procedure will be something like this:

- Set the wiper voltage to close to 1 V, so that you get a close to full scale reading on the Basys3, say FFF1h (for the averaged XADC value).
- Then read the voltage precisely with the DMM, and use both the values (e.g. FFF1h and 0.992 V) and create a scaling factor using the above procedure, and come up with your own scaling integer and shift value, following the example, but for the ratio you calculated.

10. Incorporate the new scaling factor into your design and test on the Basys3, with the DMM.
DO: when completed, record your calculations for the scaling factor, state your integer and shift value, and comment on the results from testing, in your Design Record.
DO: take a snip of the Timing window and paste it into your Design Record. Show the calculations for the maximum clock frequency for your design, based on the WNS.

11. The averager is parameterizable. Explore the values of N and find what is the largest value of N that can still fit into the Basys3 (while the averager is still part of the overall design). The Project Summary Overview gives a nice summary of key aspects of the design, including Utilization and WNS (see image below). This activity can be done in parallel by team members, to save time.

    DO: copy and paste the Project Summary Overview of the largest design that will fit in the Basy3 (based on N), in your Design Record.
    DO: show your working design to your TA.



Optional: Write your own averager to replace the given averager. The existing averager creates a binary adder tree to efficiently calculate the sum of all 2^N values. Note that the existing averager was initially written by ENEL 453 students in Fall 2019, and they did a superb job. It has since been translated from VHDL to SystemVerilog. For your averager, use an ***accumulator*** (i.e. only one adder to accumulate the additions). In this case, after 2^N samples have been added, and a new value comes in, subtract out the oldest value. Your accumulator-based averager should be a functionally equivalent, drop-in replacement for the provided averager. Note that an accumulator-based averager will be much more hardware efficient than the averager we used in this lab project. Feel free to use AI to help you for this optional activity.

Deliverables

By the end of the lab period **or** the beginning of the next lab period, demonstrate to the TA:

1. Your Basys3 board running with the latest iteration of the project, be prepared to explain the design and any part of the Vivado design and programming flow.
2. Your Design Record document. Ensure that the Design Record is uploaded to the D2L Dropbox, before seeing the TA.

Your TA may ask any team members at random to answer any questions. Work together to ensure that all team members fully understand the lab project and its deliverables. You may also be asked to demonstrate your ability to proceed through the entire design flow, including:

1. Create and start a new project.
2. Synthesize and download your design to the Basys3.
3. Simulate your design and setup the waveforms.

Rubric

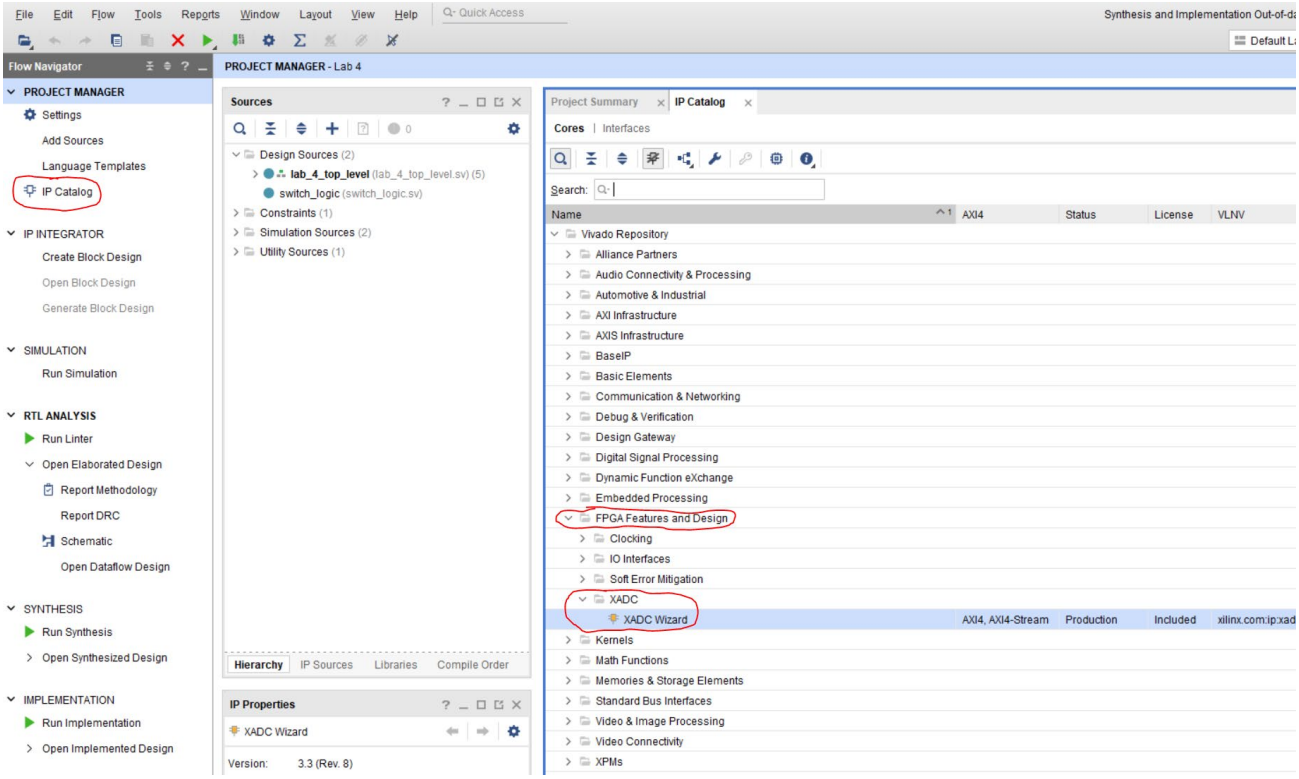
As the term progresses, the expectations will increase as your skills develop. All team members are required to participate in the entire lab period and to present their project to the TA. Missing team members will not receive a mark for the project. The three strikes policy outlined in the Course Outline, will be in effect for all labs.

Points	Criteria
4	Fully complete and high quality, questions answered well.
3	Fully complete and high quality, some questions not answered well or had to have other team members answer.
2	Mostly complete or answers are weakly answered by team members
0	Below acceptable for credit.

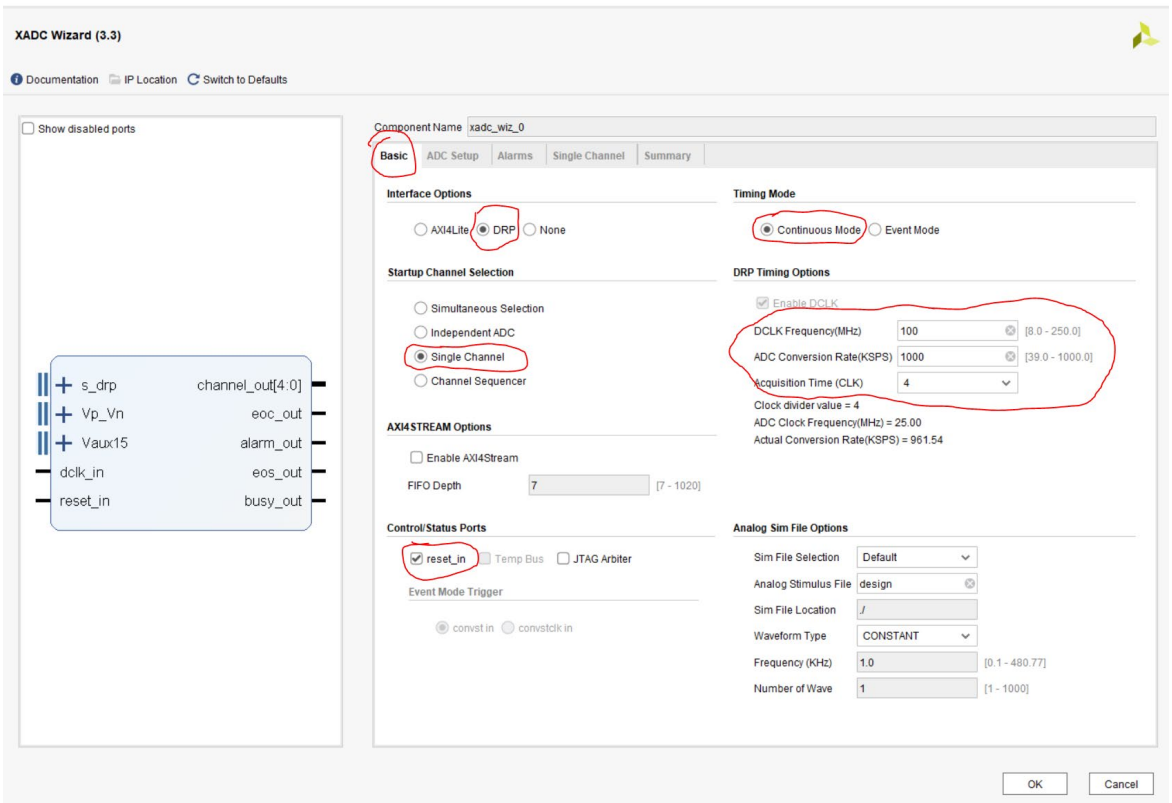
There is no 1 point given. Fractional points, e.g. 2.5, are not given.

Appendix – XADC configuration steps/screenshots

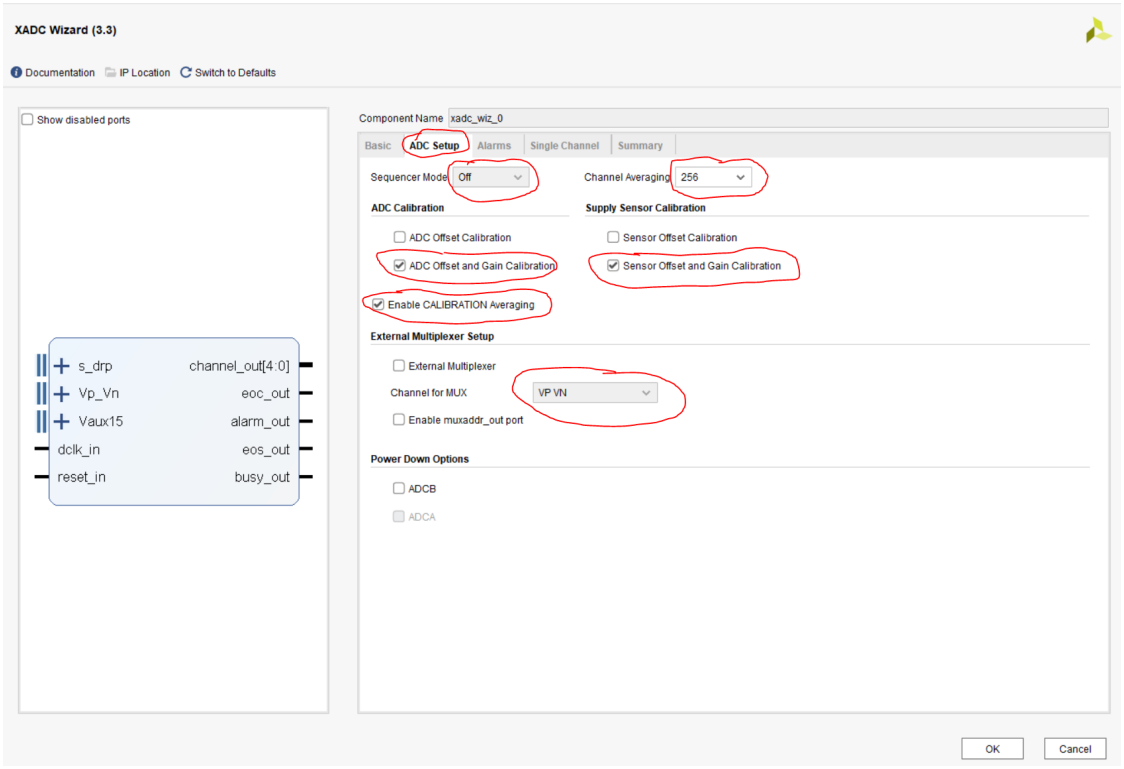
Project Manager > IP Catalog > FPGA Features and Design > XADC > XADC Wizard (double-click)



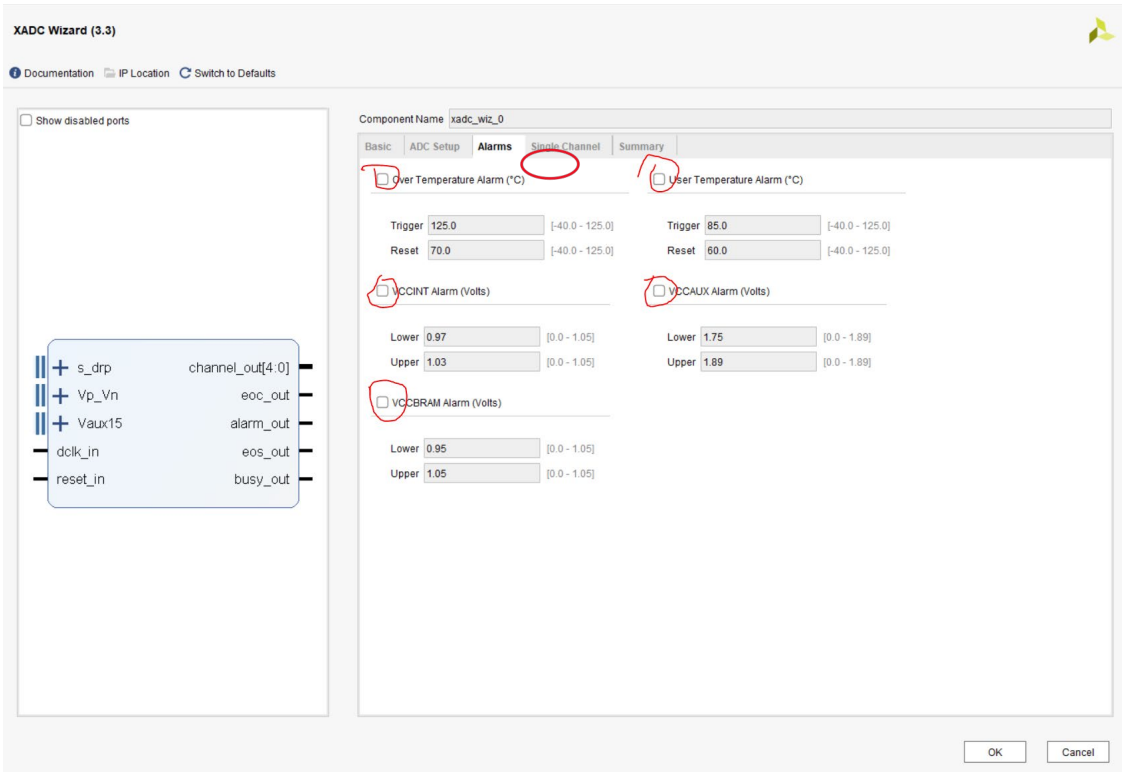
The XADC Wizard will pop up. Under the **Basic** tab, make sure the following are selected, then press OK. Continuous Mode means the ADC keeps automatically sampling, whereas Event Mode has sampling under user control for each sample. Notice that the ADC sampling rate has been set to 1000 KSPS (kilo Samples per second), or 1 MSPS.



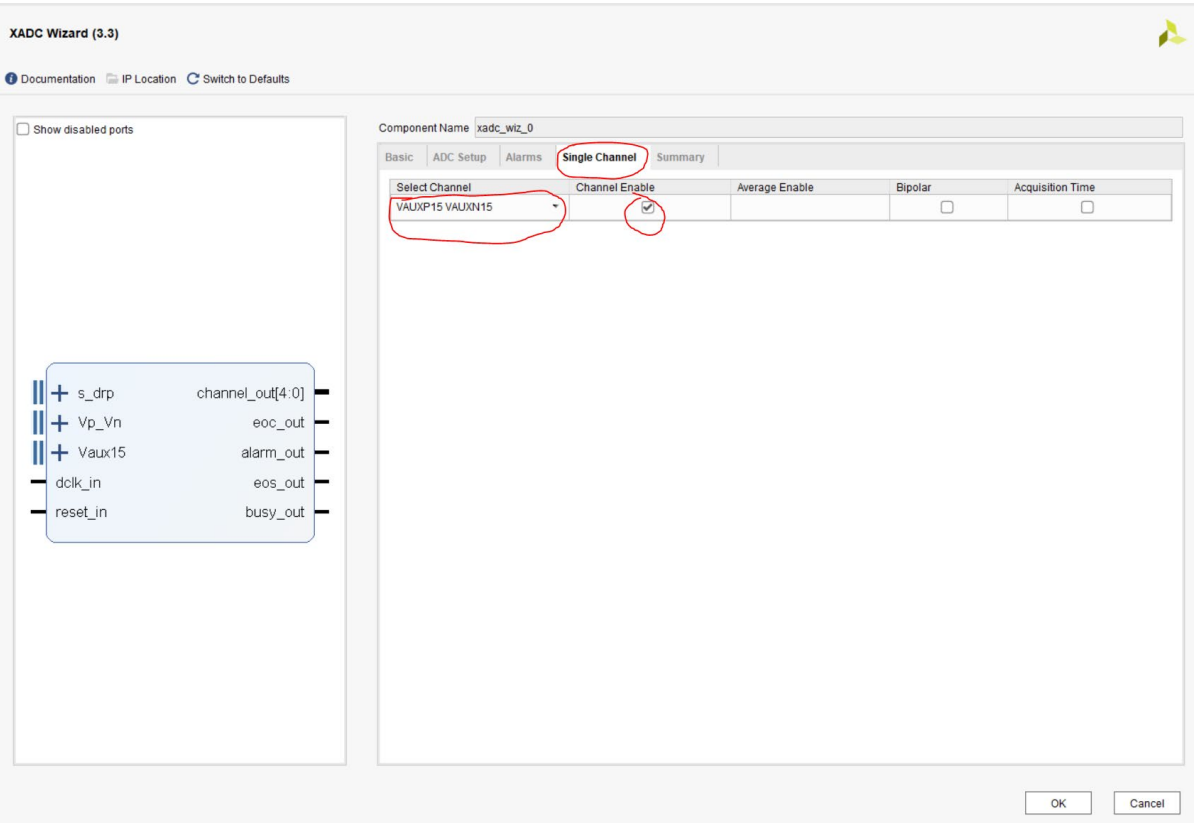
Under the **ADC Setup** tab, make sure the following are selected and press OK. Channel Averaging is selected at 256. This means that 256 samples will be averaged to produce an ADC output sample. This will be a "smoother" and less noisy sample, but the effective sample rate will be 1 MSPS/256 = 3.906 kSPS. See how averaging can possibly increase the effective number of bits in the ADC conversion, in the paper in the D2L Lab 5 folder. From the paper, 256 samples is 4^4 = 256, and this can add 4* bits of resolution to our 12-bit ADC, making it effectively a 16-bit ADC. ***The exponent of 4^N is the number of N additional ADC bits of resolution, provided by averaging.** It may seem counter-intuitive, but it is the "noise" in the signal that contributes to the improvement in ADC resolution due to averaging.



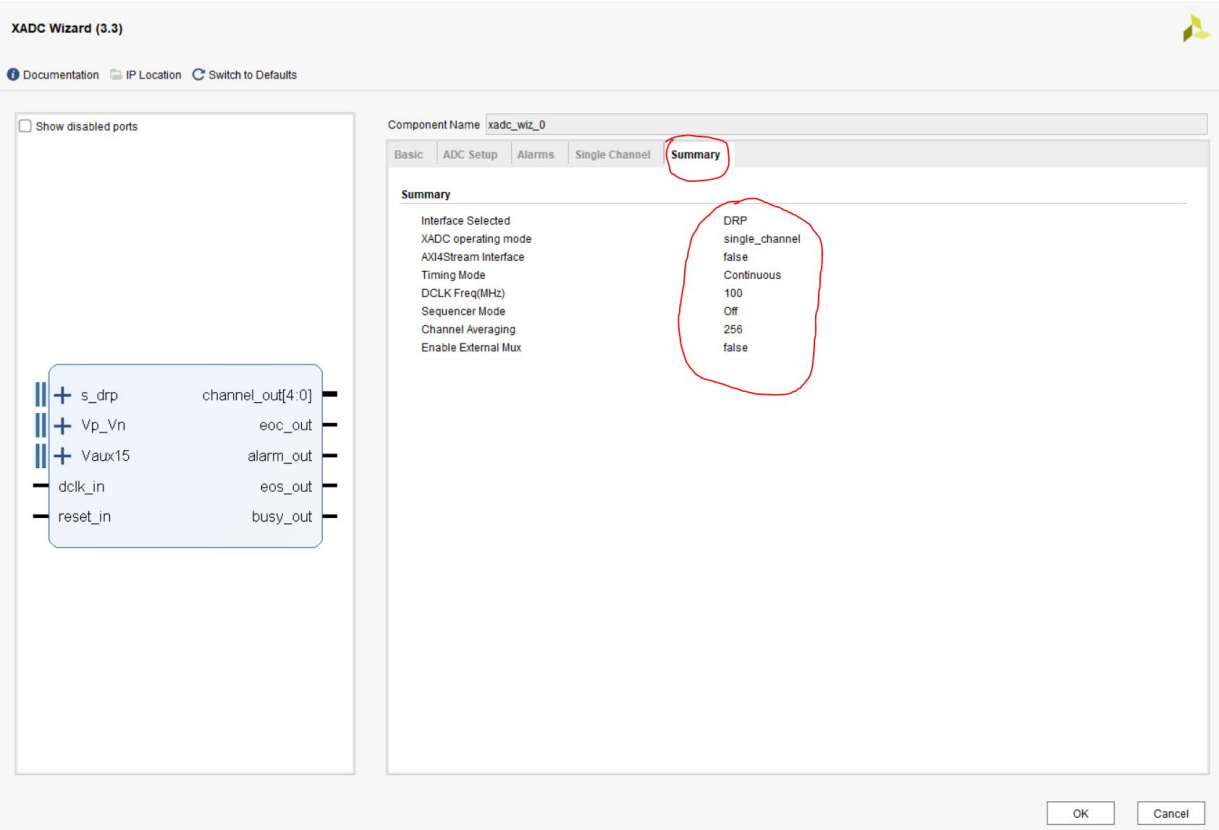
Turn off all the alarms (i.e. de-select them) in the **Alarms** tab and press OK.



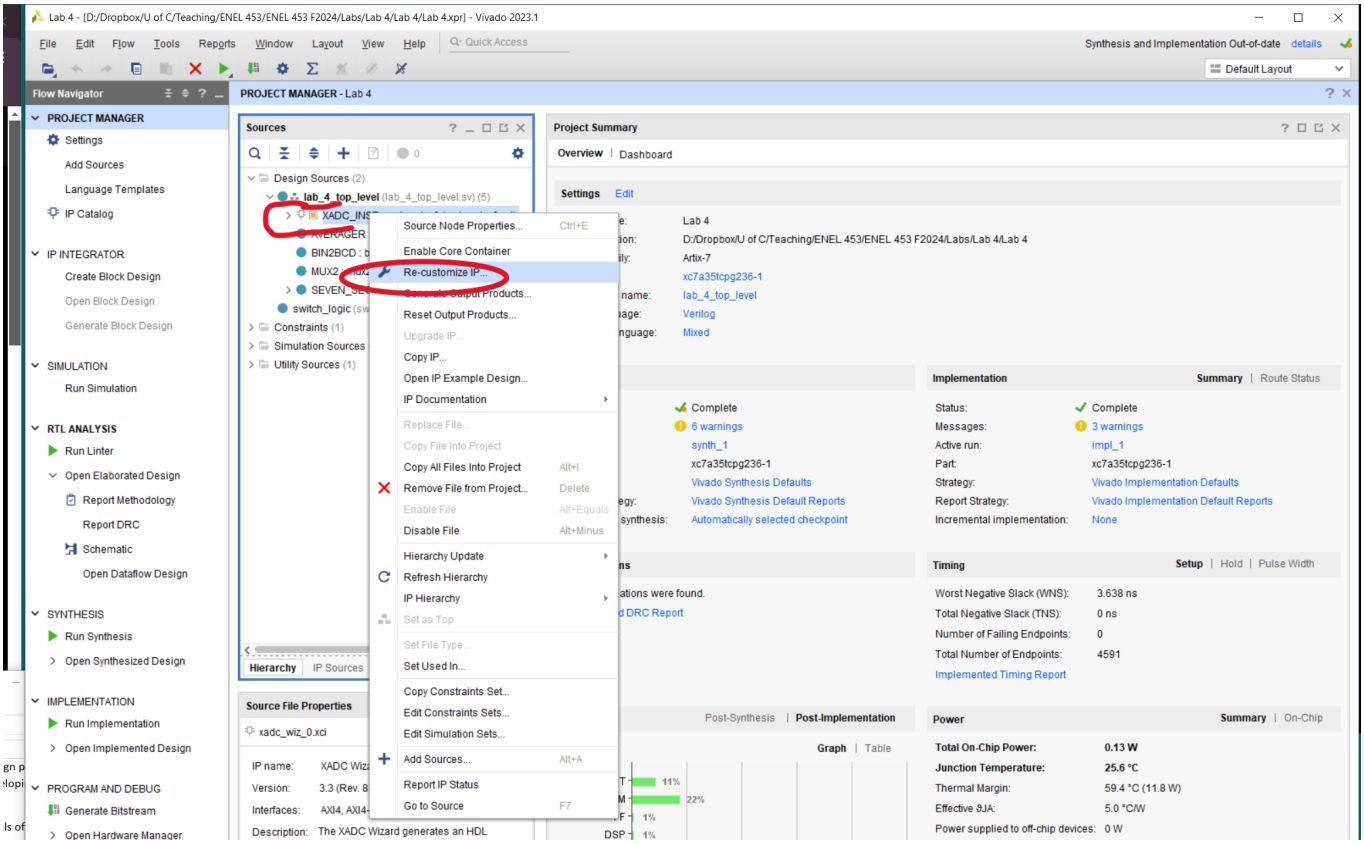
Under the **Single Channel** tab, ensure that `VAUXP15` and `VAUXN15` are selected (you'll need to scroll down!) and the channel is enabled, press OK.



Confirm the **Summary** tab looks like this, press OK. Press OK on the next screen that pops up, to generate the module.



If you need to re-edit the XADC later, you can go back into its configuration by right clicking the **XADC_INST**, then clicking on **Re-customize IP…** and that will bring you back to the XADC Wizard and you can flip through all the setup tabs shown in the previous steps.

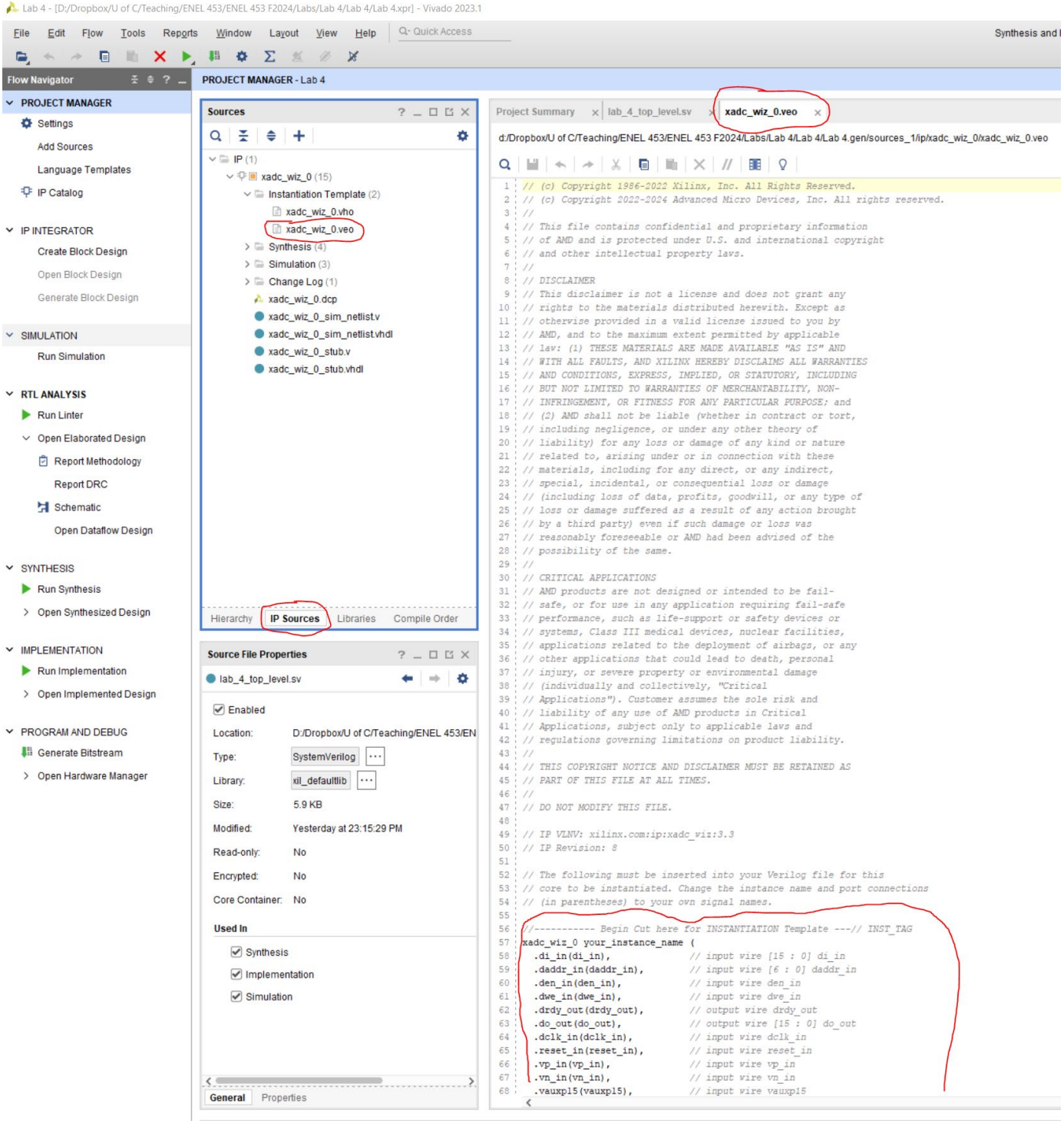


We'll need to instantiate the XADC into our top level. The instructor's solution already has the instantiation, but for reference, below is the conventional way to obtain the instantiation template.

To see the Vivado instantiation template, from Sources, select the IP Sources tab, then:

xadc_wiz_0 > Instantiation Template > xadc_wiz_0.veo (double-click).

This will open a file with the instantiation template to help you bring the XADC into your design.



Students can determine:
- SOME_DECIMAL_FRACTION
- SOME_INTEGER
- SOME_EXPONENT

The instuctor's solution has the correct .XDC. For reference, below are the edits required to correctly specify the XADC inputs within the .XDC constraints file for our specified XADC channel.

Edit the Constraints file (.XDC) to add or modify any top level signals. Be sure to uncomment and edit the signals for the XADC inputs, `vauxp15` and `vauxn15` and make sure they are associated with the correct FPGA pins (N2 and N1, respectively).

```
118 ##################################################
119 ##Pmod Header JXADC
120 #set_property PACKAGE_PIN J3 [get_ports vauxp5]; #set_property -dict { PACKAGE_PIN J3   IOSTANDARD LVCMOS33 } [get_por
121 #set_property -dict { PACKAGE_PIN L3   IOSTANDARD LVCMOS33 } [get_ports {JXADC[1]}];#Sch name = XA2_P
122 #set_property -dict { PACKAGE_PIN M2   IOSTANDARD LVCMOS33 } [get_ports {JXADC[2]}];#Sch name = XA3_P
123 set_property -dict { PACKAGE_PIN N2   IOSTANDARD LVCMOS33 } [get_ports {vauxp15}];#Sch name = XA4_P
124 #set_property PACKAGE_PIN K3 [get_ports vauxn5]; #set_property -dict { PACKAGE_PIN K3   IOSTANDARD LVCMOS33 } [get_por
125 #set_property -dict { PACKAGE_PIN M3   IOSTANDARD LVCMOS33 } [get_ports {JXADC[5]}];#Sch name = XA2_N
126 #set_property -dict { PACKAGE_PIN M1   IOSTANDARD LVCMOS33 } [get_ports {JXADC[6]}];#Sch name = XA3_N
127 set_property -dict { PACKAGE_PIN N1   IOSTANDARD LVCMOS33 } [get_ports {vauxn15}];#Sch name = XA4_N
128
```