Lab 8: Digital Oscilloscope

UC Davis Physics 118
Rev: March 12, 2024

Introduction

In this lab, you're going to learn to do three things:

- Combine custom Verilog code with IP in the block design.
- Use the XADC directly.
- Use mapped dual ported memory to communicate with your custom configuration.

You'll need the control module you designed in the last homework, but if yours didn't work, you can download a working version at

```
Files/FPGA/Lab 8 Files/mem_control.v
```

You'll also need the `base.xdc` file from last time, or you can download it again.

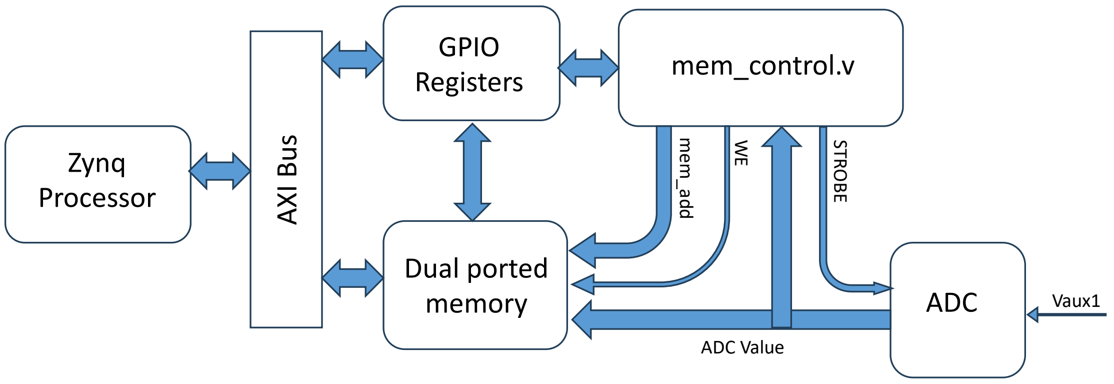


Figure 1: Conceptual view of project.

A conceptual view of the project is shown in Figure 1. The processor will communicate with the Verilog module using GPIO registers. The Verilog module will control the address and write strobe for port B of a dual port memory. It will also issue the convert command to the ADC. In triggered mode, it will monitor the ADC value to determine the trigger memory address and when to begin counting writes.

The processor will monitor the state of the Verilog module to see when it's finished, then read out the data through Port A of the memory.

This is a pretty straightforward, but has a lot of steps. It's basically just following instructions. If your Verilog module works, then all you need to do is connect things correctly.

Building the Oscilloscope

Initializing the Project

As before, create a new project, selecting the Pynq-z2 board. Select RTL Project and “Do not specify sources at this time”. I’ll assume it’s called “Lab_8”. Click “Create Block Design”. Call it “lab_8” or similar.

Instantiate the Zynq processor and do “Run block automation”.

Adding You Verilog Code

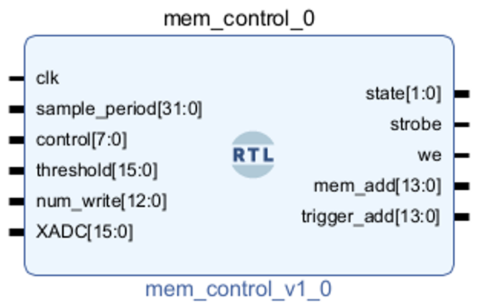


Figure 2: Memory control module.

In the “Sources” tab, right-click on “Design Sources” and select “Add sources...”. Select “Add or create design sources”, click “Next”, and select “Add Files”. Browse to “mem_control.v”, select it, and click “Finish”. It should now appear in “Sources→Design Sources”. Drag it into the block diagram window, and it should appear as a symbol in your block diagram, with the inputs on the left and the outputs on the right, as shown in Figure 2. Run “Connection Automation” to connect the clock.

Setting Up IO Ports

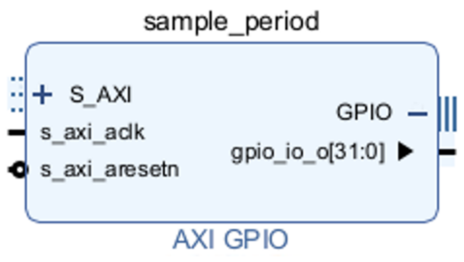
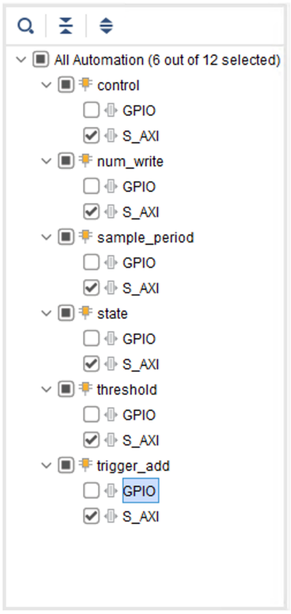


Figure 3: Sample period GPIO register.

We will communicate with our Verilog module via GPIO ports. We could combine some functions into single ports, but in the interest of readability, we’ll create a separate instance for each port.



- All Automation (6 out of 12 selected)
  - control
    - [ ] GPIO
    - [✓] S_AXI
  - num_write
    - [ ] GPIO
    - [✓] S_AXI
  - sample_period
    - [ ] GPIO
    - [✓] S_AXI
  - state
    - [ ] GPIO
    - [✓] S_AXI
  - threshold
    - [ ] GPIO
    - [✓] S_AXI
  - trigger_add
    - [ ] GPIO
    - [✓] S_AXI

Figure 4: Configuration to auto-connect GPIO interfaces.

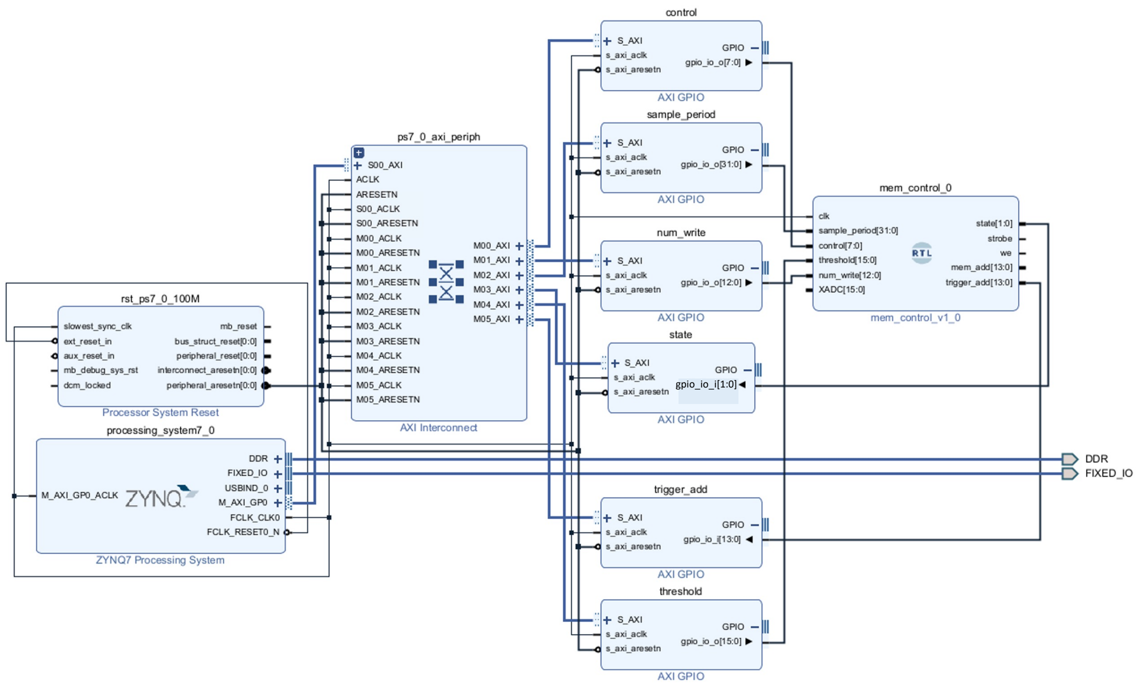


Figure 5: Processor to mem_control interface complete.

Use the IP adder to add instance of "AXI GPIO". Name it "sample_period". Double-click on it, select "IP Configuration", select "All Outputs", set GPIO Width to 32, and click "OK". Expand the GPIO port and verify that it has a single 32 bit output port, as shown in Figure 3.

Now repeat this naming and configuration procedure for all the processor interfaces, setting them to "All outputs" or "All inputs" as appropriate.

- All outputs:
  - **control:** 8 bits.
  - **threshold:** 16 bits.
  - **num_write:** 13 bits.
- All inputs:
  - **state:** 2 bits.
  - **trigger_add:** 14 bits.

When you're done, expand all the GPIO ports and *carefully* verify the size and direction of each port, as well as all the names.

Click "Run Connection Automation". Select all, but then *unselect* all the GPIO interfaces, as shown in Figure 4. This will prevent the automation from connecting the GPIO interfaces to external ports. Click "OK". As before, you'll see an "AXI Interconnect" and a "Processor System Reset" IP appear and get connected all the AXI GPIO instances; however, all the GPIO ports should still be unconnected.

Carefully connect each GPIO port to the associated port on the mem_control symbol. If you do this correctly, the green "Run Connection Automation" options should go away at the top. If it doesn't, *you connected something wrong!* Carefully check your connections again.

Regenerate the layout and it should look something like what's shown in Figure 5.

Adding XADC Instance

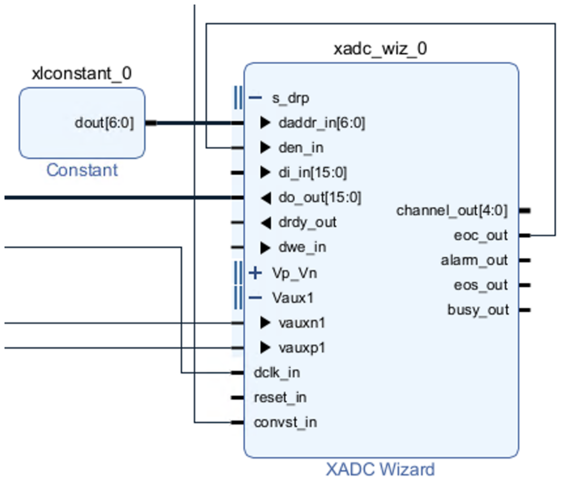


Figure 6: XADC fully connected.

Add the XADC Wizard. Double-click on it and change the defaults as follows:

- Basic tab:

  - Interfact Options: DRP
  - Timing Mode: Event Mode
  - Startup Channel Selection: Channel Sequencer

- Alarms tab:
  - Deselect all.
- Channel Sequencer tab:
  - Deselect all except “vauxp1/vauxn1”

Click “OK”. Use “Run Connection Automation” to connect the clock.

Expand the “s_drp” and “Vaux1” interfaces and make the following connections:

- `eoc_out` on xadc_wiz_0 to `den_in` on the same instance.
- `do_out[15:0]` on xadc_wiz_0 to `XADC[15:0]` on mem_control_0
- `strobe` on mem_control_0 to `convst_in` on xadc_wiz_0.
- As in the last lab, create two input ports `Vaux1_v_n` and `Vaux1_v_p` and connect them to `vauxn1` and `vauxp1` on xadc_wiz_0, respectively.

Import the “base.xdc” file as a constraint and make sure all but those two I/O pins are commented out.

If you’ll recall from the last lab, we access the XADC via registers. These are selected with the `daddr_in[6:0]` input on xadc_wiz_0. We will hardcode this to adress 0x11, which is the address of Vaux1. To do this, add a “constant” IP. Double-click on it and set it to 7 bits and a value of 0x11.

Carefully verify that the XADC has all the connections, and *only* the connections, shown in Figure 6.

Adding the Memory Buffer

We’re going to add the memory and the memory controller. Setting the size of memory is a little confusing, so pay attenction.

From the IP catalog, select “Block Memory Generator”. Double-click on it and on the basic tab, select “True Duel Port RAM”. Click “OK”. Ignore the size at this point, because it’s meaningless.

Add an “AXI BRAM Controller” and name it “bram”. Click on it and under “BRAM Options”, select “1” for the Number of BRAM Interfaces. Click “OK”.

Now click “Run Connection Automation” and select everything under “bram”. This should connect the AXI interface to the AXI bus and the BRAM port to Port A of blk_mem_gen_0, leaving Port B of that module disconnected. Expand Port B to make connections.

Let’s start with the easy ones. Connect `clkb` to the main `FCLK_CLK0`.

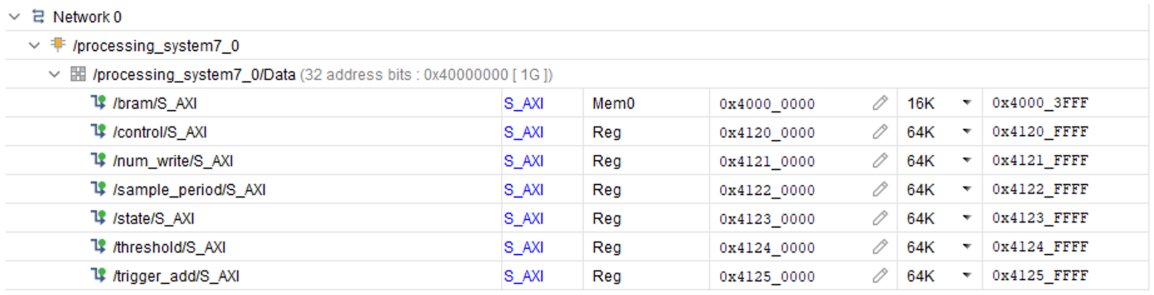
We need to permanently enable the interface, which we do by creating a 1-bit constant, setting it to 1, and tying it `enb`.

The memory is 32-bits wide, but is byte adressable by having a four-bit write enable mask `web[3:0]`. We want to drive them all with `we` from mem_control_0. To do this, we use the "Concat" IP. Instantiate it, click on it, and set the number of inputs equal to 4. Tie the output to `web[3:0]` on the block memory and tie the `we` output of mem_control_0 to *all four* inputs.

We have bus size mismatches for both the address and the data buses. These would actually compile correctly, but would give critical errors, and we want to make sure there are no ignorable errors. In retrospect, it would have been easier to deal with in the design of the memory controller, nut we can do it in the block diagram by using the Concat IP to set the unused lines to zero.

For the memory address, create a Concat instance, change the two intputs from "Auto" to "Manual", set the width of the first to 14 and the width of the second to 18. Tie the output to `addrb[31:0]` on blk_mem_gen_0, `in0[13:0]` to `mem_add[13:0]` from mem_control_0, and `in1[17:0]` to an 18 bit "Constant" instance, set to 0.

For the data, create another Concat instance, change the two intputs from "Auto" to "Manual" and set the width of both to 16. Tie the output to `dinb[31:0]` on blk_mem_gen_0, `in0[15:0]` to `do_out[15:0]` from xadc_wiz_0, and `in1[15:0]` to an 16 bit "Constant" instance, set to 0.


Network 0
/processing_system7_0
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])

/bram/S_AXI	S_AXI	Mem0	0x4000_0000	16K	0x4000_3FFF
/control/S_AXI	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
/num_write/S_AXI	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
/sample_period/S_AXI	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
/state/S_AXI	S_AXI	Reg	0x4123_0000	64K	0x4123_FFFF
/threshold/S_AXI	S_AXI	Reg	0x4124_0000	64K	0x4124_FFFF
/trigger_add/S_AXI	S_AXI	Reg	0x4125_0000	64K	0x4125_FFFF

Figure 7: Set bram to 16K.

Very important! Setting memory size is a bit tricky. Click on "Address Editor" tab. Expand out until you see the "/bram/S_AXI" entry and change it from 8K to 16K (12-bits of 4 byte words). It should look like Figure 7.

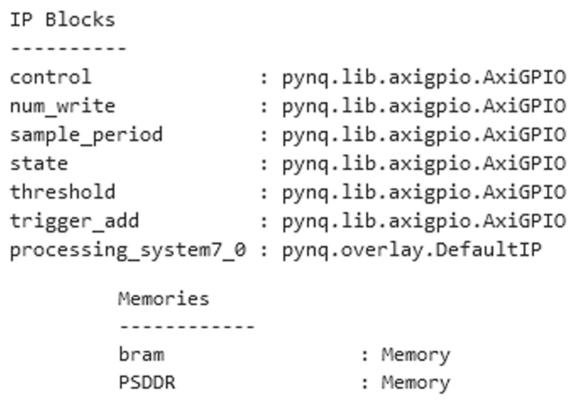
Regenerate the layout and it should look something like the figure in the Appendix.

Compiling Project

As usual, from the sources window, right-click on "lab_8.bd" and select "Create HDL wrapper..." and allow Vivado to maintain it. This should generate NO pop-up errors. If it does, it's probably because some of the bus sizes are mismatched. Read the errors and fix them. Regenerate the HDL wrapper until you get no errors.

Now click "Generate Bitstream" and allow it to run all the intermediate steps. Agian, this should generate NO popup errors. If it does, fix them. When it finishes generating the bitstream, cancel out.

Testing and Using the Oscilloscope



```
IP Blocks
----------
control             : pynq.lib.axigpio.AxiGPIO
num_write           : pynq.lib.axigpio.AxiGPIO
sample_period       : pynq.lib.axigpio.AxiGPIO
state               : pynq.lib.axigpio.AxiGPIO
threshold           : pynq.lib.axigpio.AxiGPIO
trigger_add         : pynq.lib.axigpio.AxiGPIO
processing_system7_0 : pynq.overlay.DefaultIP

Memories
------------
bram              : Memory
PSDDR             : Memory
```

Figure 8: IP blocks and memories.

As before, go to `http://169.254.150.2:9090/lab` in your browser to get to the Jupyter interface. Create a new directory for Lab 8 and copy your new .bit and .hwh files into it (see last week's lab instructions for the details). Make sure they are named "lab_8.bit" and "lab_8.hwh".

Load the files with

```
from pynq import Overlay
pynq = Overlay("lab_8.bit")
```

Execute `help(pynq)` and you should see the IP blocks and memories shown in Figure 8. If you misspelled something or forgot to change a name in your design, don't worry about it. It's not worth rebuilding. Just use whatever name appears here.

Assign all the GPIO interaces to local registers with

```
control = pynq.control
num_write = pynq.num_write
sample_period = pynq.sample_period
state = pynq.state
threshold = pynq.threshold
trigger_add = pynq.trigger_add
```

You don't need to worry about setting the tri-state mask because these were all created to be unidirectional.

Memory map the dual ported memory with

```
bram = pynq.bram.mmio.array
print(bram.shape)
```

And verify that the shape is (4096,). If it's not, go back to your design and make sure the size of bram is 16k, then recompile.

Test the memory interface by executing

```
import numpy as np
fill = np.arange(4096)
bram[:] = fill[:]
print(bram)
```

You should read back sequential numbers from 0 to 4095. Note that it's very important to actually run through the indices as shown with [:]. Simply equating the two would replace `bram` with a pointer to `fill`, which would screw things up in very confusing ways (I'm speaking from experience here).

Now let's do a sanity check. Set up the module as follows

```
sample_period.write(0,100000)  # 100,000  clock cycles
num_write.write(0,4096)        # fill the entire buffer
```

How long do you expect the readout cycle to take in this mode? Test it with the following:

```
import time
from time import sleep

starttime = time.time()
control.write(0,0)            #Always toggle the bit to start the process!
control.write(0,0b00010000)   #Start an untriggered write
while(state.read()!=0):
    sleep(.0001)
endtime = time.time()
print(endtime-starttime)
```

By keeping `control[0]=0` and toggling `control[4]`, we have initiated an untriggered read which will fill all memory locations. **How long did it take?**

If everything has worked fine up until now, let's try a real input. Set the pulse generator to make a sine wave of frequency 1 kHz, with a minumum value of 0V and a maxium of 3V. Connect the ground to a ground on the board and the signal output to input A0 of the Arduino interface. Set the sample period to a value that will give 100 samples per period at this frequency.

Repeat the readout sequency above. **How long did it take?** Fill another array with voltages by multiplying the raw `bram` entries by 3.3/65536.

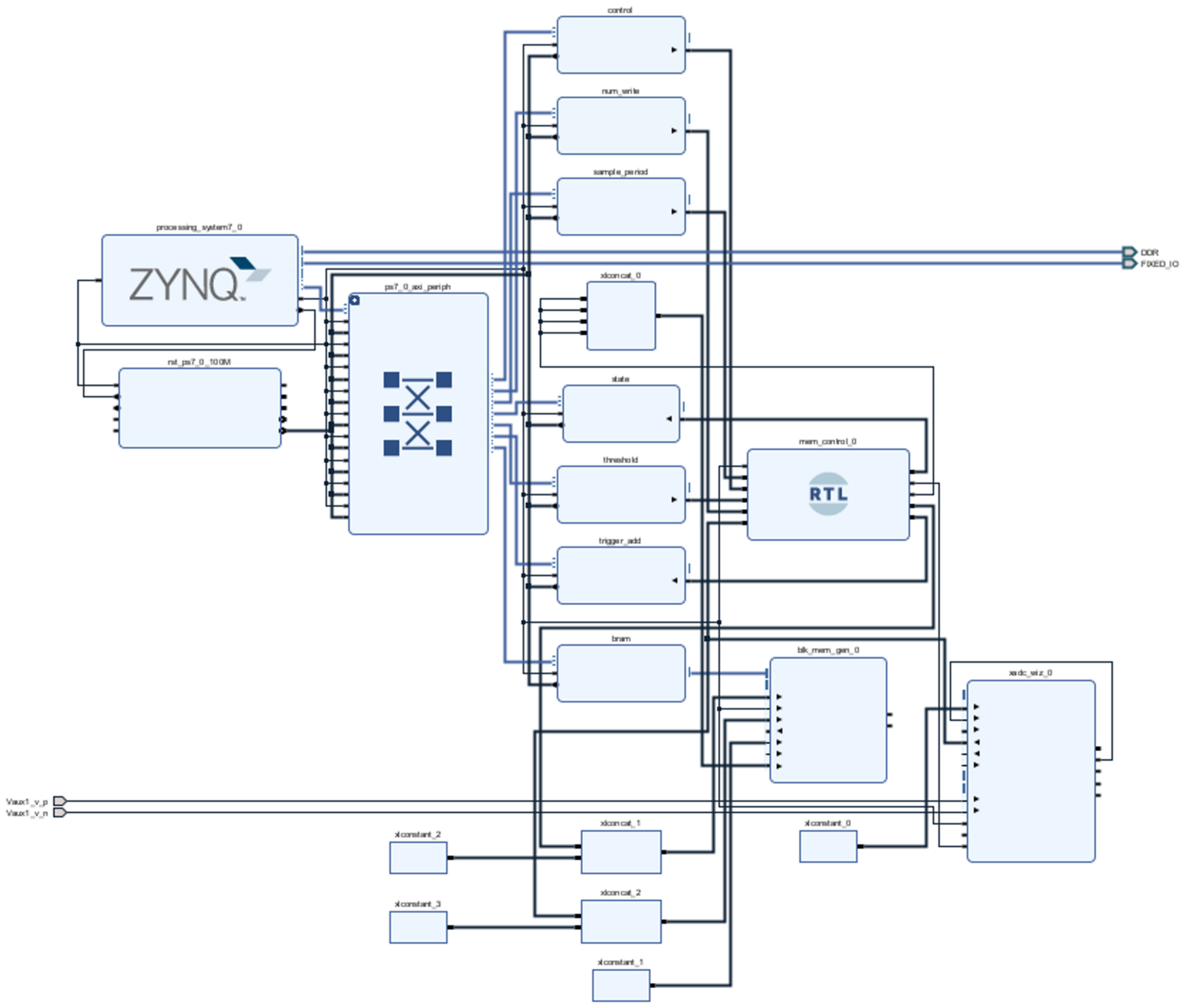
Plot the full voltage values of the full readout against sample number. **Take a screenshot.** Now just plot the first 400 points to get more detail. **Does it look like you expect? Include a screenshot.**

That's enough for now. Keep this design for next week!

Lab Writeup

Include a screenshot of your final design (yes, I know no detail will show) and follow any instructions in **boldface**.

Appendix



Complete design