Cpr E 488: Embedded Systems Design (Spring 2020)

Digital Oscilloscope

Final Project Report

Amith Kopparapu Venkata Boja
Archit Joshi
Vignesh Krishnnan
Section 2 - Group C

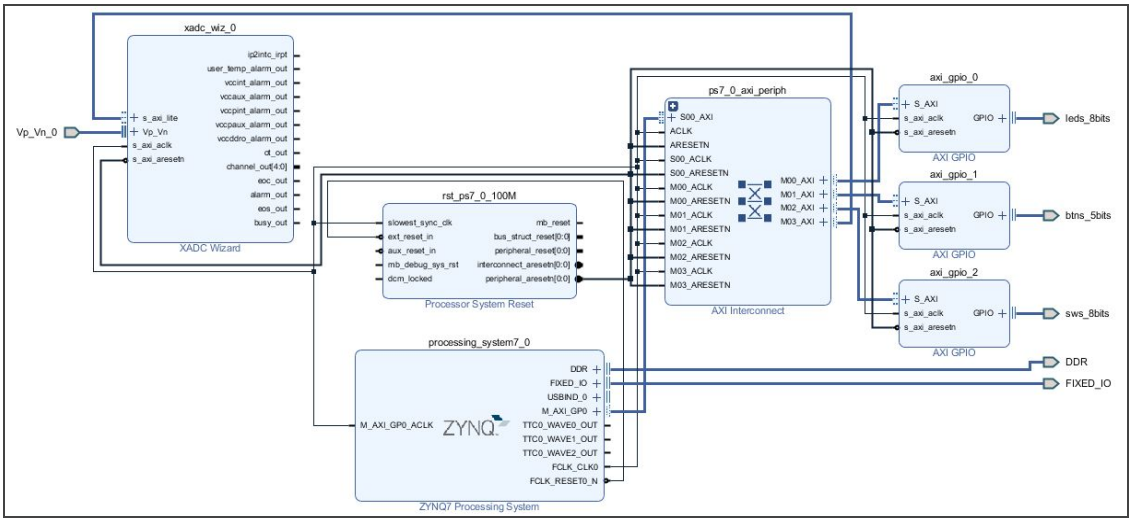
Final Project Report

Introduction:

The goal of our final project is to create a digital oscilloscope GUI in Python using the XADC module in Vivado. We will use serial communication over UART to port data to the GUI from SDK.

Vivado:

In Vivado, we used the XADC Wizard module. We connected this to the processing system and the AXI interface through the slave interface. We used the Vp Vn channel available on the XADC module to input our custom voltage. The XADC accepts 1V range signals so we chose the bipolar configuration to emulate signals with amplitudes of 500 mV ranging (-500mV, 500mV). We constrained these Vp Vn pins in our design to the XADC header pins on the board. This ensured that our voltages were going to the right place in the XADC module. The XADC samples at a rate of 1 MSPS (1 million samples per second). We referenced the XADC datasheet as well as the Zedboard reference manual for these.



SDK:

In SDK, we configured the XADC as we did for many other Xilinx drivers in previous MPs. We used the getADCData function to receive the raw data from the ADC. This function returned a 16-bit integer with 10 bits of MSB being the useful ADC data. The rate at which the getADCData function sampled was about half the speed of the ADC (500,000 samples per second).

For our communication, we used UART, so we configured our UART driver as we did in MP-4. Because we wanted to send one byte at a time over UART we trimmed 2 bits off our 10 bit ADC reading to accommodate that. We lost just a bit of precision, but nothing noticeable.

```
while(1) {
    rawData = XAdcPs_GetAdcData(&driver, XADCPS_CH_VPVN);
    rawData >>= 8;
    rawData &= 0xFF;
    // UART send
    XUartPs_Send(&uart_dev, &rawData, 1);
}
```

We wanted to minimize the processing done on the C side because that's running on the board processor, so we simply ran the getADCData() function, shifted and masked the data for 8 bits, and sent it over UART (as seen above).

Python GUI:

In Python, we read the UART data from the serial port using the pySerial library. The serial port was operating at a baud rate of 115200. With the overhead of stop and parity along with data bits in UART, the GUI could read the data at around 11520 bytes/s. Thus from 500,000 samples/s from the SDK, UART was able to obtain only 11520 samples/s making the serial data transfer as the bottleneck in the communication.

Our Python program was reading the raw values which had to be converted to the voltage data before plotting. We performed this conversion in SDK, in the beginning, to make sure the data is accurate. Then we moved the conversion operation to Python so that instead of processing all 500,000 samples in a second, we were covering only those 11520 samples that made it through the serial port.

Delay:

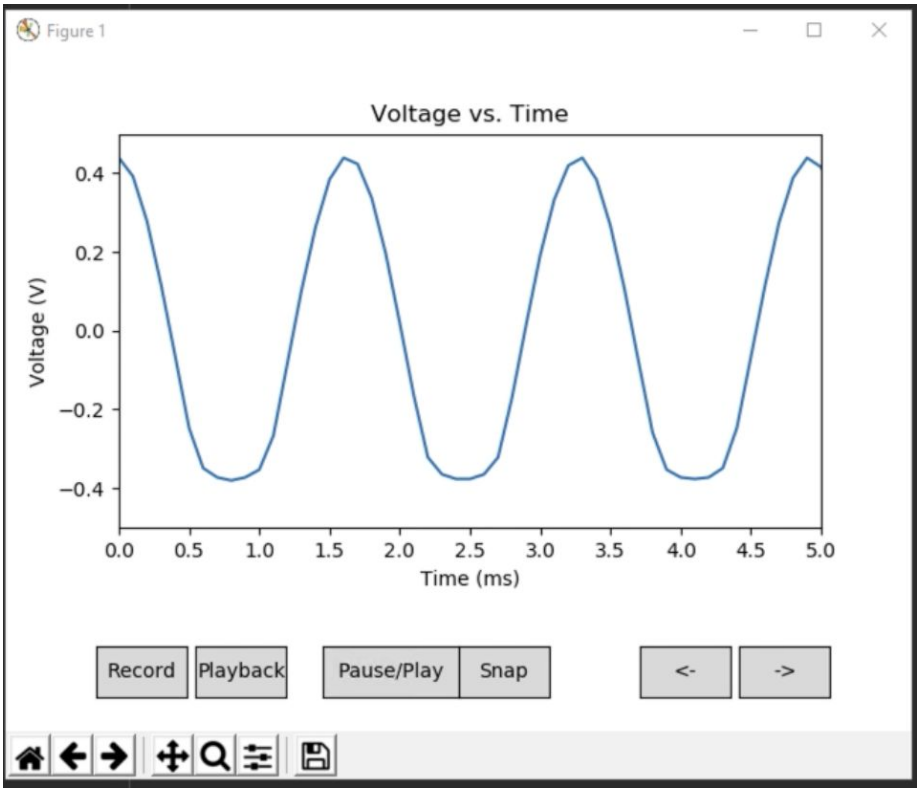
We were not able to implement a trigger module due to time constraints, but we needed some way to clear the UART buffer that was getting data sent to it faster than we could receive. Depending on the frequency, we refreshed the buffer at different points to ensure that our graph displayed with constant and minimal delay.

Plotting:

For plotting voltage vs. time, we used the matplotlib library and animation function to live plot the voltage data that was being read from the serial port.

Results and Additional Features:

- **Graphing ADC data**
  We were able to successfully get ADC data signals and display them in a user-friendly interface
- **Screenshots**
  We were able to let the users take screen snippet of the graph for analysis in a .png image
- **Controlling the graph flow**
  We were able to implement a button to let the user pause the screen, so the user got a better view of the graph when it is very noisy or fast.
- **Scaling**
  We were able to scale the x-axis on the graph so that the user can choose the time divisions as you can in PicoScope. We were also able to keep the axes and other information updated with reference to the scaling that the user decided to choose.



Challenges:

- **Data Transfer**
  The primary bottleneck in our data transfer was the rate of UART. Because it can only receive on the Python side at 11500 bytes/s, we can only display up to 2 kHz signals in a user-friendly manner. The max we can display is 5 kHz (Nyquist rate), but this would be very jagged as there are only 2 samples per wave.

- **Delay**
  We would've liked to have a trigger module that could get ADC data at its sampling rate when certain events happen, but we didn't have time. So refreshing the UART buffer wasn't ideal, but it was our best shot at having a low amount of delay.

- **Recording Data**
  Even though we were not able to play data, we were able to successfully record the raw data, which could be stored in files for users to do further data analysis.

- **Scaling axes**
  In the beginning, our data was plotting with reference to the time the data was being read which kept updating on every instance. We wanted the X-axis values to always start from 0 and show only 10 more values. In this way, the X-axis always has uniform values and it never gets too crowded.

Rubric:

Attributes	1. Beginning- Unsatisfactory- Low Level (25 pt)	2. Accomplished- Satisfactory- Medium Level (50 pt)	3. Exemplary – Beyond Satisfactory- High Level (75 pt)
Capturing ADC data	Able to read ADC values successfully	Able to read ADC values and store the values in memory successfully	Able to read ADC values and store them in memory in real-time
Displaying ADC Data	ADC values received by GUI, but not able to display it in a useful manner	ADC values are received by GUI and the data is displayed in a user-friendly manner	ADC values are received by GUI and the data is displayed in a highly useful manner with interactive features like recording and screenshots
Jitter	Displays signals that do not update to live changes or update with significant delay	Displays signals with acceptable delay to live changes	Displays with minimal and constant delay
Demo and Report	Limited demo and vague report	Working demo and report that hits key points of the project	Fully working and entertaining demo with a thorough report that evaluates all system requirements

Capturing ADC Data:

We think we've reached the satisfactory level here because although we didn't make a hardware module to store values in memory, we are definitely able to read ADC values and store them in Python and graph as such. There is a delay because of our buffer, so it's not exactly in real-time, but pretty close.

Displaying ADC Data:

We think we're exemplary here because we created a few extra features for our GUI other than just displaying the signal. We have buttons for screenshots, changing time scale, pause/play, and a semi-functional recording feature.

Jitter:

In terms of delay and jitter, we are somewhere between satisfactory and exemplary because we display pretty accurately to the PicoScope signal, especially with reference to time scale, but with a constant delay when changing frequency or turning the signal on and off, as seen in our demo video.

Demo and Report:

Pretty good demo and report. Up to the TAs and Dr. Jones!