Reconfigurable Arbitrary Waveform Generator with PYNQ-Z1 Board

Trenton Condrin, Timothy Nunez, Mahan Bastani, Anthony Ho, Carlos Ramos

***Abstract -* In this project we took advantage of the PYNQ-Z1 board to create a reconfigurable waveform generator. The PYNQ board is a derivative of the Z1 board that uses the PYNQ open-source framework. The PYNQ framework allows FPGA developers to import programmable logic circuits as software libraries into Python code. Python “drivers” can then be written to create a simple interface to control the logic circuits. In our project we have a couple hundred lines of Python code that calls a programmable logic circuit we designed in the C programming language and compiled to a binary file using the Xilinx SDK. With this design, we can easily generate any waveform saved to a CSV file through the Python driver we designed.**

***Keywords*PYNQ (Python Productivity for Zynq): An open source framework that enables engineers to design IPs without using conventional ASIC-style design tools.**

**Python Driver: Python class that creates an easy to use interface (i.e. a set of class methods) for using programmable logic circuits (in our case a binary file)**

**.bin File: A binary file that represents a block of logic we can use to program the FPGA portion of the board. We compile these from C code using the Xilinx SDK.**

**Overlay: Premade Python driver made by Xilinx that is on the PYNQ image we used.**

**PMOD DA4: Digital to analog converter that we used in this project.**

**MicroBlaze Processor: A ‘soft’ microprocessor we utilize to send commands and data to our programmable logic circuit.**

I. INTRODUCTION

The motivation of this project comes from the need to be able to generate waveforms of arbitrary shape. This might be used for signal processing purposes or for verification and debugging of hardware devices. For example, an engineer might want to test communication devices by generating specific waveforms and seeing how the device reacts. Using our project, the engineer would only have to create the waveform desired in a CSV file and then run our Python driver.

In addition to the code we wrote for this project, we also utilize the overlays written by Xilinx. For example, in order to interface with the PMOD DA4 chip we used for digital to analog conversion, we used the PMOD overlay on the PYNQ image. An overview of the PYNQ ecosystem and where the drivers, overlays, and binaries/bitstreams fit in, can be observed in Figure 1.

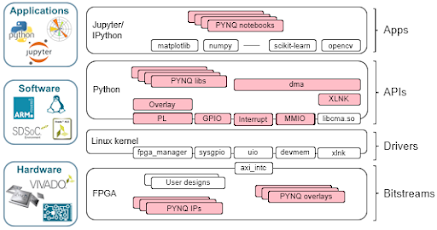
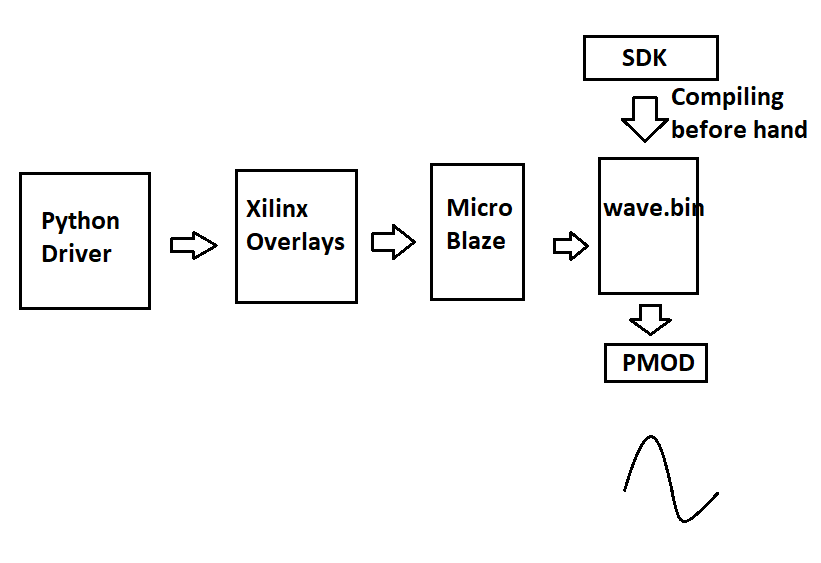


Fig. 1. PYNQ applications and uses across different disciplines of software/hardware engineering.

II. PROJECT DESIGN

An overview of our project’s design and structure can be seen below in Figure 2.

Fig. 2. Overview of our project.

The first component of our project is the most abstract: the Python driver we designed for this project called pmod\_dac.py. Inside of this file we import a few python libraries and Xilinx overlays. We use Python’s built-in library for reading CSVs. This is the only input for our entire project. We also use matplotlib, a python library for plotting graphs, so we can see a software only preview of the wave before sending the data off to the MicroBlaze. Finally, we import the PMOD overlay which allows us to instantiate an instance of the MicroBlaze processor. We must also pass the name of our .bin file to this instance.

After these imports, we begin by calling the write\_arbitrary function of the Pmod\_dac class. The name of CSV file containing the wave in it must be passed as a parameter. Additionally, the PMOD DA4 has a resolution of 4096 and a voltage range of 0 to 2.5V, so the waveform inside the CSV file must be converted to multiples of (2.5 / 4096) so that the waveform can be generated properly. After the proper conversions are made, the values are encoded into 32 bit integers in preparation for writing into memory. Since our step size is only 4096, we encode each value into 12 bit representations since log2(4096) = 12. We then encode each 32 bit memory value with two 12 bit values from the waveform to double the amount of data we can send given a specific amount of memory to use. We then write to the memory (BRAM) using the MicroBlaze’s write\_mailbox method.

After writing to memory, we send an encoded 32 bit command to the .bin file that we then decode using an algorithm wrote in C. Within this command, we pass the length of memory written, so that the C code knows how long the waveform should be. Without this, the C code may cut short or extend the waveform past it’s full period. Inside the C code, we then loop over the memory based on the length received from the command and pass the voltage values through serial protocol interface into the PMOD DA4 chip. One interesting thing to note here is that because of the nature of serial transfer, we are automatically limited to lower frequencies than we might desire from the waveform. For example, say you want a frequency of 500 Hz. Because each value send to the PMOD DA4 is a 32 bit integer (since we must in addition to the 12 bit voltage value encode which channel we wish to select), your maximum frequency know falls to 500 / 32 = 15.6 Hz. Regardless, from here we can read the voltage from the PMOD DA4. We use channel A.

III. RESULTS

In this section we compare the waveform generated within Python to the real analog waveform observed through an oscilloscope

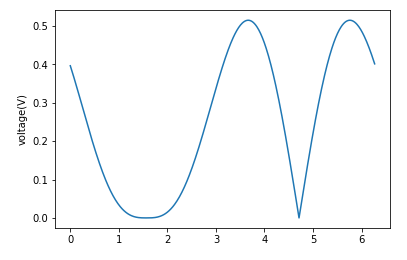


Fig. 3. Waveform 1 plotted in Python.

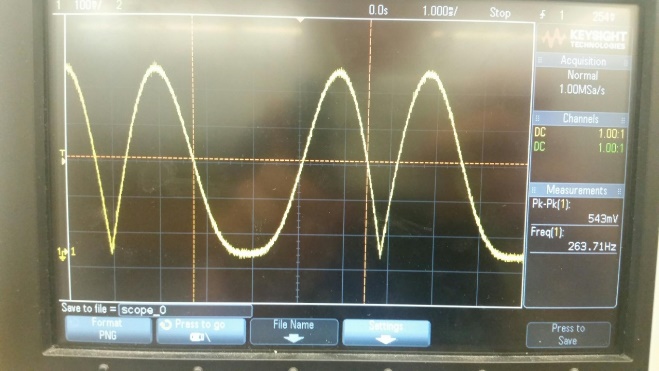


Fig. 4. Waveform 1 as observed on an oscilloscope

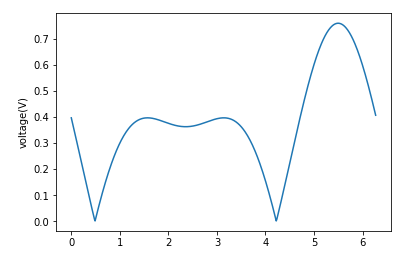


Fig. 5. Waveform 2 plotted in Python.

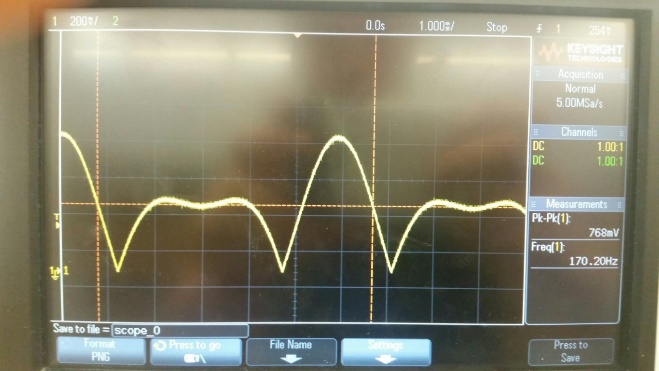


Fig. 6. Waveform 2 as observed on an oscilloscope

IV. ANALYSIS

Our results are spot on to what the data in the CSV file specifies. In addition to the two waveforms shown here, we also tested tons of different waveforms like sawtooth waves and random waves and had success with each test.

V. CHALLENGES AND CONCLUSION

We had plenty of challenges throughout the length of this project. First, we had to understand the nature of the board we were dealing with. It’s not quite a microcontroller or an FPGA, but it has elements of both. The board must be controlled over a network, so it took more than a couple hours just to be able to run a single command. Our method we eventually used for the remainder of the project, is connecting directly to the board through ethernet and configuring our IP address to match the subnet of the board’s IP address. So since the board is 192.168.2.99, we had to set the IP address of our ethernet port to 192.168.2.XXX where XXX is any number that’s not 99 or reserved.

Since we had no exact example to follow for our project, next came the obligatory weeks of trudging through documentation and slowly compiling clues from random forum postings. Eventually we came upon some C code that code generate waveforms of specific types using a PMOD DA4. Using these, we were able to understand how to use SPI to transfer data from our C code to the analog output of the PMOD DA4. We also eventually discovered how to write to memory from Python using the MicroBlaze microprocessor. With these two key components of our project figured out, the rest was just filling in the gaps.

In conclusion, our project achieved exactly what we set out for in our requirements. For future developments, we would like to increase the set of parameters for our arbitrary wave generator, such as allowing the user to set a specific frequency from a range of different frequencies allowed by the length of the waveform file.

VI. SOURCE CODE

In this section is just a brief explanation of the structure of our source code so that any other student or researcher can more easily understand how to setup and run our project.

In the root directory is the pmod\_dac.py file. This file must be placed in the pmod directory of the PYNQ image. This can be found in the links section. This is the class we instantiate in Python to be able to call our arbitrary wave generator. The pmod\_dac\_arb\_gen\_v2.bin file is the compiled .bin file that must be placed in the same directory as the pmod\_dac.py file. This binary contains all the necessary information for the programmable logic on the board. Also can be found in the root is a test CSV file and short script we used to write to the CSV. The only other files in the source is the ‘workspace’ folder. This is the folder that would be imported into the Xilinx SDK in order to compile the C code (pmod\_dac.c, also in the workspace folder). The only caveat is that that pmod\_dac folder within the workspace folder MUST be imported from the PYNQ repository, otherwise the SDK will not be able to locate the necessary header files for pmod\_dac.c. More details about the process for the SDK can be found in instructions.txt file in the workspace folder.

VII. FINAL REMARKS

This project would not have possible without support from Xilinx in allowing us to use the PYNQ board free of charge. We also would like to thank Xilinx for the comprehensive and useful documentation provided for PYNQ, as well as the overlays we used as scaffolding for our project.

VIII. LINKS

[Github](https://github.com/avrmp/computer-architector-project)

[Pmod directory](https://github.com/Xilinx/PYNQ/tree/master/pynq/lib/pmod)

PYNQ [documentation](https://pynq.readthedocs.io/en/v2.4/)