EDL 2024 P16-MON-08
An FPGA-instrument for RRAM characterization with on-board oscilloscope

Documentation

for programming, laptop-FPGA interface & GUI

- Deshpande Varad Shailesh
(for whole S/W)
21d070024

- Harshit Raj and Grampurohit Shreyas Jayant (for GUI)
20d070033 and 21d070029

Salient Features:

1. Microsecond pulsing: Square pulse as small as **100us** can be achieved
2. ADC sampling happens at a rate around **80-100 ksps** (kilo samples per second)
3. Communication of FPGA with laptop happens through the **same USB cable** used to dump code from laptop to FPGA and no other modules are required.
4. A **GUI** on which *pulse value, resistor selection and R_sense* selection can be carried out. Upon hitting the send button, all necessary signals are sent out to the board, which returns the *voltage waveforms for V1 and V2*; These are, in turn, displayed along with the *calculated value of resistance*.
5. Future scope: Can very **easily implement** **pulse width value** selection and **intelligent R_sense selection** since we have kept provisions in place for these.

Code structure:

Entire code is essentially contained in these 5 parts

1. **EDL_merged_code.v**: The file that implements driver for Intel Avalon interface to initiate uart jtag communication with laptop and instantiates the entity for ADC & DAC.
2. **ADC_DAC_control.vhd**: This file has drivers for ADC(MCP3008) and DAC(MCP4921) and also the control logic for storing samples from the ADC, receiving square wave amplitude to the DAC.
3. **uart_jtag**(folder): This has Intel's IPs for the avalon interface. Cannot be changed directly, any manipulation is performed through the entity declared in the .qsys file.
4. **New_gui.py**: Running this file launches the GUI.
5. **helper.py**: To encode the chosen resistor in the network into control signals to be sent to the FPGA.

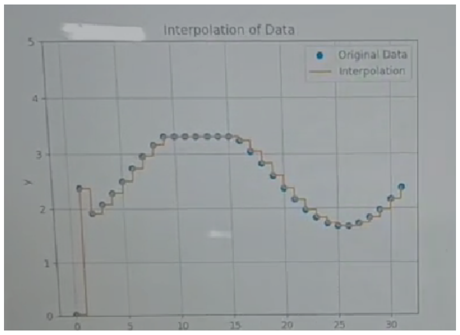
Some important points to note:

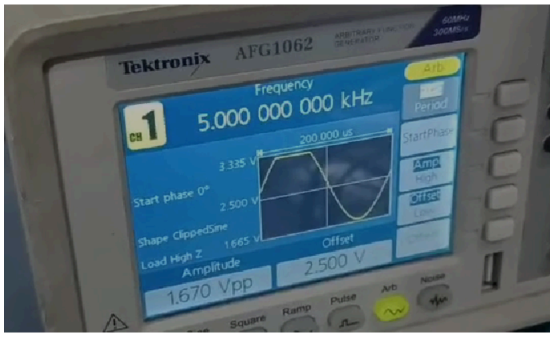
1. Quartus needs to be installed on the device and dumping of .sof file from the computer to the FPGA must be sorted. The device must have any latest version of python installed and the Intel jtag uart python wrapper installed : https://pypi.org/project/intel-jtag-uart/, credits to *Tom Verbeure*.
2. **Everytime** before running the setup, the .sof file needs to be dumped from the Programmer in Quartus to the FPGA by pressing the 'Start button'. Once this is done, the GUI can be launched after ensuring that the power supply to the PCB is at proper values of +5 and -5V. This needs to be done for performing every experiment. This is because there is some problem in resetting the FSM to the initial state, and requires some debugging the state 130;
3. ***The current code sends out a square wave*** to have ease of visualization, this can be changed to send a pulse by suitably modifying code in ADC_DAC_control.vhd
4. Provision has been made to send a pulse width from the laptop to the FPGA with proper encoding in the FSM. Changes would be required to be made in new_gui.py and both the HDL files.
5. Functionality for -5V reset has been provided on PCB and in the code, but needs to be tested, and would involve sending +9V and -5V using the
   Power PCB made by
   Debasish Panda(21d070021)
   to the 4066 switches on board the PCB.

Reason why minor things remain to be given a final touch:
Honestly, I feel looking back that it was way too much work for one person. Especially since programming an FPGA is generally more challenging than programming a microcontroller and that I have very little experience working with a physical FPGA. I was not familiar with verilog and had to learn this on the go. Intel's IP files were in verilog and to implement the driver in vhdl, I would anyway need to experiment with verilog, and since attempts at doing this in vhdl failed, I had to:

a. Learn verilog on the go
b. Write driver for Intel Avalon interface in verilog while experimenting with how it worked

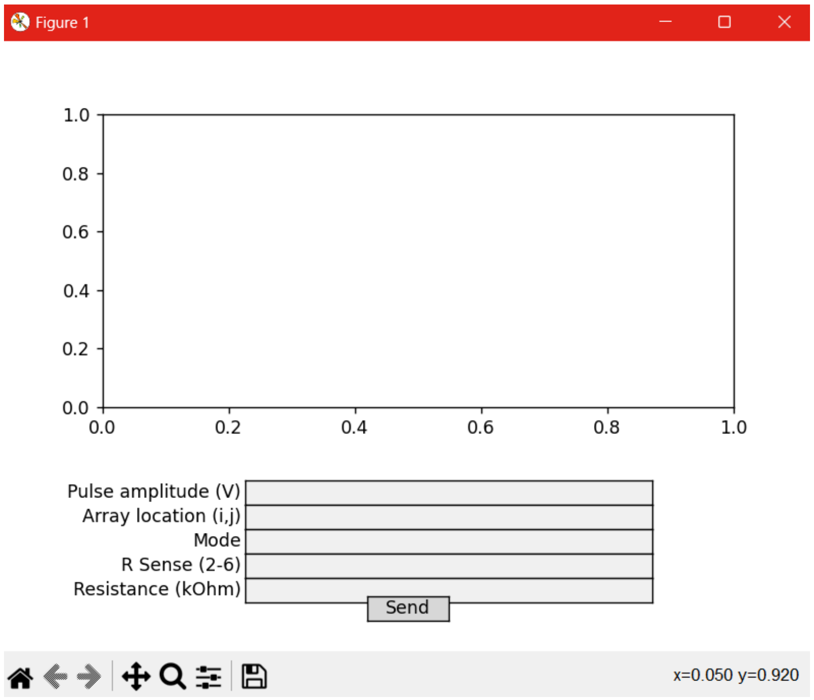
Since the drivers for ADC and DAC were already written in VHDL, I found a way to compile both .v and .vhd files together instead of writing driver files from scratch in verilog.
Had I taken VLSI Design Lab in this 6th semester, my life would have been much much easier.





Sampling a clipped sine: something I'm very happy about. ADC samples correctly, and sends data correctly to the laptop through jtag uart.

GUI Operating Instructions *(and some points on its working):*



1. CHECK First compile the quartus project and dump the code to FPGA using programmer
2. Then run `new_gui.py` to open the GUI. It is in the folder `EDL_merged_code`. It has a dependent file — `helper.py` which contains the logic for control signals and it should be in the same folder for the GUI to run properly. The sections to fill follow below.
3. `Pulse amplitude (V)` — The amplitude of the pulse to give out. Just write a floating point number which is at least 1.5V away from the max supply of the opamps (supplies connected to the `-15V` and `+15V` power headers). For example: `2.56`
4. `Array location (i,j)` — The coordinates of the memristor/resistor to choose for programming/reading in the crossbar. Don't fill in the brackets. For example: `2,3`

5. `Mode` — To decide the mode of operation. It should be either: `Read`, `Write` or `Reset`. **Note:** for resetting, there should be a negative voltage at the `-5V` power header. Only `Read` has been tested extensively.
6. `R Sense (2-6)` — To choose which sense resistor to use (only in `Read` mode). Just enter an integer between 2 and 6 (both inclusive). For example: `3`. The below table shows the corresponding resistance value:

R Sense	Resistance (Ω)
2	1M
3	300k
4	100k
5	30k
6	12k

7. Press the `Send` button after entering the above fields. The resistance value will be calculated in the `Resistance (kOhm)` field. The input and output waveforms, `V1` and `V2` will be plotted on the graph above the entry fields (only in `Read` mode will it be accurate). **Note:** we have accidentally swapped the labels V1 and V2 in the plot. The higher amplitude will be V1.
8. To re-run with new values, close the window and re-program the FPGA using the Quartus programmer. Then again run the `new_gui.py` file. We know it's not ideal but the FPGA code at the moment doesn't support looping back reliably. The GUI instead does support it, in case it is to be used in the future.

Challenges in development:

"Every little signal, every state, every byte destined to be transferred had to be rehearsed with a dance on the on-board FPGA LEDs. I had kept a portable lab handy with a breadboard, some jumpers, a trim pot and a DMM to test my ADC, DAC codes just about anywhere, without a DSO."

1. ADC(MCP3008) and DAC(MCP4921): Took me nearly 1.5 months to make reliable drivers for these. I also wrote drivers for AD5443 and ADS7223. The code might appear somewhat complicated due to its lengthiness. However, it is simple to understand with a counter depicting every clock cycle in the respective timing diagrams.
2. Talking to the FPGA from a laptop using the pre-existing jtag channel which is used to dump code to the FPGA. I followed this tutorial : https://tomverbeure.github.io/2021/05/02/Intel-JTAG-UART.html by Tom Verbeure, which involved writing a driver on the FPGA side that communicates with the Intel Avalon interface,and another of his guide: https://tomverbeure.github.io/2021/05/08/Write-Your-Own-C-and-Python-Clients-for-Intel-JTAG-UART-with-libjtag_atlantic.html for the python side.
   The interface is very intricate and difficult to grasp. I had to spend a great deal of time experimenting with this, changing how every enable signal affected transmission and reception with every clock cycle.
   To ensure reliability, I had to encode every byte transferred to denote its purpose or its place in the sequence. For example, a byte written from the laptop stays in the Intel avalon interface for an unknown amount of time, and the receiving end at the FPGA keeps receiving this byte for an unknown amount of time, until it changes due to transmission of a subsequent byte from laptop. To ensure that the byte being read is indeed the subsequent one and not the previous, I had to toggle the top 2 MSBs of the byte and use this information in conjunction with the knowledge of the present state in the FSM to ensure communication. This "jugaad" if you permit, had to be done due to a lack of time, lack of proper access, and guidance to the Intel Avalon interface.
3. Mixed synthesis of verilog and vhdl files: I had to again guess, try step-by-step these things from the basics for having a poor background in this matter. From passing a single signal to the vhdl entity instantiated in the verilog file to sending large vectors, clock and trigger signals between files.
4. Overall debugging: The painful feeling of realising that something was not working despite being careful. Mostly used LEDs to figure out what went wrong by mapping the signal to the LED output onboard the FPGA. Very limited information accessible at one instant, coupled with the time it took to compile the code(~1 min) made it a very lengthy process.