Running the SPLIFF Firmware

Introduction: You're in the Jungle, Baby

Congratulations! You have been selected as a human sacrifice for the Spectroscopic Lock-In Firmware Fabric (SPLIFF) testing program! The SPLIFF algorithm is a completely novel, patent-pending-pending method of mimicking the functions of an arbitrary number of commercial lock-in amplifiers over a specified spectral range. This FPGA-based system has been specifically designed for use on the Vertex-6 logic array built into the ROACH2 readout platform with the MKID-MUSIC 512 Msps ADC. If that previous sentence seems like word-salad to you, welcome to the cool and carefree world of signal processing! Rest assured you will thoroughly despise your time here.

Part 1: Setting up Hardware

1.1 Hardware Inventory

In order to even begin using the SPLIFF system and all of the fun it has to offer, you first need to have the proper instrument setup. This setup is comprised of numerous parts including:

- The ROACH2 platform housing a Vertex-6 FPGA
- A 512 Msps ADC knows as the "MKID-MUSIC" board which connects to the ROACH2 by way of ZDOK connector
- A clock source running at twice the FPGA fabric rate
    - (2 × 256MHz = 512 MHz clock source)

    - This signal is produced by synth1 on the Valon digital synthesizer, included in your care package
- Noodles, the 2012 Macbook which survived a 70,000 foot drop in the middle of the Arctic Circle
- Virtual Noodles, the Linux virtual machine which lives inside its host, Noodles, like a poorly-behaved tapeworm
- Some signal source which can be modulated by the sync_out port on the ROACH2 PCB (0-3.3V output)
    - This will either be your AOM module or the VDI source

1.2 Hardware Setup

If you are setting up this system from ground zero, follow these steps:

1. Plug the ROACH2 into a wall outlet
2. Ensure that the ADC board is properly attached to the ROACH2 PCB by ZDOK connectors
3. Power on the Valon Synthesizer with 5V power source
4. Connect output of synth1 on Valon to clock port on MUSIC ADC
5. Connect sync_out port on ROACH2 PCB to mod_in port of signal generator to be modulated
6. Connect FPGA to Noodles with the following:
    1. Ethernet cable attached to PPC port on the back panel of ROACH2 (near power cable)
    2. USB-Ethernet connector with USB side plugged into Noodles and ethernet connected to PPC

3. WARNING: Virtual Noodles will **ONLY** connect to the FPGA if it is done over this specific usb-ethernet connector as it has a unique IP address associated. You can connect it to any USB port on Noodles, but the system is constrained by that particular USB-ethernet adapter.

7. Power on the FPGA and load up Virtual Noodles through VirtualBox on Noodles itself

1.3 Testing Connection between Noodles and FPGA

Especially in situations that require frequent disconnections of the ethernet cable between Noodles and the ROACH2, you may find that you are not able to run the plasmon_readout.py script due to local network communication issues. Therefore, every time you start up Virtual Noodles, the FPGA, or whenever you disconnect and reconnect the two, I highly recommend pinging the FPGA to ensure a connection. This can be done by opening up a terminal on Virtual Noodles (shortcut: ctrl-alt-t) and entering the following command:

```
ping 192.168.40.79
```

If the connection is successful, you will receive in response a bunch of information on the packets sent between the two systems and how long the response time is. If the two are not connected, you will receive a similar response except instead of some time in milliseconds for latency, it will read HOST_UNREACHABLE.

1.3.1 Uh Oh Host is Unreachable

In the very likely situation that you do receive this error message, don't panic (unless you lost the USB to Ethernet adapter, in which case you are free to panic at your leisure). First, try waiting 30 seconds or so before pinging the FPGA again. Especially if you have just connected the ethernet into either source, it can take up to two minutes for the dnsmasq to properly assign a hardware address. Keep pinging that same address and 85% of the time it will eventually start responding.

In the case that smashing the enter button more times doesn't fix the problem, consider the following:

1. Check the ethernet to USB adapter. If you dont see two lights, one blinking occasionally and the other mostly stable, its likely that you forgot to turn on the FPGA. Hit the black button on the face of the ROACH2, chuckle to yourself about how silly of a move that was, and try pinging again in a minute.
2. If you DO see lights on the adapter, you will have to restart the dnsmasq protocol and clear the cache of assigned hardware addresses
    1. Open a terminal on Virtual Noodles
    2. Run the command: `sudo systemctl restart dnsmasq`
    3. Enter the password Remember25
    4. Unplug the ethernet cable from either end of the connection
    5. Wait 30 seconds
    6. Reconnect the ethernet cable; this will hopefully reassign a hardware address to the FPGA

    7. Wait another 30 seconds
    8. Return to pinging the FPGA until it eventually responds
    9. If after 2 or 3 minutes it still does not respond, return to step 1
3. If it STILL won't connect, just call me and save yourself a day of misery and heartache

As ridiculous as the may seem, I have had situations where I have had to restart the dnsmasq 6 or 7 times before finally getting the two systems to communicate. So don't give up!

Part 2: Firing up the Readout Script

2.1 Introduction to Software

As you are probably aware, almost all of the functions needed to talk to and operate the SPLIFF are located in a single python script on Virtual Noodles. This script is named "plasmon_readout.py" and is located in the directory:
`~/Destop/Photomixer_LISS`
Therefore, to access this directory from a terminal, first open one up (duh) and use the directory command:
`cd ~/Destop/Photomixer_LISS`

Inside the Photomixer_LISS folder is a variety of scripts, CSV files, and organic gluten-free memes from the 2013 Golden Era of the internet. All that you care about is one, and that is plasmon_readout.py

2.2 Opening an Interactive Python Console

NOTE: Make sure that your terminal window has navigated the Photomixer_LISS folder before trying the following:

1. Use the simple command `ls` and hit enter to see a directory listing of all the files in the current folder directory
2. Make sure that the folder you are in contains a file named "plasmon_readout.py"
3. Once confirmed, use the command `ipython` to open up an interactive Python environment in the Linux terminal

Now that you are in a python environment, feel free to play around with any basic python functions you want to get a handle on. While not necessary, I would suggest understanding the NumPy package and in particular array slicing and indexing so that if you want to change any of the arrays of data in the readout script, the language isn't completely foreign (this mainly applies to people who have extensive MatLab experience and may be tripped up by the compiler indexing arrays at 0 instead of 1).

2.2.1 Side note: Navigating iPython

iPython is a simple command-line environment with full python3 capabilities but has a few oddities to it, especially in combination with the monstrosity that is the plasmon_readout script.

- Once you execute the `ipython` command, you will be in a separate design environment than before but still in the same terminal window. This means that hitting the 'up' arrow while in iPython will go through only the history of iPython commands and not any terminal commands used previously

- In order to exit the iPython environment without closing the terminal window entirely, use the command `exit()`
- If you are running a function such as a live-plotter that updates semi-infinitely, you can close out of the function with the keyboard shortcut `ctrl-c` . If for any reason that does not end the function and you cannot exit out of iPython with the `exit()` command but do not want to close your precious terminal window, the shortcut `ctrl-z` will kill any running processes and auto-exit iPython while maintaining your current terminal environment window

2.3 Running plasmon_readout.py

Assuming you are now comfortably acquainted with the iPython design environment, we can finally move ahead with calling the readout script itself. This can be executed in a single line with the command `run plasmon_readout.py` . If done successfully, you should see some lines print out that tell you that you are connected to the FPGA and that the firmware is loaded or already has been loaded. Some more lines will print with debug information as well as a flex on how fast I made the flashing process. Once you see a message telling you that the NAND gates have been flashed and that the firmware is ready to go, you can begin using the functionality of the readout script.
NOTE: You can actively open and edit the plasmon_readout.py script and even overwrite the file through the TeamViewer file transfer system while it is running. Any saved changes to the readout script will not take effect until you rerun the command `run plasmon_readout.py` , allowing parallel software editing and system measurement.

Part 3: One Script to Rule them All

Alright, you've made it. You have navigated the labyrinth that is Linux iPython and now have successfully executed the plasmon_readout.py program; the FPGA logic has been flashed with the SPLIFF algorithm and your Linux terminal is able to actively control it. Now it is time to understand what DSP incantations you have at your fingertips. Welcome to Hogwarts kiddo.

*Editor's Note: This explanation is for the plasmon_readout program as of 3-20-2023. With modifications or added functionality, the exact references made here may not carry over perfectly. I will do my best to make this a generalized guide and keep it updated as time moves forward, but there may be slight inaccuracies. I get paid $25k a year so either deal with it or pay me to start caring. Jk jk (but actually not jk), love you <3.

3.1 Pre-Function Declarations

Before diving into the different functions available to you in this program, you may be curious about what on Earth is going on in the lines of code before the first defined function. Long story short, this is where we declare all of the different global variables including: IP address of the FPGA, hardware port address, sampling frequency of the digitizer, the length of the FFT, and most importantly, the .fpg file to be used. If you decide to just ignore all of that spaghetti code, that is completely fine and will not effect how much value you can extract from the program, however, anybody running the script needs to be aware of where the firmware file is declared.

3.1.1 Packages Needed and Packages Not So Needed

Currently, the readout script imports 7 different python packages to help lower the complexity of writing all functions from scratch. The declared packages are as follows:

1. *import casperfpga* -- casperfpga is arguably the most essential package required for interfacing with a ROACH2 FPGA. It is the result of nearly two decades of open-sourcing the blood sweat and tears poured out by the CASPER astronomy readout community and has contributions from over 100 different authors (*including yours truly*). You will only see the package called once since a few lines after its declaration, we use its KatcpFpga class to wrap all of the properties of out ROACH2 board in a simple little class variable we call 'fpga'. This is one of the few packages you cannot download through pip3 so if for some reason Noodles evacuates its bowels and deletes all of its Python packages, you will need to refer to the official CASPER Berkeley website to redownload.
2. *import time* -- The time package is a simple set of functions that allows the Python compiler to do things such as 'sleep' in between lines of code. Doing so is helpful both for user experience and for metering the CPU demand of certain functions, thus allowing Noodles to not burst into flames while plotting an FFT or accumulator.
3. *import matplotlib.pyploy as plt* -- Matlab is good for one thing: plotting. The point of the matplotlib package is to allow Python users the same tools that are available for plotting in Matlab. By importing specifically the pyplot class as plt, we

are able to quickly plot functions the same way they would be plotted in Matlab with the simple plt.plot() command.

4. *import struct* -- The structure package is not commonly used and there's no reason to learn why I'm using it. I just like being able to optimize a code by 0.5% through proper data-structure morphing.
5. *import numpy as np* -- Oh a Matlab license costs $1000/yr outside of educational bundle packages? It would be a shame if a group of 10,000 programmers decided to rewrite all of the functions into Python for free. On a more serious note, the true benefit of numpy is in the compiler. Typically Python compiles your code and runs it in series, line-by-line. Numpy on the other hand (much like Matlab) has its own C++ based compiler that allows for parallel computation, exponentially increasing the performance of cumbersome code.
6. *import scipy.stats as stats* -- Python has a lovely set of baked-in statistics functions that work fine and dandy if you're just some goofy lookin', "hello world" writing, "Serpinski-triangle-is-my-homework" talkin' script kiddie who just wants to automate their fantasy football draft. For those **adults** who want to take **unnecessary amounts of data to plot a semi-arbitrary statistical correlation and call it "Allan variance"**, we need something with a little more heft to it. The stats package lets us do statistics faster, but only if you're already melting the thermal paste on the MOSFETs of your gaming laptop because you're too impatient to wait 15 minutes for a code to finish.

7. *import csv* and *from csv import writer* -- In the scenario where you want to collect so much data that storing it all in the (admittedly meager) RAM of the virtual machine isn't possible, the next best move is to save it all to hard drive. The way I have done this throughout the readout script is by writing data to CSV files which can be opened in Excel or re-read by separate Python scripts.

3.1.2 Swapping out .fpg files

If you take nothing else from this section, just make sure to read this:

Replacing the .fpg file -- If and when the time comes where I need to modify something in the firmware that requires a full re-compiling of the entire design, there is only a single line in the code that needs to be changed. After changing the design, I will provide you with a small file over email with the file extension ".fpg". In order to switch out the current firmware design for the updated one, take the .fpg file I send and move it to the folder from which you are running the python script (i.e. `~/Desktop/Photomixer_LISS/`). You do not have to purge the previous .fpg file in use as they should have unique names, however make sure to note the name of both the old and the new .fpg files. Once placed into the correct directory, open up plasmon_readout.py in your favorite text editor (VIM is the easiest to access but I wont judge you for taking the easy way out and using the built-in Linux text editor). Inside the python script, look for the statement around 20 lines in that reads:

```
firmware_fpg = 'liss_gold_enhanced_v1.fpg'
```

All you need to do is change the name of that file to the newest .fpg file I have provided, making sure to put 'quotes' around the file name so that python knows it is supposed to search its current directory. If you plan on playing around with more than one .fpg file for some god-forsaken reason, copy and paste the line above with the other .fpg file name and comment it out using the default python commenting command '#'. For example, if I were to send you a new design called 'liss_gold_enhanced_v2.fpg' and you wanted to load it in without completely deleting the old file name, just keep it in with commenting, i.e.,

```
# Using the new v2 firmware, 100% more sauce than v1

# firmware_fpg = 'liss_gold_enhanced_v1.fpg'
firmware_fpg = 'liss_gold_enhanced_v2.fpg'
```

Now, when you run the plasmon_readout.py script, it will use the v2 .fpg file since v1 is commented out. When you want to change back, all you have to do is swap which line has the '#' mark and rerun the python code:

```
# Actually, looks like v1 had more sauce, so let's use that one
again

firmware_fpg = 'liss_gold_enhanced_v1.fpg'
# firmware_fpg = 'liss_gold_enhanced_v2.fpg'
```

3.1.3 Other Declarations

Apart from the packages and the .fpg file, there are a few more variables which we want to reset every time we execute the plasmon_readout.py script. The most relevant of these are at the end of the preamble in the block of code that reads:

```
fpga.write_int('fft_shift', 2**9)
fpga.write_int('cordic_freq', 1) #
fpga.write_int('cum_trigger_accum_len', 2**24-1) # 2**24/2**9
=2**15
fpga.write_int('cum_trigger_accum_reset', 0) #
fpga.write_int('cum_trigger_accum_reset', 1) #
fpga.write_int('cum_trigger_accum_reset', 0) #
fpga.write_int('start_dac', 0) #
fpga.write_int('start_dac', 1) #
```

The *FFT shift* is a scaling factor dependent on the length of the FFT and will never change unless the digital polyphase filter bank circuit changes, which is not going to happen any time soon, so don't mess with it. On the other hand, the *CORDIC frequency* is defaulted to 1 which is equivalent to ~3.9KHz and sets the initial modulation/demodulation rate of the lock-in system. Changing the integer value of 'cordic_freq' from 1 will give you integer multiples of 3.9KHz. This default value can be changed as needed and should be set to 26 if you want a ~100KHz modulation rate. The only other declaration in this block that has any reason to change is the fabled 'cum_trigger_accum_len' or in English, the *sample length of the accumulator*. It defaults to a length of $2^{24} - 1$ and cannot be set any higher, lest the rest of the

functions cease working. If for some reason you want to lower the amount of integration done on hardware, lower the exponent from 24 to the value you choose, but **keep the -1**, otherwise all of your accumulator plots will look like butt. Since it is a base-2 exponential value, every integer reduction of the exponent from 24 will cut your hardware integration time in half ($2^{24}-1 \rightarrow 65\text{ms}$ while $2^{23}-1 \rightarrow 32.5\text{ms}$).

The remaining declarations before the first functions are flipping trigger switches in order to start up the counters which keeps the firmware from completely unwinding and going haywire. I would suggest leaving them alone.

3.2 Basic FPGA Control

Although the SPLIFF firmware is mostly set in stone, there are ways to interact with it through communications with the FPGA. As seen in the previous section on miscellaneous declarations, there are numerous variables that allow some knob turning within the firmware itself. The most important variables to worry about for general use are 'cordic_freq' and 'cum_trigger_accum_len' which change the modulation frequency and hardware integration time respectively. These variables, along with any other modifiable fpga variable parameter can be viewed as a list with the command: `fpga.listdev()` . This can come in handy when you forget the precise spelling of any variables or just have a brain fart and forget what everything is called.

Along with listing the tunable parameters of the firmware, you can read or overwrite their current values. To write an integer value to any FPGA variable parameter, use the command:

`fpga.write_int('variable_name_in_quotes', integer)` replacing the inputs for your desired variable and new value. If you want to check what the current value for a firmware variable is, use the similar command: `fpga.read_int('variable_name_in_quotes')` .

If not immediately noticeable, all of the aforementioned commands available to modify firmware parameters are prefixed by " `fpga.` " which in Python implies that we are executing functions available to some class of variable that we have deemed the same name. Indeed this is true and in our particular case, " `fpga.` " is just convenient shorthand for the address and IP of our ROACH2 board which is declared a member of the 'casperfpga.katcp_fpga' class of objects in the first few lines of code. While this again may seem like word salad, 'casperfpga.katcp_fpga' is a class of Python object created by the CASPER group that inherits a series of useful FPGA tweaking functions such as the aforementioned '`read_int`' and '`listdev`'. It is not at all necessary or even recommended to learn the intricacies of the functions we impart onto our ROACH2 board and its firmware by declaring it a member of the KatCP class, but it is useful to understand what this declaration generally means in order to debug and find answers when problems arise or new features are needed.

With that covered, it is time to move on to the meat and potatoes of the script; the functions. I will be grouping functions together here by their usage for the sake of convenience, but in the readout script they are scattered about. This will hopefully get cleaned up in further updates to the software whenever Carly gets back from her trip.

3.3 Defined Functions

3.3.1 Reading Raw RAM Data -- `read_accum_snap()`

Arguably the most important function in the entire script, this gives you access to the in-phase and quadrature components of the complex accumulator data in its raw spectral format. The digital values from the accumulator block are read in as streams of $2^4 \times 2^9$ datapoints. Why that particular length? Well, the factor of 2^9 comes from the length of the FFTs; this value is then doubled in length as our ADC expects a double-ended input, doubled *again* since each spectra is complex with an I and a Q component, doubled **again** since we actually have a true spectral length comprised of 1024 channels and doubled ***AGAIN*** since every clock cycle of the FPGA fabric contains two samples from the digitizer, in total multiplying the datarate of each spectra by a factor of 2^4. Once the datapoints are read from the RAM block, they are split into real and imaginary components and returned to the user. Note that this means the `read_accum_snap()` function returns two arrays when run, requiring two separate datastructures to properly load the hardware accumulator information into software.

3.3.2 Plotting Functions `plot_ADC( )`

```
**TO BE CONTINUED**
```