Xilinx-ZCU111 with Primecam firmware: lab usage

Logan Foote and Chris Albert

Spring 2024

The Primecam firmware for the Xilinx-ZCU111 is set up for KID observations using the Observatory Control System (OCS) designed for the Simons Observatory. Lab usage requires working around these features to extract the desired information from the board.

1 Limitations

Hard-coded numbers in the Primecam firmware restrict the usage of the Xilinx board to below its full capabilities. The bandwidth is limited to 500 MHz, so multiple LO frequencies are required to readout a chip with a larger bandwidth. The firmware is capable of running four channels at once, so it is possible to combine the outputs and split the inputs with filters to cover a band of 2 GHz. This work is in progress. The other major limitation is the readout frequency of 488 Hz, while the board is capable of up to 500 kHz.

2 Initial setup

2.1 Imaging the SD card

First, the board SD card needs to be imaged. The image can be found here.

2.2 Set up Ethernet connections

After imaging the board, control (COM) and noise streaming (NOISE) Ethernet cables should be connected to the DAQ computer, and the board can be turned on. The noise streaming Ethernet cable should be connected to the built-in Ethernet port on the DAQ computer. The IP address, subnet mask, and gateway should be set manually on the DAQ computer to

COM: 192.168.2.25, 255.255.255.0, 192.168.2.1

NOISE: 192.168.3.40, 255.255.255.0, 192.168.3.1

Also, the mtu on both ports should be set to 9000. At this point, it should be possible to ssh into the xilinx board using

```
ssh xilinx@192.168.2.98
```

The password to the xilinx board is 'xilinx'. If the firewall is blocking access, run

```
sudo ufw allow 6379
```

If the COM port is not connected, check the physical connections and make sure the RFSoC is on. If it is still not connected, the SD card was imaging was not successful. The NOISE port will remain unconnected until initialization code is run on the board in a later step.

The home directory on the board is /home/xilinx/. The image has a few folders that include the name 'CCATpHive' that should be deleted.

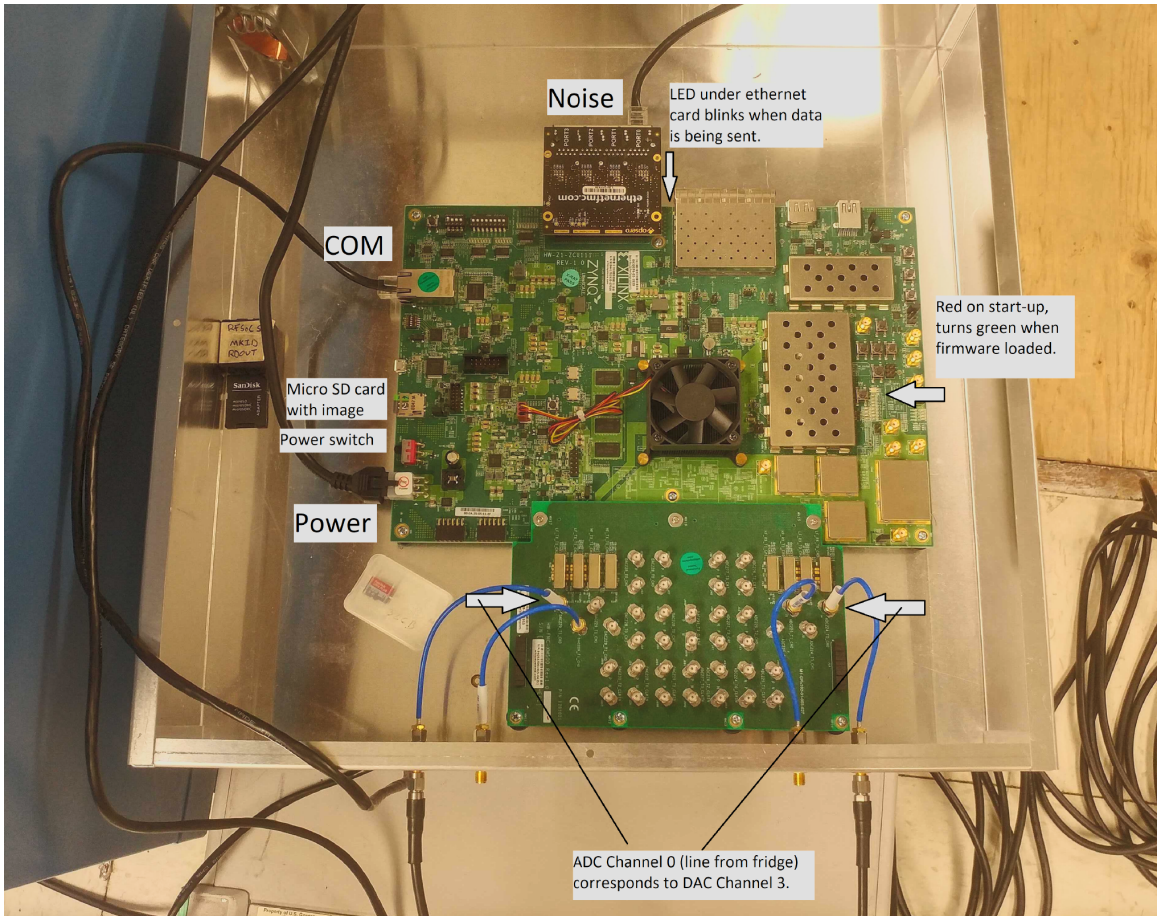

Figure 1: Xilinx board layout

2.3 Clone repositories

Clone primecam_readout and citkid onto the daq computer. primecam_readout is the Primecam readout software that is run on both the Xilinx board and the DAQ computer, so it must be cloned to the Xilinx board and pulled whenever it is pulled on the DAQ computer. From the Xilinx board, run

```
git clone <user>@192.168.2.25:<path to the repository on daq computer>
```

To pull, run

```
git pull <user>@192.168.2.25:<path to the repository on daq computer>
```

The file structure in primecam_readout changes a lot. I will do my best to keep citkid up to date with it, but I would recommend against pulling often unless you want to access a new feature.

Install citkid as a module via the README file. If you install in developer mode, you won't have to reinstall each time you pull. This repository contains the code you need to work around the primecam software to use the RFSoC as a lab system. It also contains some data acquisition and analysis procedures that may be of interest.

2.4 Set up configuration files

In the primecam_readout folder on both the DAQ computer and the xilinx board, configuration files must be set up. First, navigate to /cfg/ and copy the files '_cfg_queen.bak.py' to '_cfg_queen.py' and '_cfg_board.bak.py' to '_cfg_board.py'. in '_cfg_queen.py', set the following parameters:

```
host = 192.168.2.25
```

In '_cfg_board.py', set the following parameters:

```
host = 192.168.2.25
udp_dest_mac = <daq computer mac address>
```

The DAQ computer MAC address should be the MAC address of the NOISE ethernet port. Ensure that both of these new configuration files are the same on both the board and the host computer.

2.5 Set up redis server

The redis server is required for noise streaming. First, ensure that the right packages are installed by running

```
sudo apt install lsb-release curl gpg
```

Then add the repository to the apt index, update it, and install

```
curl -fsSL https://packages.redis.io/gpg | sudo gpg --dearmor -o\
/usr/share/keyrings/redis-archive-keyring.gpg

echo "deb [signed-by=/usr/share/keyrings/redis-archive-keyring.gpg]\
https://packages.redis.io/deb $(lsb_release -cs) main" | sudo tee\
/etc/apt/sources.list.d/redis.list

sudo apt-get update
sudo apt-get install redis
```

The configuration file for redis is stored in citkid/primecam/redis.conf. Navigate to this directory on the DAQ computer, and run

```
redis-server redis.conf
```

If the redis server is shut down incorrectly, the task will still run in the background. It needs to be shutdown before another server can be run. Run

```
ps aux | grep redis-server
```

to list the task numbers containing redis-server, and shut them down using

```
sudo kill -9 <task number>
```

If this doesn't work, restarting the DAQ computer will fix it. Use CTRL-C to shut down the server correctly before closing the terminal window.

3 Running the RFSoC

3.1 Setup

With the ethernet cables connected, start the redis server on the DAQ computer using

```
redis-server redis.conf
```

The redis server must be running to connect to the RFSoC. Next, ssh into the board and navigate to the 'init' folder. Run

```
sudo python3 init_multi.py
```

to upload the firmware to the board. This will take a minute. Then navigate to the 'src' folder and run

```
sudo python3 drone.py 1
```

The index 1 represents channel 1 on the board, which is the outer-most SMA connectors (see figure 1. You can also run a separate drone for channels 2, 3, and/or 4 in a separate window. This drone will recieve and execute commands from the DAQ computer, and send data back. It is helpful to keep this window visible while taking data, since it will print out the status of commands. At this point, one can send commands via the terminal on the DAQ computer. To make sure the connection is working, I usually use the built-in graphic interface. Navigate to /src/ on the DAQ computer and run

```
python queen_gui.py
```

Try setting the LO frequency with the command 'setNCLO'. The board.drone argument is 1.1 for board 1 and drone 1, and the argument is the LO frequency in MHz. Then, try writing a VNA comb using 1.1 for the board.drone and no arguments. This will output 1,000 evenly-spaced tones in a 500 MHz bandwidth around the LO frequency. At this point, noise can be streamed using the graphic interface to test the noise connection. If noise is not streaming, go back to the previous steps and ensure that everything was set up correctly. Close down the graphic interface when you are done testing.

3.2 Basic commands

Start with the output of the RFSoC connected to the input of the cryostat, but the output of the cryostat disconnected. To run the RFSoC from python, first import the RFSOC class.

```
from citkid.primecam.instrument import RFSOC
```

The primecam firmware cannot transfer data directly. Instead, it is automatically saved in a temporary directory named 'tmp' in the same location as the python code. It also saves log information in a directory names 'logs' one directory up from the python code. The class RFSOC finds the appropriate files after they are transferred, renames them, and transfers them to a directory specified by the user as 'out_directory'. The RFSOC class also takes the board ID, drone ID, and NOISE IP address as parameters. The user must supply the local path to the primecam_readout repository, since it cannot be installed as a module. After creating an RFSoC instance, the user can run commands to set parameters and take data.

3.2.1 LO frequency

First, the LO frequency can be set with

```
rfsoc.set_nclo(frequency)
```

where the frequency is in MHz with 1 MHz precision.

3.2.2 Tone output

To output tones, run

```
rfsoc.write_targ_comb_from_custom(fres, ares, pres)
```

where fres is a list of tone frequencies in MHz, ares is a list of powers in RFSoC power units, and pres is a list of phases. To auto-generate pres or ares, set the parameters to None. The RFSoC power unit is proportional to power. For 1,000 tones, a safe value to set all of the resonators is 260. This corresponds to a power of roughly -55 dBm per tone. This value can go up if the number of resonators is lower. When pres auto-generates, it generates random phases until it gets a total waveform that doesn't saturate the DAC. You may run into issues if the tone powers are too high.

With tones generated at roughly the level you expect, check the input power into the RFSoC. It should be at 0 dBm, but not much higher. Adjust the attenuation/gain accordingly, then plug the output cable into the RFSoC.

There is a special case of tone generation, which creates 1,000 evenly-spaced tones around the LO frequency at the highest power. This case is used for a full VNA sweep, and run with

```
rfsoc.write_vna_comb()
```

3.2.3 S21 sweeps

To execute an S21 sweep, run

```
rfsoc.target_sweep(filename, npoints, bandwidth, N_accums)
```

where filename is the name of the saved file (must end in '.npy'), npoints is the number of points (per tone), bandwidth is the width of the sweep in MHz, and N_accums is the averaging number. The saved data is (frequency, I, Q) of the sweep, where each npoints in the arrays represent a single tone.

There is a special case of S21 sweeping where the bandwidth is set to the spacing between tones. This case is used for the full VNA sweep, and run with

```
rfsoc.vna_sweep(filename, npoints, N_accums)
```

3.2.4 Noise

When tones are outputting, noise is streaming over the NOISE Ethernet port. Noise packets can be captured using

```
rfsoc.capture_noise(seconds)
```

where seconds is the number of seconds of data to capture. The output is I and Q timestreams, where I, Q are lists where each index corresponds to the resonator index and each value is an array of timestream data. This function will crash if there is not enough memory to store the noise data. 200 s is a safe timestream length. The number of tones should not affect this. For longer timestreams, it is safer to call the function multiple times in a row. If you really care about syncronizing in between the calls, it is possible to save while taking data, but I have not set up code to do this.

To capture and save the noise data, instead run

```
rfsoc.capture_noise(seconds, filename)
```

where filename is the name of the file (ending in '.npy').

3.3 Procedures

Two procedures are currently implemented in primecam.procedures: 'take_iq_noise' and 'optimize_ares'. A function named 'make_cal_tones' is also included for adding calibration tones to a list of resonances.

3.3.1 Sweeps and noise

The function 'take_iq_noise' is used to take S21 sweep data around resonators after the resonance frequencies have been found, and optionally take noise. The user passes in a list of resonance frequencies, list of amplitudes, list of Qs for cutting data, and list of calibration tone indices. The user also passes in a file suffix for naming the output files.

Qres is the list of Qs for cutting data. This is not a list of the total quality factor of the resonators, but rather values that correspond to the total width of the resonator. Resonances should span fres / Qres. Qres is used to ensure that the frequency updating code does not bounce between nearby resonators. Later, Qres will also be passed into the analysis code for cutting resonators out of the gain scan.

The data required for analysis is a fine scan of the resonances, gain scan of the gain at each resonance, and noise timestream. Before calling the function, the user must have set up the rfsoc and set the LO frequency. The tones also must be centered on the noise frequencies of each resonator. If the user wants to fine-tune frequencies within the function, they can turn on the optional 'rough sweep', which runs at the beginning of the function and performs a rough sweep of the resonances to update their frequencies. The user can specify the frequency update method as one of the following: 'distance' is the point of furthers distance from the off-resonance point in complex IQ space, 'spacing' is the point of maximum spacing between adjacent points in complex IQ space, 'minS21' is the point where the magnitude of S21 is at a minimum after a first-order polynomial is removed from the data, and 'none' bypasses updating the resonance frequency.

The user can specify the bandwidth and number of points of each of the three scans: rough, fine, and gain. The user can also specify the length of the noise timestream, or set the parameter to None to bypass taking noise. The maximum noise timestream length is around 500 seconds before the code will crash. It is a subject of future work to figure out how to take longer timestreams. For now, the parameter 'nnoise_timestreams' is included to take multiple noise timestreams in a row. There will be a small time gap in between each timestream.

For analysis of data taken with this function, see Sec. 3.4.1 and 3.4.2

3.3.2 Power optimization

The function 'optimize_ares' is used to optimize tone powers for each resonator. The user passes in a list of frequencies, starting tone powers, Qs, and calibration tone indices (see Sec. 3.3.1. A safe starting value for the amplitudes is 50. The user also specifies the maximum output power per tone for the optimization. The user passes in some parameters that are in turn passed into 'take_iq_noise'. The user also specifies a plot directory to plot progress at the end of each iteration.

It is recommended that the user first runs 'take_iq_noise' with noise turned off and analyzes the IQ data with plots to ensure that their system is set up properly. The procedure iterates through IQ loop fitting, and updates power at each iteration. Power is not updated on the calibration tone indices. Also, the user may choose whether or not frequencies are updated and using which method with 'fres_update_method'. The first round of updates scales the power by a factor related to the nonlinearity parameter to get the nonlinearity parameter close to the target value. The target value is currently enforced to be 0.5, but I want to make this variable in a future update. The last few iterations, specified by 'n_addonly', only add or subtract 1 dB to try to get all the resonators within a certain threshold of 0.5. The combination of these two methods works well to optimize powers.

For highly irregular resonators that do not fit well the the resonance model, it may be possible to optimize powers using noise time streams by measuring the ratio of parallel to perpendicular noise. We are experimenting with this method, but it is not implemented yet.

3.3.3 Make calibration tones

The function 'make_cal_tones' is used to insert calibration tones in between resonances. The function sorts the resonance frequencies by greatest spacing and inserts tones in between them until there are tones between all resonators or maximum number of tones specified has been reached.

3.4 Analysis

Analysis code is provided for the output of 'take_iq_noise'. The analysis is broken into two functions: IQ sweep fitting and noise analysis.

3.4.1 IQ sweep fitting

IQ sweep fitting is performed with 'fit_iq'. The data directory and output directory are specified, along with the file suffix that was used when taking data. Some parameters are included for logging purposes: the power and temperature indices, for power and/or temperature sweeps, the attenuations, the temperature, and any other desired parameters to log with 'extra_fitdata_values'. The resonator indices can be specified as a list if they are different than the indices into the data. Fit plots can be turned on/off, and the user can specify 'plot_factor' to plot only a subset of the data. The output is plots in a folder called 'plots_iq' in the plot directory, and a csv file in the out directory with the name f'fitdata{file_suffix}.csv'.

3.4.2 Noise analysis

After IQ sweeps have been fit, noise can be fit using 'analyze_noise'. The user must specify the directory containing the IQ sweep fit data csv file, the file suffix, and the noise index. When multiple sets of noise are taken, 'analyze_noise' must be called multiple times for each noise index. The rest of the parameters are passed into the noise analysis function, except 'catch_exceptions'. If 'catch_exceptions' is True, the code will

continue running when the noise analysis fails on a particular resonator, and insert NAN into the output data. The function saves plots into a directory named 'noise_plots', noise data a into a directory names 'noise_data' and a csv file names 'fitdata_noise{file_suffix}_{noise_index:02d}.csv'. See the noise analysis write-up for a description of the rest of the analysis.