Synopsis - Week 13 Integrated Project: Hardware Acceleration

Prof. Jordan C. Hanson

October 16, 2023

1 Finite Impulse Response (FIR) Filter

A finite impulse response (FIR) filter takes digitized data as the input, and returns modified data of the same shape as the output. Let n represent the number of samples of digitized data. Let $x[n]$ represent the vector of digitized input data, and $y[n]$ represent the digitized output data. Let z^{-1} represent an operation that selects the prior sample, or the $n-1$ sample relative to sample n. Let the coefficients b_n multiply the samples $x[n]$ as they pass through an amplifier. Finally, let Σ represent an adder that sums to inputs and produces one output. A generalized circuit diagram for the FIR filter is shown in Fig. 2, and the formula relating the input to the output is given in Eq. 1.

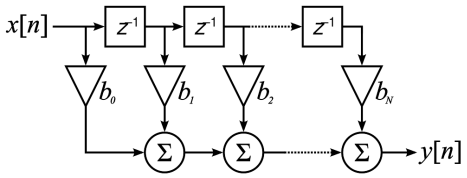

Figure 1: A generalized circuit diagram for the FIR filter.

$$y[n] = \sum_{i=0}^{N} b_i x[n-i] \tag{1}$$

In this lab, we are going to create a PYNQ overlay that will implement Eq. 2 in the SoC PL layer. We will show that:

- Implementing the FIR filter in the PL layer *accelerates* the results
- The acceleration depends on N, the number of data samples

2 Example of an FIR Filter in SciPy

Copy the following code into a Jupyter Notebook on the PYNQ-Z1 board:

```
import matplotlib.pyplot as plt
def plot_to_notebook(time_sec,in_signal,n_samples,out_signal=None):
    plt.figure()
    plt.subplot(1, 1, 1)
    plt.xlabel('Time (usec)')
    plt.grid()
    plt.plot(time_sec[:n_samples]*1e6,in_signal[:n_samples],'y-',label='Input signal')
    if out_signal is not None:
        plt.plot(time_sec[:n_samples]*1e6,out_signal[:n_samples],'g-',linewidth=2,label='FIR output')
    plt.legend()
```

This Python function will plot time samples in microseconds versus data samples in Volts. If a second signal is given, it will be plotted with the same time samples. The following code creates a list of time samples, sampled at 100 MHz, that is 1 ms long. The result is 10^5 samples. There are 10^5 samples because the *sampling rate* is 100 million samples per second, for 1 ms. The `linspace` function is used to create a list of $N = 10^5$ samples evenly spaced between 0 and 1 ms.

```
import numpy as np
# Total time
T = 0.001
# Sampling frequency
fs = 100e6
# Number of samples
n = int(T * fs)
# Time vector in seconds
t = np.linspace(0, T, n, endpoint=False)
# Samples of the signal
samples = 10000*np.sin(0.2e6*2*np.pi*t) + 1500*np.cos(46e6*2*np.pi*t) + 2000*np.sin(12e6*2*np.pi*t)
# Convert samples to 32-bit integers
samples = samples.astype(np.int32)
print('Number of samples: ',len(samples))
# Plot signal to the notebook
plot_to_notebook(t,samples,2000)
```

The signal samples are the sum of three sinusoidal functions, with frequencies 0.2 MHz, 46 MHz, and 12 MHz. The plot shows a noisy sine wave. Our goal is to filter the higher frequencies, leaving just the first signal at 0.2 MHz. Let V_{in} be the input amplitude in Volts, and V_{out} be the output amplitude in Volts. Filtering the higher frequencies means we must use a *low-pass filter*. It follows that $P_{\text{in}} \propto V_{\text{in}}^2$, and $P_{\text{out}} \propto V_{\text{out}}^2$. Let the *gain* of a filter in decibels be

$$G_{\text{dB}} = 20 \log_{10} \left(\frac{V_{\text{out}}}{V_{\text{in}}} \right) \tag{2}$$

$$G_{\text{dB}} = 10 \log_{10} \left(\frac{P_{\text{out}}}{P_{\text{in}}} \right) \tag{3}$$

The gain versus frequency of our FIR low-pass filter is shown in Fig. 2. Notice that 0 dB means the input and output have the same amplitude and power. A gain of -3 dB implies the output power is half that of the input.

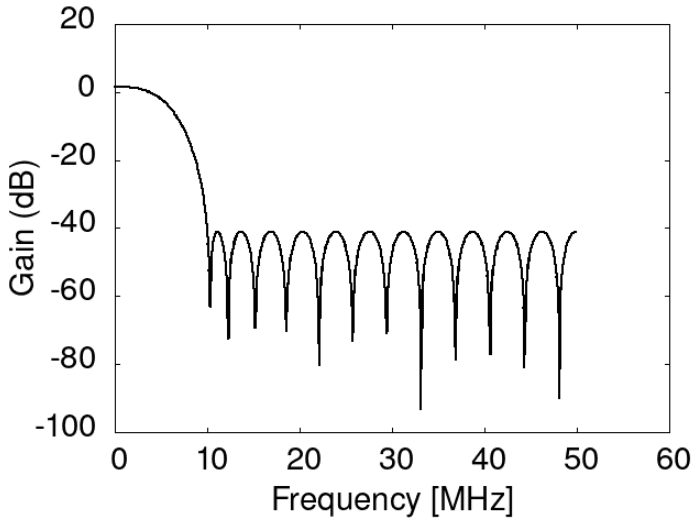

Figure 2: The gain in dB versus frequency of the FIR low-pass filter we will use in this lab activity.

1. What will happen to our signal, composed of three amplitudes at the three frequencies 0.2 MHz, 12 MHz, and 46 MHz, when it is passed through our FIR low-pass filter?

2. If we pass a 1 Volt, 20 MHz signal through our FIR low-pass filter, what will the final amplitude be in (a) dB (according to Fig. 2), and (b) in Volts?

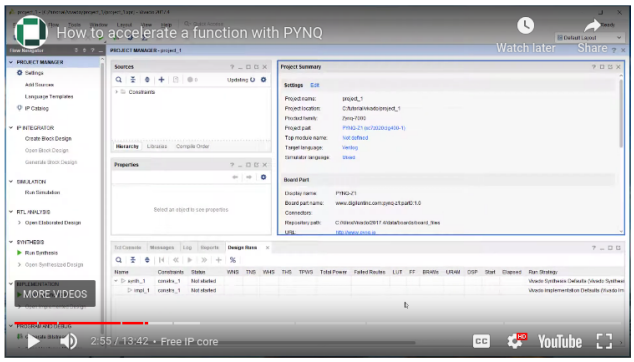

Figure 3: Beginning a firmware project in Vivado.

The following code implements the FIR low-pass filter in Python. Copy this code into your notebook for execution. Make note of the SciPy function `lfilter` that implements Eq. 1 with the coefficients b_i given by the array `coeffs`.

```
from scipy.signal import lfilter
import time
coeffs = [-255,-260,-312,-288,-144,153,616,1233,1963,2739,3474,4081,4481,4620,4481,4081,3474,2739,
1963,1233,616,153,-144,-288,-312,-260,-255]
start_time = time.time()
sw_fir_output = lfilter(coeffs,100e3,samples)
stop_time = time.time()
sw_exec_time = stop_time - start_time
print('Software FIR execution time: ',sw_exec_time)
# Plot the result to notebook
plot_to_notebook(t,samples,2000,out_signal=sw_fir_output)
```

Notice the execution time of the `lfilter` function is recorded in the variable `sw_exec_time`. What is a typical execution time for the `lfilter` function. How does it appear to scale with N, the total number of samples?

3 Creating a Block Diagram in Vivado with FIR Filter and DMA

On the ASUS laptop, open the application *Vivado* using the menu at the bottom left of the desktop environment. Create a new project by choosing a project name and ensuring that it is an RTL (register-transfer level) project. In the Default part dialogue, choose the Boards tab, then search for the PYNQ-Z1 board. Select the PYNQ-Z1 board, then click Finish. The screen should now resemble Fig. 3. Locate the IP Integrator section on the left-hand column. Click IP Integrator, and click OK in pop-up window named Create Block Design. Using this functionality, we are going to add some digital logic to the standard PYNQ-Z1 PL. In the diagram space, click the + button, search for Zynq (the Xilinx SoC that powers the PYNQ-Z1). Add the Zynq system to the diagram, which should resemble 4 below. Notice the bar at the top of the block diagram inviting the user to run block automation. Run this to set the normal initial conditions to the Zynq block.

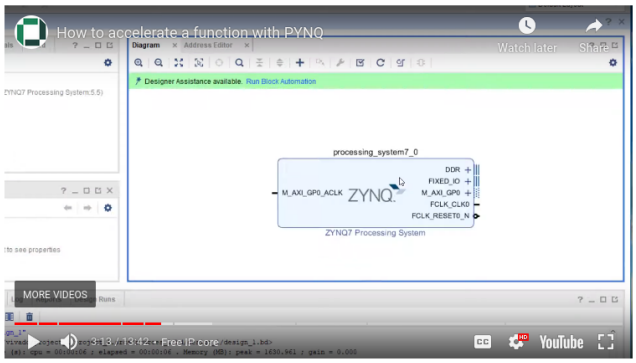

Figure 4: Adding the Zynq SoC to the block diagram.

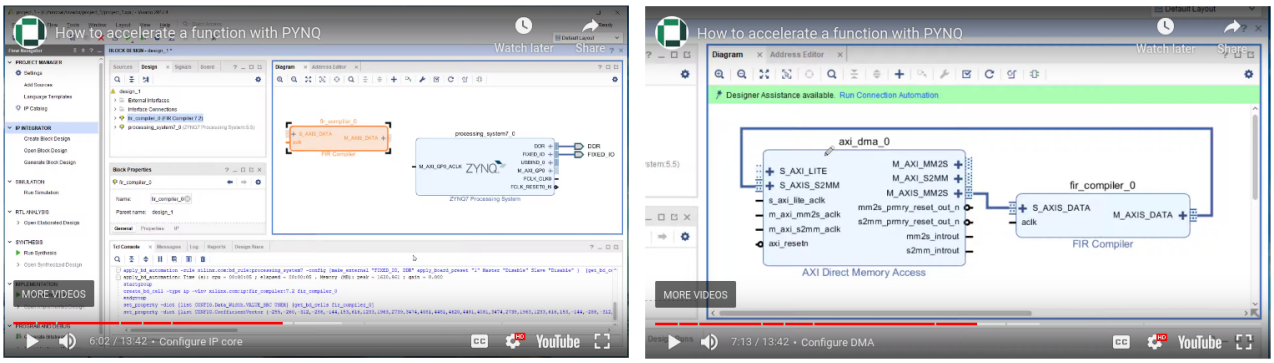

Figure 5: (Left) Adding the FIR filter block. (Right) Connections between DMA and FIR filter.

At the top of the block diagram, click the + button and search for FIR Compiler. Once the FIR block is added to the block diagram, double click it to open its settings. Configure the FIR Compiler in the following ways:

1. Copy the `coeffs` list from the Jupyter notebook to the Coefficient Vector field in the FIR Compiler.
2. Click the Channel Specification tab, and under Hardware Oversampling Specification, set the input sampling frequency to 100 MHz.
3. Just below the sampling frequency, set the clock frequency to 100 MHz. This will set the FIR filter to accept one new sample per clock cycle.
4. Click the Implementation tab, and scroll down to Data Path Options. Set the input and output data widths to be 32 bits, just as they are generated in the Python3 code. Set output rounding mode to "non-symmetric rounding up." Full-precision output is not possible with 32 bit samples as inputs.
5. Under the Interface tab, select Packet Framing for TLAST, and check the Output Ready box. These choices will help transfer the data between the PS and PL layers.
6. To finalize the FIR configuration, click OK. The block diagram should now resemble Fig. 5 (left).

The next IP component we will add is a direct memory access (DMA) block. We need to find the input signal data by address in our Zynq system, pass it to the configured FIR filter, and pass the output back to the Zynq. The DMA will manage this. To add the DMA, click + in the block diagram, and search for DMA. Choose AXI Direct Memory Access, and wait for the new block to appear in our design. Double click on the DMA block to configure it. Perform the following tasks:

1. Disable "scatter gather engine," (not supported in PYNQ-Z1).
2. Set the width of buffer length register to 23 bits. This was the maximum when this project was created. Click OK.

Now connect the proper outputs and inputs between the DMA and the FIR filter, as shown in Fig. 5 (right). Double click the Zynq block, and enable a high-performance AXI port by clicking the green block (bottom right of the diagram).

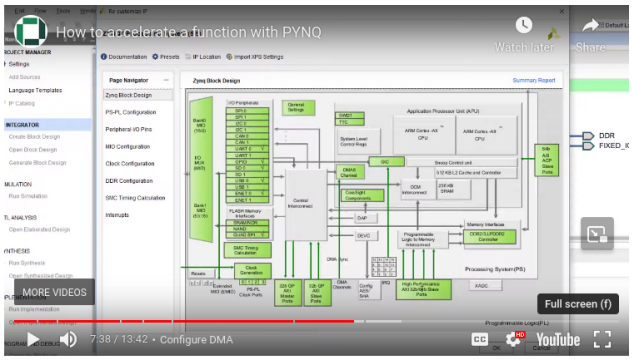

Figure 6: The inner workings of the Zynq SoC.

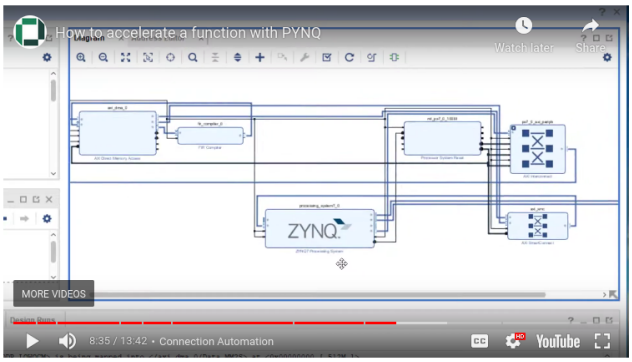

Figure 7: The inner workings of the Zynq SoC.

Under PS-PL Configuration, enable "S AXI HP0 interface." Click OK, and leave the Zynq block selected. Now we can run connection automation via the green bar above the overall block diagram. In the ensuing dialogue, check the boxes for the AXI DMA, FIR Compiler, and processing system connections. Vivado will discern which connections make sense in light of the usual functionality of these blocks. Click OK to run the automation. Run the automation again, and leave the boxes checked. The full block diagram should now resemble Fig. 7. Click the DMA block, then under Block Properties, enter `fir_dma` in the Name field. Similarly, re-name the FIR block `fir`. Select both the DMA and the FIR blocks by clicking one, then holding `Shift` key, and clicking the second. Right-click the pair of blocks and select Create Hierarchy. Name the hierarchy filter. This will make the new PL easier to reference in Python3. Save the block design by clicking the disk icon at the top left of the Vivado window.

4 Creating a Custom PYNQ Overlay

Locate the pane with Design, Sources, Signals, and Board tabs. Click Sources, and under Design Sources, right-click on the design file `design_1`. Select create HDL wrapper, and click OK in the pop-up window.