ECE 5727

Lab 4: Audio Filtering

Due Wednesday, May 5
Total Points: 38

Objectives

In this lab you will:

1. Explore using Vivado IP Integrator (block design) and IP Cores
2. Design and simulate a FIR filter on an audio stream
3. Examine ways the programmable logic and processor are able to interact

Lab Overview

This lab is derived from Digilent's *Zybo Z7 DMA Audio Demo* project (https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-z7-dma-audio-demo/start). DMA (Direct Memory Access) is a way for the programmable logic to quickly read/write data to memory that the processor can directly access, in this case the 1GB DDR memory on the Zybo board. Often, this is to allow the digital logic to manipulate the data before handing it to the processor for higher-level processing, such as running an edge-detection transformation for computer vision.

For the Digilent demo, DMA is used solely for access to a larger memory than is available within the programmable logic directly. This done so that the audio signal can be recorded and stored for playback. The processor does not ever access the audio recording itself. And, for the purposes of this lab, we will be bypassing the DMA portion of the design entirely to simplify and speed up simulation.

The Digilent demo uses the Analog Devices SSM2603 Audio Codec IC on the Zybo-Z7 board for both input and output of the audio signal. The IC has both Line-In and Microphone inputs, and a Headphone output. This lab will still use the Zybo Headphone jack connected to the SSM2603, but will not make use of either input jack. Instead, we will use a Digilent Pmod-Mic3, which is an expansion card that can be plugged into any of the five pmod connectors on the Zybo. This card contains a MEMS microphone connected to a 12-bit ADC that we will use to take periodic readings from over a 3-wire SPI interface. Information on the Pmod-Mic3, including a timing diagram for the SPI interface, can be found at https://reference.digilentinc.com/reference/pmod/pmodmic3/reference-manual.

Instructions

Setup

1. Download the lab files from the course Canvas or GitHub. You will notice that within the folder are two main directories: `hw` and `sw`. The latter contains a few .h and .c files for use by the processor and can be ignored for now. The `hw` directory contains all the files necessary for FPGA design for this lab.

2. Open `hw/Lab4.xpr` in Vivado. There will be some warnings/errors about missing files, etc. That is OK, as we are about to get all that set up!

3. If you open the IP Catalog (`Window > IP Catalog`) you should see that there are two top-level repositories, a Vivado Repository and a User Repository (at `hw/Lab4.ip/repo/`). If the latter is not present, go to Settings, then `IP > Repository` and add it.

4. Now it is time to generate the block design for the project. To begin with, right-click and delete system.bd from under Design Sources. Then, use `Tools > Run TCL Script...` and select `hw/genSystembd.tcl` to run a tcl script that will generate the full system.bd block design. This may take a few minutes while it imports all the relevant IP cores into the design and then configures and connects them together.

5. Once the block design is all set, right-click it under Design Sources and then select `Create HDL Wrapper...`. From the pop-up, make sure the option selected is: *Let Vivado manage wrapper and auto-update*. Click OK. This will create a Verilog wrapper file to use as the top-level file in the FPGA design. *If Vivado doesn't make this the top-level, instead leaving it as a non-design file, try deleting the file and generating it again.*

6. Run Synthesis on the design. Vivado will first run OoC Synthesis on each of the IP cores within the system block design, before running Synthesis on the overall project. This may take upwards of 10 minutes or more depending on your computer.

7. Run Implementation (you don't need to generate a bitstream).

Create an Audio Filter

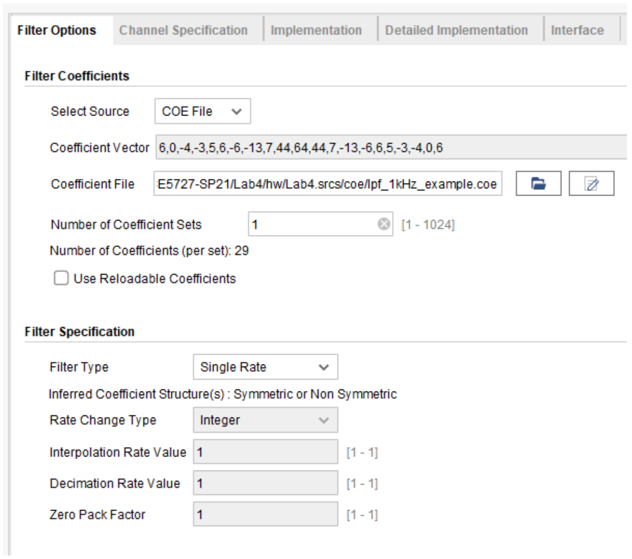
1. With the base design now fully set up within Vivado, it is time to add a finite impulse response filter into the system. If you'd like, design your own low, high, or band-pass filter using MATLAB (or however you'd prefer). Alternatively, I have already designed a 1kHz low-pass filter example that you can use.

   - If you design your own filter, make sure the coefficients are all positive. You can do this by finding how many bits (N) are required for the largest coefficient value and adding 2^N to all the coefficients. For example, if the largest coefficient is 200, then $N = 8$, so add 256 to all coefficients.

- Normally this would not be required, but it keeps things simpler for the purposes of this lab. The microphone readings are all unsigned values, so by keeping the FIR coefficients also unsigned, we avoid having a signed output from the filter that would have to then be converted to an unsigned value before being sent to the SSM2603.

2. Open the block design, then right-click anywhere within the diagram and choose `Add IP...` from the list. Search for the `FIR Compiler` core and drag it into the diagram, or select and hit Enter. Double-click the core to bring up the customization GUI. You will notice there are a few tabs of customization options.

Filter Options



Filter Coefficients

Select Source: COE File

Coefficient Vector: 6,0,-4,-3,5,6,-6,-13,7,44,64,44,7,-13,-6,6,5,-3,-4,0,6

Coefficient File: E5727-SP21/Lab4/hw/Lab4.srcs/coe/lpf_1kHz_example.coe

Number of Coefficient Sets: 1 [1 - 1024]

Number of Coefficients (per set): 29

☐ Use Reloadable Coefficients

Filter Specification

Filter Type: Single Rate

Inferred Coefficient Structure(s) : Symmetric or Non Symmetric

Rate Change Type: Integer

Interpolation Rate Value: 1 [1 - 1]

Decimation Rate Value: 1 [1 - 1]

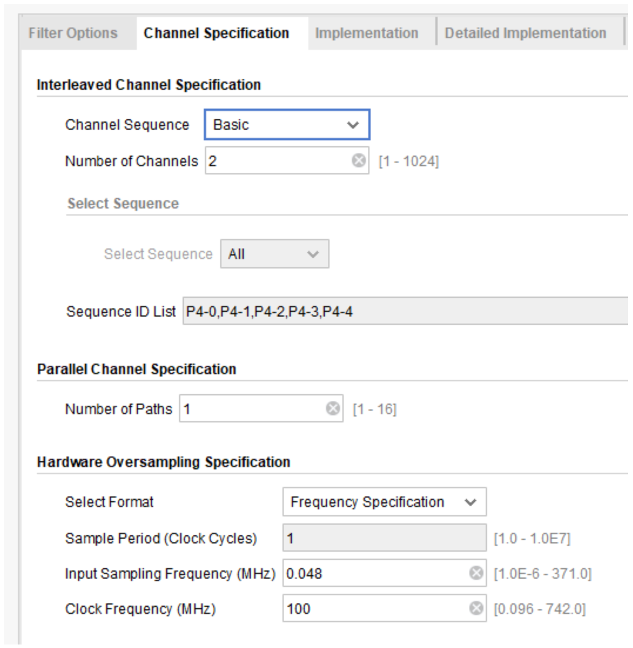
Zero Pack Factor: 1 [1 - 1]

- `Select Source`: If using your own filter, you can select `Vector` and replace the default values under `Coefficient Vector` with your set of filter coefficients. Or, you can select `COE File` and then from `Coefficient File` select either my example filter located at `hw/Lab4.srcs/coe/lpf_1kHz_example.coe` or your own .coe file (use the example file as a reference for how to format your filter coefficients in the file).
- `Number of Coefficient Sets`: Leave this at 1, as we are only going to be using a single filter. However, this number could be increased if you wanted the ability to provide multiple sets of filter coefficients that could then be dynamically switched between at run time.
- `Number of Coefficients (per set)`: This should automatically update to reflect the number of filter coefficients provided (29 if using the example filter).
- `Use Reloadable Coefficients`: If we wanted to dynamically update the filter during runtime (e.g. for an adaptive filter), this would be checked. However,

unless such behavior is desired, it is always better to leave it unchecked as it allows Vivado to perform optimizations to minimize resource usage based on the coefficients provided.

- `Filter Type`: Leave this as the basic `Single Rate`. The other options allow for selection of other more advanced types of FIR filters.

Channel Specification



Interleaved Channel Specification

Channel Sequence: Basic
Number of Channels: 2 [1 - 1024]

Select Sequence

Select Sequence: All
Sequence ID List: P4-0,P4-1,P4-2,P4-3,P4-4

Parallel Channel Specification

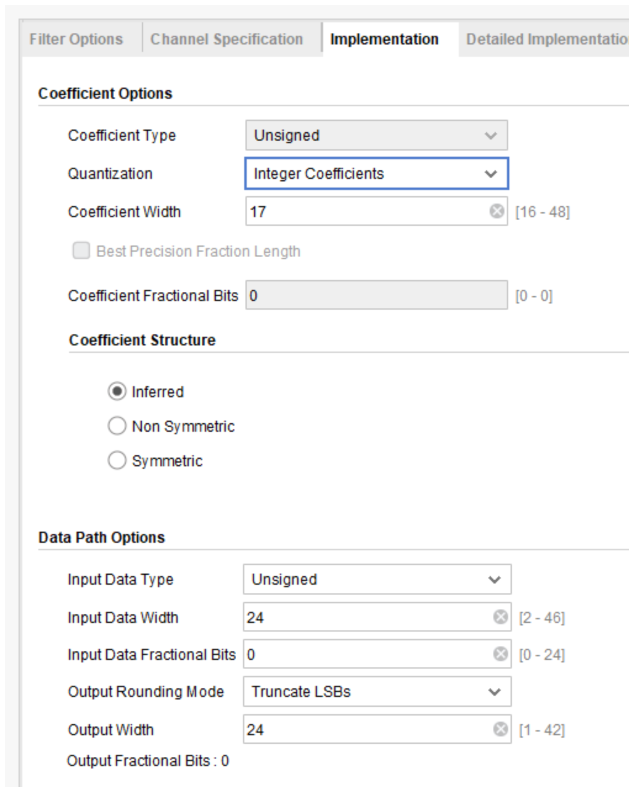
Number of Paths: 1 [1 - 16]

Hardware Oversampling Specification

Select Format: Frequency Specification
Sample Period (Clock Cycles): 1 [1.0 - 1.0E7]
Input Sampling Frequency (MHz): 0.048 [1.0E-6 - 371.0]
Clock Frequency (MHz): 100 [0.096 - 742.0]

- `Channel Sequence`: Leave this as `Basic`.
- `Number of Channels`: Change this to be 2. The PmodMic3 IP core I wrote will sample the microphone a single time and then output the sample twice as the Digilent audio IP core expects stereo - left and right channels - not mono data. As such, the filter needs to know that within the single stream there are 2 interleaved channels of data that it will need to be applied to independently.
- `Input Sampling Frequency (MHz)`: The audio is being sampled at 48kHz, so update this to be 0.048 MHz.
- `Clock Frequency (MHz)`: Update this to 100MHz, which is the frequency of the clock that will be running the FIR filter logic.
- All other items can be left at their default. Combined, all these settings determine how many clock cycles are available for calculation cycle of the filter, which will then be used to determine the number of resources required, or if the design requirements cannot be met.

Implementation



Coefficient Options

Coefficient Type: Unsigned
Quantization: Integer Coefficients
Coefficient Width: 17 [16 - 48]
Best Precision Fraction Length
Coefficient Fractional Bits: 0 [0 - 0]

Coefficient Structure

[✓ Inferred]
Non Symmetric
Symmetric

Data Path Options

Input Data Type: Unsigned
Input Data Width: 24 [2 - 46]
Input Data Fractional Bits: 0 [0 - 24]
Output Rounding Mode: Truncate LSBs
Output Width: 24 [1 - 42]
Output Fractional Bits : 0

- `Coefficient Type`: This should be showing `Unsigned`. If it is not, then you have one or more negative (signed) coefficients in your filter. See my earlier note about making all coefficients positive (unsigned).
- `Quantization`: If you are using my example filter, or your filter coefficients are all integer values, this will read `Integer Coefficients`. If you have decimal values, that will not be an available option. You can leave it as the default (`Quantize Only`), or you can select one of the other options available that trade increased performance relative to theoretical design performance at the expense of greater resource usage.
- `Input Data Type`: Toggle this to `Manual` and then select `Unsigned`.
- `Input Data Width`: The output from the Mic3 core is 32-bits, and the FIR core will automatically detect this. However, the actual mic data is only 24-bit, so we need to manually set this so that the output is scaled correctly.
- `Output Rounding Mode`: To keep things simple, we will opt to `Truncate LSBs`.
- `Output Width`: Set this to 24, which is what the Digilent core is expecting.

Detailed Implementation, Interface

- These two tabs can be left unchanged

3. Before closing the customization GUI, you can find multiple tabs on the left summarizing various aspects of the core as customized. Select the `Freq. Response` one to see the calculated frequency response for the filter you have designed (yes, I know the response for my example one is pretty ugly). Click OK to save your changes to the core and close the GUI.

4. The FIR core now needs to be connected into the design. We will be placing it between the Mic3 (PmodMic3_0) and Audio (d_axi_i2s_audio_0) cores within the block design.

   (a) Find and delete the thick blue line (representing a bus interface connection) connecting the `m_axis` output from Mic3 to the `AXI_MM2S` input to Audio. This was the AXI-Stream interface connection sending the audio data from the microphone to the SSM2603.

   (b) Create an AXI-Stream connection from the microphone to the FIR filter. Hover over the `m_axis` Mic3 output until you see the mouse turn into a little pencil symbol, then click and drag to the `S_AXIS_DATA` input to the FIR core. A green check-mark should appear and a dark line should snap to it. Release to make the connection, and a dark blue line should appear showing the connection has been made.

   (c) Do the same thing to connect the FIR `M_AXIS_DATA` to the Audio `AXI_MM2S`.

   (d) With both interface connections made, a green bar should pop up about *Designer Assistance*. This is because the design is now able to infer what clock should be attached to the core (the `aclk` input). Click `Run Connection Automation` to open a GUI, then click OK to make this happen auto-magically.

5. The filter is now connected into your design. Take it through synthesis (which will run OoC synthesis on the FIR IP core first). You will get two critical warnings about size mismatches in/out of the FIR compiler. This is because the AXI-Stream data channel for the Mic/Audio cores is configured for 32-bits but the FIR filter is only 24-bits.

Simulation

We are now ready to simulate the design to check that the FIR filter is working! The testbench is run from `system_tb.sv`. Within this file, the top-level system module is instantiated, and code is written to simulate the ADC sending data over the SPI interface to the PmodMic3 module within the design.

1. Every time a new sample is requested from the ADC by the design, a new value for the audio signal is calculated. This signal is a 50/50 mix of a high-frequency and a low-frequency sine wave, tunable by changing the `HF_FREQ` and `LF_FREQ` local parameters respectively. For my example filter, since it is a 1kHz low pass filter, I set the low frequency to 500Hz, and the high frequency to 5kHz. This means the output of my FIR filter should show only a 500Hz signal in simulation. If you designed your own filter, tune the parameters as needed.

2. Run a behavioral simulation. It will take a few minutes for everything to compile and elaborate. Once the simulation is open, set up the waveform to window as follows:

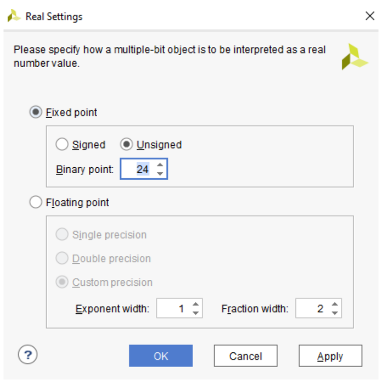
   - Remove all the top-level signals from the waveform, except for `mic3_*`. You can also leave any of the ADC related signals if you'd like to see your input signals; I'd recommend at the very least leaving in the `audioSignal`.
   - Under the `Scope` tab, select `DUT > system_i > fir_compiler_0`. Then from the `Objects` tab drag in `M_AXIS_DATA` and `S_AXIS_DATA` into the waveform. These are the two AXI-Streams in/out of the FIR filter. Toggle both of them open within the waveform to see all the signals available within the interface to be monitored.

3. Into the `Tcl Console` type the following command: `run 1 ms`
   You can run for a shorter amount of time than 1ms, but you may end up without enough runtime to get a good look at the filter input/output sine waves

4. Sit back and wait for the simulation to run. It will take a while (around 10 minutes for me).

5. Once simulation is finished, it is time to see if the filter is working. Select the `.TDATA[23:0]` signals for both AXI-Stream interfaces.

   - Right-click and select `Radix > Real Settings...` and then configure it to be Fixed point, Unsigned with Binary point 24.


Real Settings

Please specify how a multiple-bit object is to be interpreted as a real number value.

[✓ Fixed point]

Signed [✓ Unsigned]

Binary point: 24

Floating point

Single precision

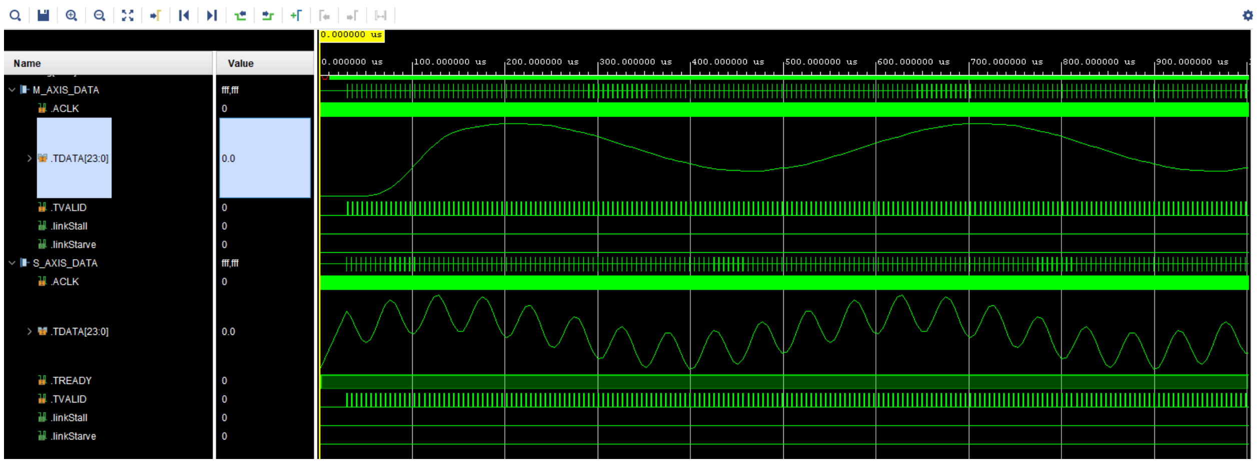
Double precision

[✓ Custom precision]

Exponent width: 1 Fraction width: 2

   - Right-click again and select `Waveform Style > Analog`.
   - If you included the `audioSignal` in your waveform, do the same thing but with 12 as the Binary point.

6. You should now see two sine waves - the `S_AXIS` one will be the input audio wave (and look like the `audioSignal`) and the `M_AXIS` one is the filter output. *Take a screenshot of your filter working (5 points)*



Example Waveform: "noisy" bottom signal is filtered to create "clean" top signal

Lab Questions

1. (*5 points*) Take a look at the `mic_*` (SPI interface) signals in the simulation. What edge of the clock is data changing on? Why do you think this is?

2. Take a look at *PmodMic3_wrapper.sv* and *PmodMic3.sv*. You can find them in: `Lab4.ip/repo/pmod_mic3/Pmod_Mic3.srcs/**/`.

   (a) (*5 points*) There are three `pmod*Reg` registers declared at line 78 within the wrapper module. These are clocked by the 100MHz `s_axi_l_aclk`. The PmodMic3 module logic is clocked by a 12.288 MHz clock (`pmodClk`). Why can't this logic use the data stored in those three registers directly? What is being done to handle this problem?

   (b) (*5 points*) At lines 99 and 105 in the testbench we are forcing the pmodSampleReg and pmodControlReg to specific values. Why? What does this do to the PmodMic3 logic? (What happens?)

3. Take a look at the software source files: `audio.{c,h}`

   (a) (*3 points*) In what function(s) are we setting the registers to the values the testbench is (from 2b).

   (b) (*3 points*) What Xilinx function call is being used to set those registers? What do you think that function call is triggering?

   (c) The pmod registers all have memory addresses that are defined using an enum in `audio.h`. How are those addresses determined? In other words:

      i. (*3 points*) What is being determined at the block design level?

      ii. (*3 points*) What is being determined at the module (PmodMic3 wrapper) level?

      iii. (*3 points*) If a new register needed to be added what might need to change? What if thousands of registers were to be added?

(d) Looking at the function `fnAudioWriteToReg`:

   i. (*3 points*) What is this function achieving?

   ii. (*3 points*) What Xilinx function call is being used here? How does it differ from the function in 3(b)? (What interface is being used here)