525.742 SOC FPGA Design Lab

Laboratory Project 4A

FIR Filter

**Introduction**

In Lab 3, you started the signal path for the SDR, which took in raw A/D data (simulated by a DDS) and forwarded it directly to the audio output, decimating by a factor of 2560 in the process.  While using that lab, you became familiar with the aliasing that occurs when you change the sample rate from 125MHz to 48kHz.  In Lab 4 (A + B) we will extend that infrastructure towards the development of a software defined radio (SDR) on our evaluation board. To this end, we will implement the most computationally intensive section of the signal processing chain, the channel selection filter. You will be tasked with designing and implementing a filter chain that eliminates all frequencies that will alias into our band when we downsample to 48kHz

Our filter chain should:

* accept 16-bit signed input data at a 125MHz input sample rate
  + For the purposes of this design, we can assume that the input amplitude will never go above 30000 counts.
* pass all frequencies below 18kHz relatively unmodified in amplitude (ripple max .5dB)
* attenuate all frequencies above 30kHz by 80dB.
* Perform the decimation, providing an output sample rate of 125MHz/2560, or approximately 48.8kHz.
* bit-growth and rounding should be accounted for such that we have a unity gain filter (In other words, if you put a signal (say 10kHz) into the filter, the amplitude of that signal when it comes out of the filter should be approximately equal to what it was when it went in)

This filter will ultimately (in later labs) be inserted into our signal processing chain directly after a mixer which is used to mix an arbitrary channel down to DC.  However, for this laboratory exercise, we will test the filter via our familiar methods, sending data to it from our fake-A/D converter (our trusty DDS) and sending the output to the DAC interface for playback

**Goals (Lab 4A)**

1. design in Matlab the filter chain which will serve as a channel selection filter for our radio
2. Experiment with the Xilinx provided FIR Compiler “Bit Accurate Model” to come up with the proper settings for the FIR Compiler core
3. using the FIR Compiler bit accurate model, measure the actual performance characteristics of the soon to be implemented filter.
4. When finished with this exercise, be ready to put a filter into our hardware design and have it work the first time! (We will actually do this in Part B of this lab)

**Before You Start**

As you have no doubt already figured out, VHDL simulation is not the ideal place to be precisely analyzing signals and experimenting with core settings. Modelling of signal processing is much faster (and visualization is much easier) in a tool like Matlab. Most signal processing systems will be modelled in Matlab before they ever make their way into an FPGA. For this reason, for many of the signal processing cores, Xilinx provides a “bit accurate model” of the core which will run in C/Matlab. The goal of these models is to let you see exactly how the core will perform. All of the configuration options are modelled, and you can see the results of every configuration change quickly. For this lab we will be making use of the bit accurate model for the FIR compiler. When you create the IP, the C model is added to your project in a subdirectory under the core. For your convenience, it is also located in the module in Canvas. Grab the zip file appropriate for your OS. There is a windows version and linux version to choose from. Complete the installation of the model and return here once you can run the sample code. (See the Appendix for some installation instructions… these are basically identical to the DDS bit-accurate model)

**Implementation**

Designing the Filter

The Xilinx FIR Filter Core will implement a FIR filter with at most 2048 coefficients.  This is not enough to perform the task requested of you in this lab.  To meet the desired requirements, you will generate 2 separate filters, and chain them together such that data will go through Filter 1 first, and the filtered data from filter 1 will be sent to filter 2.

*Filter 1*:

Input Sample Rate = 125MHz

Fpass = 576.000kHz

Fstop = 2.549MHz

Astop >= 80dB

Apass <= 0.5dB

Decimate by 40

The output of this filter will then be data, sampled at 3.125MHz, containing frequencies at 576kHz and below. Our next job will be to limit that bandwidth to 18kHz and then cut the sample rate even further.

*Filter 2:*

Input Sample Rate = 3.125MHz

Fpass = 18kHz

Fstop = 30kHz

Astop >= 80dB

Apass <= 0.5dB

Decimate by 64

The output of this filter will then be data, sampled at 48k, containing frequencies at 18kHz and below.  Any signal that would have aliased into that 18kHz band will be attenuated by at least 80dB.

**The deliverable for this lab is a Matlab script(s) that will do the required analysis, and a Word document which summarizes the results and answers some of the questions posed in the lab. The instructions below will lead you through the development of this script, and requirements will be highlighted along the way. A checklist for the requirements all in one place can be found in the rubric**

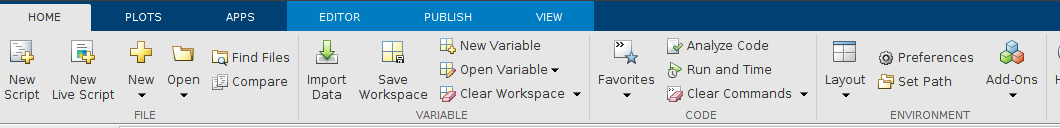
1. Start a new Matlab script, named “lab4a\_full\_simulation.m”. The first thing that should be placed in the script is generation of coefficients for each of the two filters described above. A good way to do this easily is to use FDAtool as shown in class, and then use the “File->Generate Matlab Code-> Filter Design Function”. This will create code to make the filter coefficients which you can copy or call.
2. Your script, when run, should also write a “.coe” file for each filter in the format shown in class. Code in the lecture slides shows how this can be done. You may use floating point format or scaled integers, your choice.
3. Using the bit-accurate model, generate a filter object for each of the two filters. For each, you will want to modify the configuration structure to model the settings that you want, including but not limited to adding your coefficients.
4. Using matlab, you will be testing your cascaded filter chain with a couple of different inputs, and using those tests to validate the performance of your filter (and of course change the settings of the filter core until it behaves the way you like).
   1. Basic Functionality : create an input signal with a tone in the passband at 2kHz. Run that signal through your filter and validate that it passes the filter chain. Your input signal should be long enough to make an output signal with a length of at least 1024 samples. Plot the results, paying attention to the input amplitude vs output amplitude. Using the data sheet for the core, and the lecture slides, experiment with the settings of the filter cores until you have attained unity gain. (Note : don’t just multiply your output by a scaling factor at the end, though division by a power of 2 could be used if you desire – since that is easily implemented in hardware). Plot both the input and the output data for this final setup. Label the plot : “Reponse to signal in passband”
   2. Qualitative Observation of Filtering: add another tone (30kHz) to your input signal, so that now your input signal is the sum of two sinusoids, one in the pass-band and one in the stop-band. The 30kHz tone should be of equal amplitude or greater to the 2kHz tone. Note that you don’t want the total input signal amplitude to go beyond the input range of the filter, so keep it below 30000. Plot both the input and output data for this case and label the plot : “Response to two-tone input”
   3. Observation of Transition Band: It is likely that the result from part B showed that the signal in the stopband was essentially “gone”. 80dB is a lot of attenuation, and so it will be difficult to even observe the presence of the 30kHz signal in your output. For this next experiment, we are going to change the frequency content of the input signal to put something in the “transition band” of the filter, namely higher than the passband, but before it gets to the stopband. Pick a frequency in this range (suggested: 25kHz) and make your input signal be the sum of two frequencies again, the 2kHz tone and this new 25kHz tone, both at equal amplitude. Run this through your filter and plot the input and output data like before, labelling the plot: “transition band experiment”.
   4. Quantitative Spot Check of Accuracy: As we saw in class, the performance of the filter as designed in Matlab may be different from what ends up in the FPGA due to fixed-point effects like coefficient quantization. This model takes those effects into account, so it is the ideal place to check that we are precisely getting what we designed. The results from the experiment in step C can be used for this purpose. Use matlab to measure the power spectrum of the output (you can use a variety of methods, a convenient one is “periodogram(dout\_f2,[],'onesided',512,48.828e3)”. In dB, how much lower is the signal in your transition band vs the signal in the passband. (If these started at equal amplitude, this is an easier experiment). Plot the resultant spectrum. What was the attenuation of the signal at the frequency you chose? Did it match what was expected from the original design?  *(Hint: if you no longer have the fdatool open, you can visualize your filter by typing “freqz(H2,512,3.125e6)” Where H2 is the second filter in the chain, and 3.125e6 is the input sample rate of that filter. There you can zoom in and find the frequency in question and what it’s attenuation should have been)*
5. *Documentation: At this point, you have done all the required simulation in order to smooth the way for an easy implementation of this filter in the FPGA. You should start that assignment as early as possible, but first, let’s collect some documentation to submit for posterity. Create a word document which includes all of the highlighted plots, and any comments where necessary (to explain any results that were not expected). Make sure to answer the question in step 4d – which asks you to compare the transition band attenuation with what was expected from the original design of the filter. Submit this word document, as well as a zip of your complete matlab in a state that can be easily run. (you need not include the bit accurate model itself as part of your submission, but you certainly may). Please make sure the word document is not in your ZIP file*
6. *+10 pts Extra Credit: In class, we talked about how different numbers of bits used to represent the coefficients could affect the performance of the filter. Scaling the coefficients to use more bits (more precision) made the end-result of the FPGA fixed point implementation more closely match that of what you originally designed. Illustrate this by doing the same experiments where your second filter uses only 3 bits used for coefficients. Show either the “Transition Band Experiment” plot or the “Response to Two-Tone Input” plot and compare it to the one you made originally*

**APPENDIX : Installing and running the bit accurate model (windows instructions)**

**(you probably don’t need this instructions, because it works just like the DDS bit accurate model)**

1. Unzip the zip file to a convenient location, this will become your work directory for this project.
2. Once you have the model unzipped, open MATLAB and navigate to the directory you just created.
3. Execute the function make\_fir\_compiler\_v7\_2\_mex.m

If everything builds, you are lucky; however, it is likely if you’ve never compiled C to matlab before that you will get an error saying “no compiler found”. No problem, getting a compiler is free. Go to the Home bar in matlab and click Add Ons, and select Get Add Ons.



Search for MinGW, and select it. Click on the Add button, and then select Add To Matlab. Installation should progress. When it is complete, execute the function in Step 3 again.

1. Once this completes succesfully, it won’t need to be done again. From this point forward, you can execute the model with simple API calls. To test that the model is present and working, just type in the command window : “my\_filter = fir\_compiler\_v7\_2\_bitacc()” this creates a default filter object and prints out the values of all of the parameters. Notice that these are all the parameters that you see in the gui when configuring a fir filter core.