ECE4703 B Term 2008 Laboratory Assignment 2

Project Code and Report Due by 3:00pm 13-Nov-2008

The goals of this laboratory assignment are:

- to familiarize you with the digital filter design tools in Matlab,
- to demonstrate the differences between floating-point and fixed-point processing,
- to develop an understanding of filter coefficient quantization, and
- to familiarize you with real-time finite impulse response (FIR) filtering on the TMS320C6713

This assignment considers real-time DSP implementation of FIR filtering as described by the convolution operation

$$y[n] = \sum_{k=0}^{M-1} b[k]x[n-k]$$

where $y[n]$ is the current filter output, $b[0], \dots, b[M-1]$ are the filter coefficients, and $x[n], \dots, x[n-M+1]$ is a buffer containing the current and $M-1$ previous inputs.

While the code you write for this assignment will allow you to realize any type of FIR filter, you will be evaluated on your ability to realize a filter that satisfies the following requirements:

- Sampling rate 32kHz.
- Least-squares design method.
- Filter order of 40.
- Frequency edges: fstop1 = 4kHz, fpass1 = 4.5kHz, fpass2 = 5.5kHz, fstop2 = 6kHz
- Band weights: Wstop1 = 1, Wpass = 1, Wstop2 = 1

Use the Matlab filter design tools, e.g. `fdatool`, to design an FIR bandpass filter that satisfies the requirements. Plot the impulse response, magnitude response, and phase response of your filter. You can export the coefficients to the workspace (note that there will be 41 coefficients for a filter of order 40) and you can save these coefficients to a .mat file using the `save` command. Type `help save` for details on how to save just the coefficients. You can also export your filter coefficients to a C header file from `fdatool`.

1 Part 1: FIR filtering in Matlab

With the design of the filter complete, we will now simulate the FIR filter in Matlab to ensure that it is working correctly and to also measure certain internal characteristics of the filter that will be useful for when we realize the filter in fixed-point and on the DSK.

1.1 Part 1a: Floating Point Implementation of an FIR filter

For the floating-point implementation of an FIR filter, we will assume that the filter coefficients $b[0], \dots, b[M-1]$ and all intermediate calculations (any results of multiplies and sums) are represented as 64-bit double-precision floating numbers. Note that this is the normal datatype for Matlab variables; you do not need to do anything special here for the internal calculations of the filter. Only the input and output of the FIR filter need to be quantized. The inputs to the filter are assumed to be 16-bit signed integer (short datatype) samples from the AIC23 ADC. The outputs of the filter must also be cast to 16-bit signed integers for proper interpretation by the AIC23 DAC. Figure 1 summarizes the operation of the floating-point FIR filter.

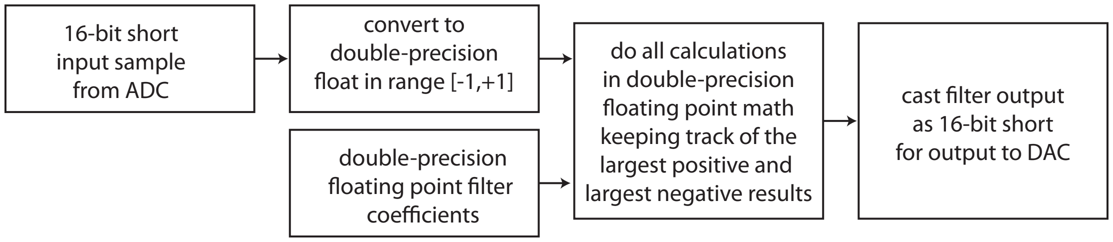
16-bit short input sample from ADC
convert to double-precision float in range [-1,+1]
double-precision floating point filter coefficients
do all calculations in double-precision floating point math keeping track of the largest positive and largest negative results
cast filter output as 16-bit short for output to DAC

Figure 1: Floating point implementation of an FIR filter.

Some important details:

- Confirm that your input/output quantization code works correctly by testing it with a small number of bits, e.g. $N = 3$. Your code should perform both saturation/clipping and quantization as discussed in lecture.
- Do not use the Matlab commands `filter` or `conv` to compute the output of your filter for each input sample. Instead, compute the convolution manually with `for` loops (or `while` loops if you prefer). You can certainly use the Matlab commands `filter` or `conv` to verify that your code is working correctly, but it is important that you perform the FIR filtering operation manually, as if it were written in C and running on the DSK.
- Note that you will have at least two `for/while` loops in your code: a `for/while` loop corresponding to the arrival of new samples from the codec and another `for/while` loop for the computation of the FIR filter output.
- For each multiply and each addition, you should keep track of the largest positive and largest negative valued intermediate results (including the final output) in the filter. While no overflow will occur in the double-precision floating-point intermediate results of your FIR filter, *you can use this information to determine if overflow will occur at the output of the filter.* If you are getting overflow at the filter output, you can scale the input, the filter coefficients,

or the final floating-point result prior to casting it as a 16-bit signed integer. This largest positive and largest negative valued intermediate result information will also be very useful to avoid overflow and select proper scaling factors for the intermediate results when converting to a fixed-point implementation of the FIR filter.

- You may want to initialize the input buffer samples to zero.
- You do not need to write any special functions to perform floating-point math. The DSK already has functions that do this for both single-precision and double-precision floating-point numbers. Matlab's normal datatype is double-precision floating-point, so all calculations will automatically be in double-precision floating-point unless you force Matlab to do something different.

To test that your filter is producing the correct frequency response, you can generate a white noise input signal with the following commands in Matlab:

```
fs = 32000; % sampling frequency
T = 10; % duration
x = 2*rand(fs*T,1)-1; % zero-mean white noise
x = x*0.95; % reduce the amplitude of x to 95% of ADC full scale
```

Note that the variable x is stored in Matlab as an array of double-precision floating-point numbers. Since the AIC23 ADC produces signed 16-bit samples, you will need to quantize x appropriately. If you've done this correctly, the quantized x will be in $Q15$ format with largest positive and largest negative values approximately equal to ± 0.95. Note that this quantized vector will be stored in Matlab as a double-precision float. Hence, you don't need to do anything to cast the input samples to floating-point variables.

Your FIR filter coefficients should automatically have been generated as double-precision floating-point variables. Perform the FIR filter calculations one multiply and one addition at a time, keeping track of the largest positive and largest negative results of each multiply and each addition. Quantize the final calculation in $Q15$ format (checking for overflow and handling it appropriately) and store the result in a vector of FIR filter outputs.

To plot the spectrum of the input and the output of your filter, you can use the following Matlab code:

```
[Px,f] = pwelch(x_short,1024,512,1024,fs); % estimate spectrum of input
[Py,f] = pwelch(y_short,1024,512,1024,fs); % estimate spectrum of output
plot(f/1e3,10*log10(Px),f/1e3,10*log10(Py)); % plot spectra in dB
grid on
xlabel('frequency (kHz)');
ylabel('magnitude response (dB)');
```

You should type `help pwelch` in Matlab to learn more about what the `pwelch` command does. The absolute dB values of the input and output spectra probably won't make much sense, but the relative values between the input and output should agree with the theoretical predictions from the Matlab filter design tools. Try to plot the theoretical predicted magnitude response on top of your simulated filter response to see how well they match (you may need to apply an offset to get things to line up).

Please make sure your floating-point FIR filter implementation is working correctly and that you are able to determine the largest positive and largest negative intermediate results before proceeding to the fixed-point FIR filter implementation.

1.2 Part 1b: Fixed Point Implementation of an FIR filter

For the fixed-point implementation of an FIR filter, we will now use filter coefficients $b[0], \ldots, b[M-1]$ that are quantized as short datatypes (16-bit signed integers). You should determine the best Q-representation that will allow you to represent your filter coefficients with minimum quantization error while also avoiding overflow. Note that the optimum number of fractional bits might be a negative number. This is ok — it just means that you are shifting the decimal point to the right rather than to the left.

Now, rewrite your FIR filter code to store all input samples as 16-bit integers (not double precision floating-point) and to store all intermediate calculations (the results of multiplies and sums) as 32-bit signed integer variables. How many fractional bits should your 32-bit intermediate results have?

Figure 2 summarizes the operation of the fixed-point filter. Since you will be multiplying 16-bit signed integer input samples with 16-bit signed integer FIR filter coefficients and storing the result in a 32-bit signed integer variable, you should not need to worry about overflow after multiplication. You do need to consider the possibility of overflow, however, in addition operations since the sum of two 32-bit signed integers may be too large to fit into a 32-bit signed integer datatype. This may influence your decision on how many fractional bits your 32-bit intermediate results should have.

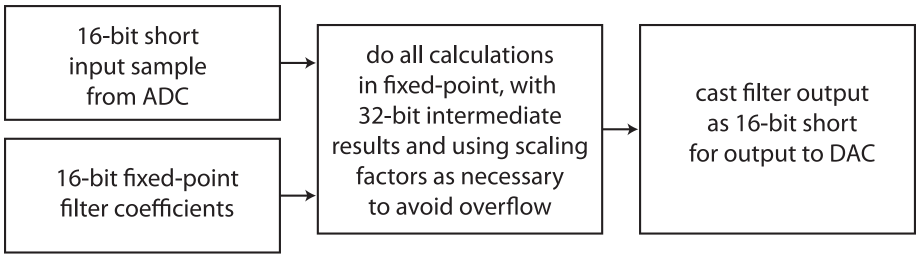
16-bit short input sample from ADC
16-bit fixed-point filter coefficients
do all calculations in fixed-point, with 32-bit intermediate results and using scaling factors as necessary to avoid overflow
cast filter output as 16-bit short for output to DAC

Figure 2: Fixed-point implementation of an FIR filter.

To be clear, all of the variables in your Matlab code will still be double-precision floating-point (Matlab does allow you to declare variables with integer datatypes, but the behavior of these datatypes in Matlab is different than on the C6713), but you will perform all of your math as if the variables were integer datatypes. This means that you must check for overflow manually *after every fixed-point calculation.* The following function emulates the behavior of the C6713 DSP and also provides an indication of overflow that you can use to troubleshoot your fixed-point FIR filter code:

```
% function [y,i]=overflow(x,N)
% x is the theoretical result of an integer calculation (should be an integer)
% N is the number of bits (assumes one bit for the sign)
% y is the actual result of the calculation on the DSP
% i=1 if overflow occured

function [y,i]=overflow(x,N)

y = x;
i = 0;
```

```
% check most positive value
while y>(2^(N-1)-1),
   y = y-2^N;
   i = 1;
end

% check most negative value
while y<-2^(N-1),
   y = y+2^N;
   i = 1;
end
```

For example, if you type `[u,i] = overflow(12345+23456,16);` you get the answer `u = -29735` and `i = 1`. This is exactly how overflow occurs on the C6713 (try it).

Refer to Chapter 6 of your Kehtarnavaz textbook for a discussion of intermediate result scaling and output scaling that may be necessary to get your filter working correctly. How should you scale your final 32-bit intermediate result to produce the 16-bit output? Can you use any of the “largest positive” and “largest negative” results from your floating-point FIR filter to determine appropriate scaling factors? Using the results you obtained from your floating-point FIR filter, Chapter 6 of your textbook, and the principles covered in lecture, you should be able to perform your fixed-point FIR filter processing without overflow and get a fixed-point filter frequency response almost identical to the response you obtained in the floating-point case.

2 Part 2: FIR filtering on the DSK

In the remainder of this assignment, you will be implementing your FIR filter in real-time on the TMS320C6713 DSK by writing code in C to compute the filter output. You should be using the same filter coefficients as in Part 1, but you may find it more convenient to export these coefficients to a C header file for inclusion in your CCS projects.

2.1 Part 2a: Floating-Point Implementation of an FIR filter on the DSK

Convert your Matlab code from Part 1a to a C6713 project that runs in real-time on the DSK at a sampling frequency of 32kHz. A suggested outline for your program is given below (you are welcome to any approach that makes sense):

1. Declare variables including a double-precision floating point buffer for the input samples. Alternatively, you could declare the input sample buffer as short, and then cast these shorts to double-precision floats each time you multiply the input samples with the filter coefficients. Feel free to see which approach runs faster.

2. Initialize the DSK, codec, set the sampling rate to 32kHz, set up interrupts, etc.

3. In the ISR:

   (a) Read in the input sample from the right channel of the AIC23 codec. Because we are reading the channels in stereo, you will also get the left channel too but you don’t need to use it.

(b) Put the input sample into the input buffer.

(c) Compute the filter output using the floating-point filter coefficients and the input buffer (either stored as floating point or dynamically cast as floating point during the calculations). See the Kehtarnavaz textbook for examples if you are sure how to do this.

(d) Cast the floating-point final result to a short for output to the DAC. You may want to check for overflow here and perhaps even light up an LED if overflow is detected.

(e) Write the short filter output to the codec (right channel).

Some good ideas (not required): You may want to also write the current input sample to the left channel so you can easily see the input and output simultaneously on the scope. Buffering the filter output might also be handy for troubleshooting and/or visualization.

All intermediate results in this part of the assignment should be *double-precision floating-point numbers*. Convert the final result to a 16-bit signed integer (short datatype) only prior to output by the AIC23 DAC. To see that your filter is working correctly, generate a white noise signal in Matlab as discussed earlier and play this white noise signal out of the computer's sound card while recording the filter output of the DSK to a .wav file. You can open this .wav file in Matlab and use the `pwelch` command as described earlier to plot the magnitude response of your filter. Make sure the white-noise signal from the sound card is scaled to almost the entire full range of the AIC23 ADC (90% to 95% is ideal). The magnitude response of your real-time FIR filter should be indistinguishable from the Matlab simulation in Part 1a. Profile your code to determine the number of cycles needed to complete the ISR. Does your code run in real-time?

2.2 Part 2b: Fixed-Point Implementation of an FIR filter on the DSK

Convert your Matlab code from Part 1b to a C6713 project that runs in real-time on the DSK at a sampling frequency of 32kHz. *You are not permitted to use any floating-point variables in this part of the assignment.* Use the same steps as Part 2a to determine if your filter is working correctly and that it is running in real-time.

3 In Lab

You will be working in the same teams as in Lab 1.

4 Specific Items to Discuss in Your Report

In your report, provide theoretical and experimental magnitude response results in the same plot (use different colors or line styles) for each part of the assignment. Be sure to explain any discrepancies. Discuss all scaling considerations that you used to get your filters working correctly and to avoid overflow in the intermediate results and the output.

Profile the execution time of your final double-precision floating-point FIR filter and your final fixed-point FIR filter. Discuss any insight obtained from the profiling results.

Also, in the fixed-point implementation of the FIR filter on the DSK, purposely set an intermediate result scaling factor or the output scaling factor to an incorrect value to cause occasional overflow. Listen to the filter output and describe the sound of overflow. Plot the magnitude response of the "bad" filter output to a white noise input using `pwelch`. Comment on the effect of overflow on the magnitude response of the filter output.