ECE4703 B Term 2012 Laboratory Assignment 5

Signoff 2-5pm 05-Dec-2012 // Report due by 3pm 06-Dec-2012

The goals of this laboratory assignment are:

- to develop an understanding of *frame-based* digital signal processing,
- to familiarize you with computationally efficient techniques that can reduce computational through *algorithm optimization*,
- to allow you to apply the FFT to a practical signal processing problem.

1 Problem Statement

In this assignment, you will implement a filter in real-time using a frame-by-frame frequency domain approach. Specifically, given the FFT-length parameter `nfft` (an integer power of two), you will

1. window the incoming signal with a periodic Hanning window of length `nfft`,
2. compute the length `nfft` FFT of this windowed signal,
3. process the FFT output by multiplying it with frequency response of your filter (note this multiplication will be complex),
4. compute the length `nfft` IFFT of the processed signal, and
5. shift and add the result to the output buffer.

All math in this assignment should be performed in single-precision floating point and all of your code will be written in C. The sampling rate for all parts of this assignment should be set to $f_s = 16$kHz.

As discussed in lecture, all frame-based processing should be performed by a function called from `main()`. The ISR should only fill/maintain buffers and set flags to indicate a new buffer is ready for processing. To run in real-time, you need to complete all of your computations before the next incoming buffer is full.

2 Part 1: STFT Analysis and Synthesis Framework

Prior to implementing your frequency-domain filter, you should confirm that you are able to successfully perform frame-by-frame processing and short-time Fourier transform (STFT) analysis and synthesis. The goal in this step is to have an output signal that closely matches the input signal on the DSK. Matlab code that performs the analysis and synthesis functions is given below:

```
%% analysis
i1 = 1;
for n=1:nframes,
    x(n,:) = y(i1:i1+nfft-1).'; % get contents of buffer
    X(n,:) = fft(x(n,:).*w);    % window and compute FFT
    i1 = i1+nfft-noverlap;      % move buffer pointer
end

Xp = X;                         % no filtering (for now)

%% synthesis
i1 = 1;
for n=1:nframes,
    u = ifft(Xp(n,:));          % compute IFFT (result should be real)
    z(i1:i1+nfft-1) = z(i1:i1+nfft-1) + real(u);  % add to output buffer
    i1 = i1+nfft-noverlap;      % move buffer pointer
end
```

The parameter `noverlap` specifies the number of overlapping samples in each window and is typically set to `3*nfft/4` or `nfft/2`. A diagram of the overlapping input buffer processing is shown in Figure 1.

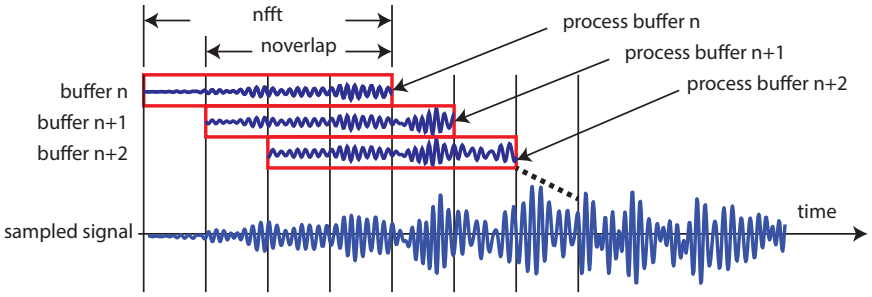

Figure 1: Overlapped input buffer processing.

In a real-time implementation, you will need to process a length `nfft` input buffer every `nfft-noverlap` samples.

2.1 ISR Functionality

Unlike sample-by-sample processing, there is very little ISR functionality of a frame-by-frame processing algorithm. Your ISR should simply read samples from the codec (casting to float), store the samples in an appropriate input buffer, set a flag/index when an input buffer is ready for processing, and write samples to the codec from the output buffer, casting to short if necessary. Details on the output buffer are provided in Section 2.5. When a new input buffer becomes available, your main code should perform the time-domain windowing operation described in the following section.

2.2 Time-Domain Windowing

When a buffer becomes available for processing, the first step is to apply the time-domain window function as shown in Figure 2.

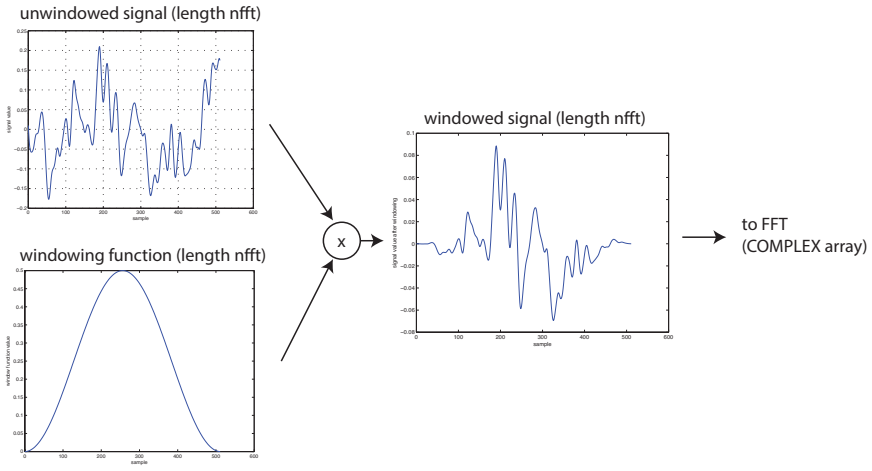

Figure 2: Applying a time-domain window prior to computation of the FFT.

In Matlab, you should pre-compute (and optionally scale) periodic Hanning windows at varying values of `nfft`, e.g.

```
w = hanning(nfft,'periodic')';
```

and store these window functions as static variables in your C code. Computation of the buffer/window product will require `nfft` multiplications and no additions. The result of the buffer/window product should be stored in a COMPLEX array for subsequent computation of the FFT. Even though buffer/window product is real-valued, the TI FFT code you will use requires the input signal to be formatted as COMPLEX, which we define as

```
// complex typedef
typedef struct {
  float re,im;
} COMPLEX;
```

Computation of the FFT using TI's optimized FFT function is discussed in the next section.

2.3 Computing the FFT

Due to the wide variety of applications for the FFT, TI provides an optimized linear assembly function to implement FFTs on the C6x. You will need three functions to use TI's optimized FFT routines. These functions are `cfftr2_dit`, `digitrev_index`, and `bitrev`. The typical usage is

```
digitrev_index(iw,N/RADIX,RADIX); //produces index for bitrev() W
bitrev(w,iw,N/RADIX);      //bit reverse W
cfftr2_dit(x,w,N) ;      //TI floating-pt complex FFT
digitrev_index(ix, N, RADIX);     //produces index for bitrev() X
```

```
bitrev(x,ix,N);      //freq scrambled->bit-reverse X
```

Note that the first two lines only need to be run once in order to get the twiddle factors in the bit-reversed order expected by the TI FFT function. It is probably a good idea to take care of that in the initialization part of your code.

You should read the header comments in the provided FFT files to make sure that you know how to use them correctly. The FFT function is called like this:

```
void cfftr2_dit( float *x, const float *w, short N)

// x  Pointer to Array of Dimension 2*N elements holding
//    Input to and Outputs from the function
// w  Pointer to an array holding the complex twiddle factors
// N  Number of complex points in z
```

The input to the FFT is in the array x and the result of the FFT is returned in the array x (the input is overwritten by the FFT function). You should compute the sin/cos portions of the "twiddle factors" during the initialization part of your code (prior to the FFT function call) and store them in W. You will probably want to include the header file `math.h` to allow for pre-computation of the sin and cos terms needed in for the complex "twiddle-factors". All input/output arrays and twiddle factors should be single-precision floating point datatypes.

You should confirm that this part of your code is working correctly before proceeding. One way to do this is just provide some known input to the FFT function and compare the output to Matlab.

2.4 Computing the IFFT

In this part of the code, since we are not actually processing the signal yet, you will now perform an IFFT directly on the output of the FFT. Refer to the comments in TI's FFT function to determine how to perform an IFFT using the same function. After the IFFT has been performed, the desired result will be in the real elements of the output buffer. The imaginary elements of the output buffer should all be zero or very close to zero. If they aren't, then you have a bug in your code somewhere or you aren't calling the FFT function correctly.

After computing the IFFT, you will now shift and add these results to the output buffer as described in the following section.

2.5 Shifting and Adding to the Output Buffer

The overlapped nature of the input buffers requires us to similarly overlap the outputs of the IFFT. This is conceptually illustrated in Figure 3.

Your code will need to maintain a global output buffer (and probably an output buffer index) that is used by the ISR when input sample interrupts are generated. The output buffer must have at least `nfft` samples.

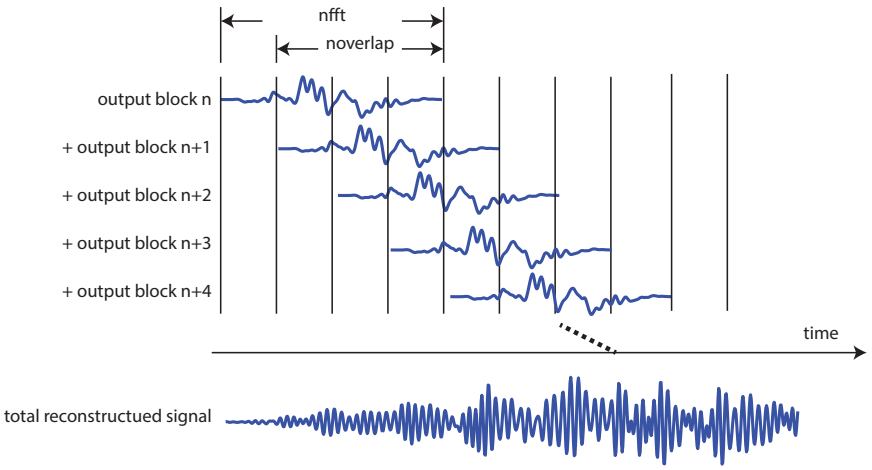

Figure 3: Shifting and adding IFFT outputs to the output buffer.

3 Part 2: Filtering in the Frequency Domain

Once you have your basic STFT framework working, you should test it at various values of `nfft` and `noverlap` to determine values that run in real time. In this part, you will add a frequency-domain filtering step between steps 2.3 and 2.4. Specifically, given `nfft`, you should compute the complex frequency response of your filter in Matlab via

```
H = freqz(b,a,nfft,'whole');
```

Note the `'whole'` option which tells `freqz` to provide `nfft` points around the whole unit circle. The `H` variable should be stored as a static array of type COMPLEX in your C code. Performing frequency domain filtering just requires computation of the element-wise complex product of the output of the FFT and the complex filter response. Recall that the product of two complex numbers

$$(a + jb)(c + jd) = (ac - bd) + j(bc + ad).$$

You can do this computation in place (like the FFT) if you like. After computing this product, you then perform the IFFT and subsequent processing in the same manner as Part 1.

4 In Lab

You will work with the same lab partner as in the prior laboratory assignments. Please contact the instructor if your lab partner has dropped the course or if you have concerns about your lab partner's performance on the prior assignment.

5 Specific Items to Discuss in Your Report

In addition to verifying the correct operation of your code, you should discuss and compare your profiling results for a direct implementation of the filter via convolution (Laboratory Assignment 2) versus frequency domain filtering as performed in this assignment for different values of `nfft` and

`noverlap`. What are the tradeoffs in selecting `nfft` and `noverlap`? Can you satisfy the filter specifications with less computations using frequency domain filtering? Can you explain your profiling results? What parts of the processing require the most cycles? How does this change as a function of `nfft` and `noverlap`?