ECEN4002/5002 Digital Signal Processing Laboratory

Spring 2002

Laboratory Exercise #6

Using a Fast Fourier Transform Algorithm

Introduction

The symmetry and periodicity properties of the discrete Fourier transform (DFT) enables a variety of useful and interesting decompositions. In particular, by clever grouping and reordering of the complex exponential multiplications it is possible to achieve substantial computational savings while still obtaining the exact DFT solution (no approximation required). Many "fast" algorithms have been developed for computing the DFT, and collectively these are known as Fast Fourier Transform (FFT) algorithms. Always keep in mind that an FFT algorithm is not a different mathematical transform: it is simply an efficient means to compute the DFT.

In this experiment you will use a provided FFT macro to perform some frequency domain processing tasks. The framework for doing real time processing with an FFT algorithm is somewhat different than what was used in the previous experiments, since the DFT requires a block of input samples to be available before processing can begin. So rather than being able to run the processing algorithm "in between" the sample period, it will be necessary to buffer a block of samples and then start the FFT while still receiving the new samples as they arrive from the A/D and providing the output samples to the D/A. This inherent delay, or processing *latency*, is separate from the time required to compute the FFT itself.

Working with the FFT

There are a large number of FFT algorithms suitable for use with the Motorola 56307. The version we will use was written for the original 56000 and is not particularly optimized for the '307, but it should work sufficiently well for our purposes. Note that there will be a bunch of pipeline stall warnings when the code is assembled. The DFT requires complex numbers (real, imaginary), and it is natural to use the Harvard structure of the processor to represent the complex data: real part in X memory and imaginary part in Y memory. The assembler allows the simultaneous declaration of memory in both the X and Y memory spaces by using the L memory specifier. For example, to declare arrays of complex numbers:

```
points      equ   256   ;length of FFT input signal

            org   L:$0200
indata      dsm   points      ;real,imag parts of signal
outdata     dsm   points      ;real,imag parts of FFT(signal)
coef        dsm   points      ;twiddle factors, forward
icoef       dsm   points      ;twiddle factors, inverse
```

You can find a copy of the FFT macro file, `fftr2cn.asm,` on the course web site. This particular FFT implementation is used as follows:

```
      fftr2cn   points,indata,outdata,coef
```

where `points` is the FFT length, `indata` is the base address of the input data (real in X memory, imaginary in Y memory), `outdata` is the base address of the output buffers (real in X, imaginary in Y), and `coef` is the base address of the FFT twiddle factors (real X, imaginary Y).

The twiddle factors (complex exponential terms) are calculated using the macro file `sincosw.asm`. The twiddle macro is used by:

```
      sincos   points,coef,icoef
```

where (again) `points` is the FFT length, `coef` is the base address of the twiddle factors for the forward FFT, and `icoef` is the base address for the inverse FFT twiddles.

The `fftr2cn` FFT macro is complex-in, complex-out, and the input and output arrays are in normal order (not bit reversed). Be aware that if the input data is real only, you must set the imaginary part of the array (Y memory) to zero before calling the FFT. Note also that this particular FFT implementation "consumes" the input array during the calculation, so don't count on the input data being the same after the FFT is run. This makes the routine somewhat extravagant in memory usage, but somewhat faster to execute.

There is a scaling detail to consider when using this FFT algorithm. These forward and inverse FFTs are implemented without scaling, so the internal storage may overflow due to "bit growth" as the FFT butterflys are added together. Although the usual DFT definition has the 1/N factor applied to the inverse transform, it is helpful to scale down the input signal by 1/N prior to the forward transform to avoid overflow. The overall FFT/IFFT gain is not changed by this adjustment.

Another detail is that the inverse FFT is computed using the same algorithm as the forward FFT, except the real and imaginary parts are exchanged. Rather than actually copying the data between X and Y memory, the IFFT routine was re-written to refer to the complementary memory spaces.

⇨Exercise A: Test the FFT using the debugger or the simulator

Write a non-real time FFT test program that can be run on the EVM debugger or with the software simulator (`sim56300.exe`). Provide a way to include an input signal generated by Matlab. One way to do this would be to modify the file-writing portion of the `firtable.m` file to create a signal file to include in your test program. Also determine the debugger or simulator commands necessary to save the DSP memory contents to a file on the PC.

The program framework should be something like the following (see the file `fft_test.asm` from the course web site):

```
;fft with input,output in normal,normal order
      include 'fftr2cn.asm'

points      equ   256         ;length of FFT signal

      org   l:$0000
data1       dsm   points      ;real,imag parts of signal
odata1      dsm   points      ;real,imag parts of FFT output
odata2      dsm   points      ;real,imag parts of IFFT[FFT(signal)]
coef        dsm   points      ;twiddle factors, forward
icoef       dsm   points      ;twiddle factors, inverse

      org     x:data1         ;put real input signal in x memory
```

Your include file for the real part of the input signal goes here. Your signal should be the same length as defined by `points` ***above.***

```
      org     y:data1         ;put imag(signal) in x memory
```

Your include file for the imaginary part of the input signal goes here (can be all zeros). Your signal should be the same length as defined by `points` ***above.***

```
      include 'sincosw.asm'         ;macro to...
      sincos  points,coef,icoef     ;...build twiddle factor tables

      org   p:$0
      jmp   begin

      org   p:$100
begin
```

You should scale the input signal to avoid overflow in the FFT. Multiplying the real and imaginary parts by (1/`points`) is a satisfactory approach. Note that since points is a power of two, you can do the scaling using an arithmetic right shift (`asr`) of the appropriate number of bits.

```
;FFT - do the FFT on data1 and store results in odata1, using the FFT
; macro "fftr2cn".  Note that the results will be scaled by 1/points
; (just done above) compared to the textbook DFT definition.

      fftr2cn  points,data1,odata1,coef

BREAKPOINT1 nop   ; Putting a breakpoint here will allow memory to be
                  ;  examined after the FFT.

;IFFT – Now do the IFFT on the odata1 FFT result.

      ifftr2cn  points,odata1,odata2,icoef

BREAKPOINT2 nop   ; Put a breakpoint here to look at IFFT result

      jmp   *     ;End by looping here forever
```

If you use the debugger you will notice that the program takes a long time to load. The main issue is that the constant declarations generated by the `sincos` macro for the FFT twiddles must be downloaded individually to the EVM board.

Test the program using input sinusoids of several different frequencies (two cycles per buffer, 8 cycles per buffer, etc.), and some other test files of your own choosing. Create the input data files (real and imaginary), assemble the program, and load into the debugger or the simulator. Then place breakpoints after the FFT and the IFFT so that you can dump the memory contents to a file. Compare the results from the 56307 FFT to results computed with Matlab. Do you get what you would expect? Comment on the results.

Additional exercise for Graduate Students:
The `fftr2cn.asm` file contains code that is compact, but doesn't handle the 56307 pipeline very well. Carefully examine the warnings generated by the assembler (look in the `.lst` file) and modify the FFT instructions to eliminate the warnings. Perhaps start by inserting nop instructions, but try to re-order the execution to make useful actions with each line of code. Be sure to verify that your changes still make the FFT work!

Short-time Fourier transform in real time

One interesting use of the FFT is to implement linear time-invariant systems. The idea is to break the input signal into blocks, perform the FFT on each block, multiply by a filter function in the frequency domain, then IFFT to reconstruct the filtered time domain signal. Because the FFT provides the means to reduce the

computational complexity of the DFT from order (N^2) to order ($N \log_2(N)$), it is often feasible to do FFT-based processing for DSP systems. Even with the computational cost of doing both the FFT and IFFT may be lower than doing the equivalent computation with conventional time domain methods.

The DFT is a frequency-sampled version of the Fourier transform, so multiplying the DFT by a filter function in the frequency domain is actually the equivalent of *circular* convolution, not linear convolution. This means that the resulting time domain signal may have "time domain aliasing" if the effects of the circular overlap are not accounted for. Refer to the lecture notes or a DSP textbook for the details of this issue.

Nevertheless, for this experiment we are going to use a somewhat crude short-time processing algorithm. The concept is as follows.

(1) We will segment the input signal into overlapping blocks. The overlap will be 50%, and each block will be "windowed" by a smooth raised-cosine function (hanning window). The window will be chosen so that the original signal can be reconstructed perfectly if no signal modification is done (see Figure 1).
(2) For each windowed block, the FFT will be calculated. This gives a spectral "snapshot" of what is going on during that short block of the input signal.
(3) We can now do some modification of the FFT data, such as multiplying by a spectral shape (filter) or some other type of frequency-domain processing. Assuming that we had a real-only input signal and we want a real-only output signal, we need to make sure that whatever manipulation is applied gets done in a way that will keep the complex DFT data in *conjugate symmetric* form. That is, if we change anything in the DFT bins between 0 and $N/2$, the same change needs to apply to the mirror image bins N-1 down to $N/2$, and the resulting modified DFT data must remain conjugate symmetric.
(4) After the spectral processing, the IFFT (inverse FFT) is calculated. The resulting block is then *overlap-added* to the output buffer, thereby reconstructing the desired signal.

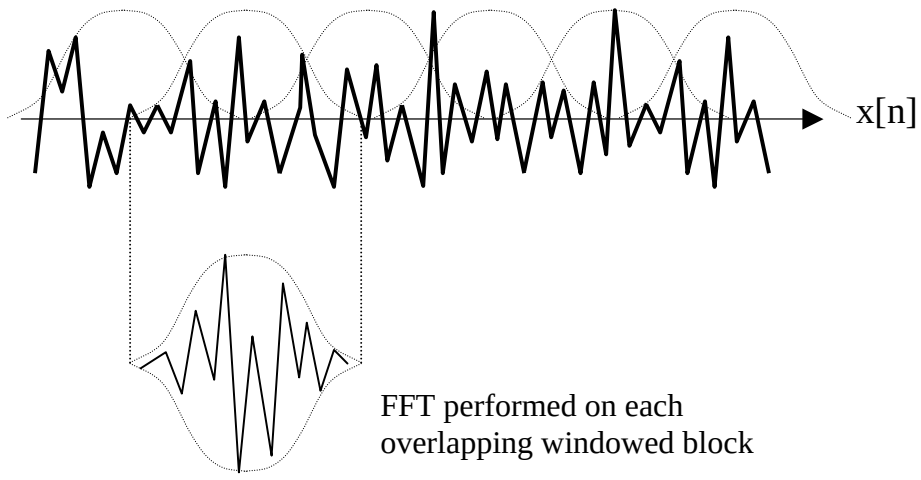

Figure 1: *Signal segmentation and overlap-add reconstruction.*

Why is it necessary to window and overlap the analysis blocks? There are several issues here. It is theoretically possible to do a huge FFT over the entire input signal, but for practical reasons we usually want to limit the time delay and storage memory of a real time system. Breaking the signal into shorter blocks provides this opportunity. Applying the smooth window function is helpful in reducing the truncation effects that otherwise would be evident in the DFT data, but this may or may not be important depending on the application. Finally, the overlapping windows provide a smooth transition from one block to the next, and this is important if the frequency domain processing varies with time.

The choice of FFT length, overlap amount, window shape, etc., can be made with a solid theoretical basis. Consult a DSP text book for more information on the theoretical underpinnings of the short-time Fourier transform.

A skeleton program `stft6.asm` for the simple short-time FFT procedure is available on the course web site. This program has a different set of interrupt service routines (ISRs) than we used for the previous assignments. In the earlier exercises we did the DSP calculations in the interval between the arrival of each new sample: the code polled the A/D for data, then called the `process_stereo` routine with a pair of {left, right} samples in the A and B accumulators. For the FFT the procedure needs to be somewhat different, since the FFT can only run if an entire block of input samples has arrived.

The STFT skeleton is organized as shown in Figure 2. In the "background," the ISRs collect the input data into left and right input buffers, and play the output data from the left and right output buffers. In the "foreground" the FFT routine waits until a block of input data has arrived, then performs the window/FFT/modify/IFFT/overlap-add sequence for that block. The foreground routine then waits for the 50% overlap period, and runs again on the overlapped data. This process continues over and over as long as the program is running.

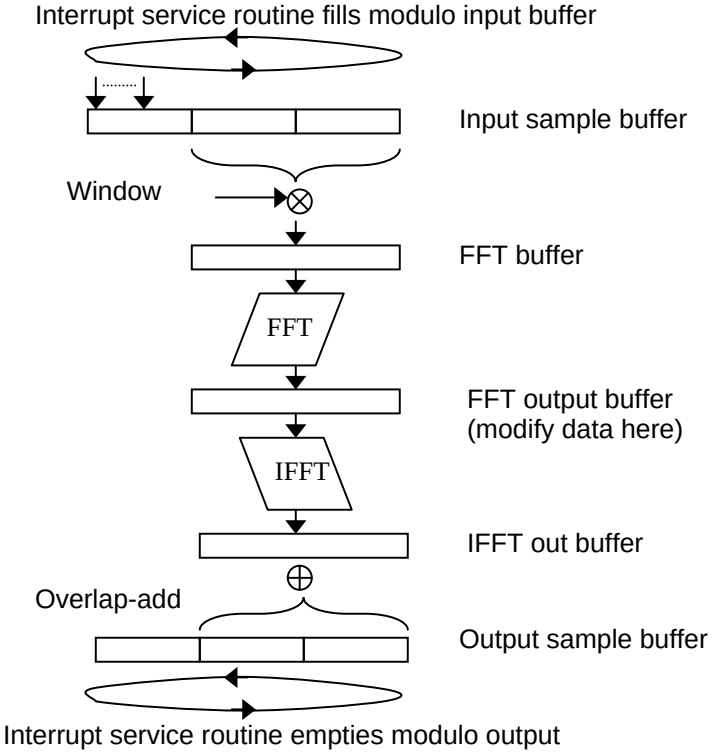

Figure 2: *STFT skeleton program flow.*

Your application code should be placed in between the FFT and the IFFT. You will have access to the complex DFT data for each block, while the skeleton code handles the data I/O.

⇨ Exercise B: Run FFT/IFFT pass program

Obtain the `stft6.asm` program and the supporting files (`fftr2cn.asm, sincosw.asm, cdc_init.asm`) from the course web site. Assemble the program, check for any errors, and download to the EVM board for testing. Note that the FFT will have numerous pipeline stall warnings (see graduate exercise above). The skeleton should act like a "pass" program: the LEFT CHANNEL input data is passed through the window/FFT/IFFT/overlap-add procedure without deliberate modification, while the RIGHT

CHANNEL is simply buffered and passed to the output (no FFT on the right channel). Verify that this is working. Also, determine the processing delay: what is the lag between the input signal and the output signal? Finally, insert some instructions in the "`USER CODE GOES HERE!`" position to change the gain of the pass program: multiply the FFTLEN real and imaginary parts of the DFT by a constant. Comment on the results.

⇨Exercise C: STFT for signal processing

Now that the basic pass program is working, you can consider some more interesting STFT-based processing. In this exercise you will modify the DFT data in a frequency-dependent manner.

Part 1: Bandpass filter

Devise a way to make a simple bandpass filter centered at 4kHz, with a narrow bandwidth. The idea is to multiply the FFT bin(s) corresponding to 4kHz by unity, while setting all the other bins to zero. Make sure you understand the DFT frequency sampling concept and the symmetry requirements. Test your code using input signals with a range of frequencies so that you can plot the frequency response.

Next, modify your code so that the program includes a data file describing the filter shape. The include file should be a sequence of "`dc 0.xxxx`" statements giving the desired gain factor for each FFT bin. Remember the symmetry requirements! Test your code for a few different filter response shapes (low pass, highpass, etc.).

Part 2: Frequency domain "noise gate"

In this part you will do some signal-dependent processing. Since the DFT gives a complex view of the input signal's short-time spectrum, we can take advantage of the spectral analysis to do some signal enhancement.

It is common to have an input signal that is contaminated with unwanted broadband noise. One way to reduce the undesired noise is to use a spectral *threshold*. The algorithm is to compare the spectral magnitude in each FFT bin to a threshold value. If the magnitude is above the threshold, it is assumed to be "signal" and gets passed unaltered. On the other hand, if the measured magnitude in an FFT bin is below the threshold, it is assumed to be noise and the bin is set to zero. If the threshold is chosen carefully, the output signal will have less audible noise than the input signal. This process is known as a "de-hisser" or a spectral "noise gate."

In order to do the threshold test you will need to write code to calculate the magnitude (or magnitude squared) of the complex DFT values. This may be tricky due to overflow issues, so you will need to determine a good way to scale the data during the magnitude calculation. You will also need to experiment with various threshold values, and try different types of input signals to verify that your de-hisser is working. Construct some test signals with various levels of broadband noise and devise a way to demonstrate your code.

Report and Grading Checklist

A: FFT testing with non-real time code

Description of your testing strategy and the test signals you used.
Scaling discussion.
Comparison with Matlab results, and comments on the behavior.
Grad student exercise, if applicable

B: STFT testing with real time code

Comments on testing of real time code.
Source code for gain adjustment between FFT and IFFT.

C: STFT signal processing

Part 1: Your source code segment that implements the simple 4kHz bandpass filter, the source code segment that uses a filter shape file, and thedescription of the way you handled the symmetry issue. Comments.

Part 2: Code listing with comments for spectral noise gate. Provide a description of your processing strategy and how you tested your code.

Grading Guidelines (for each grade, you must also satisfy the requirements of all lower grades):

F	Anything less than what is necessary for a D.
D	Exercise A results with Matlab comparison.
C-	Exercise B results with clear and concise comments, no gain adjustments.
C+	Exercise B with gain adjustment code.
B	Exercise C part 1 (bandpass and general filter).
A	Exercise C part 2 (noise gate)

Grad student grades also require the additional exercise (part A).