ECEN4002/5002 Digital Signal Processing Laboratory

Spring 2002

Laboratory Exercise #2

Introduction

In this lab exercise you will investigate the sampling and reconstruction process, implement "delay lines" using digital storage, design and implement some FIR digital filters, and learn some additional debugging techniques. The report for this exercise is due at the start of class in two weeks.

This second experiment involves MATLAB, some new software examples, and some modifications to the programs prepared previously for Laboratory Exercise #1. As mentioned before, you are encouraged always to write your code in a highly modular fashion so that you have a good basis for modifications and customizations in future lab experiments.

In preparation for the experimental procedures in this Lab, you should review the FIR filter section of your DSP textbook and also look over the modulo addressing features of the 5630x processor (see the Address Generation Unit description in the online documentation on the lab PCs).

Sampling and Reconstruction

In many DSP applications it is necessary to begin with an analog signal, sample it, perform digital processing, then reconstruct an analog signal for the output. This only makes practical sense if it is possible to perform the sampling and reconstruction process with minimal error.

Recall that the discrete-time Fourier transform $F(e^{j\omega t})$ of a sampled signal $f(kT)$ can be expressed in terms of the Fourier Transform $F(j\omega)$ of the analog signal $f(t)$ by

$$F\left(e^{j\omega t}\right)=\frac{1}{T}\sum_{n=-\infty}^{\infty}F\left(j(\omega+\frac{n2\pi}{T})\right)$$

This expression shows that the spectrum of the sampled signal consists of replicas (or images) of the analog spectrum that are scaled by 1/T and shifted in frequency by multiples of $2\pi/T$. In order to be able to reconstruct the original analog signal we at least need to be able to isolate the baseband replica of the spectrum (n=0 term, centered at $\omega=0$) from the shifted replicas, which means that the replicas cannot overlap each other. We must ensure that the sampling rate is greater than twice the highest frequency present in the analog spectrum, or in other words, ensure that the input signal is strictly bandlimited to less than half the sample rate, to avoid the overlap (*aliasing*).

How does the reconstruction process work in theory? Consider that the reconstruction task consists of isolating the baseband spectral image from the shifted images in the DTFT of the sampled signal. This implies that we design a filter function that is constant over the frequency range occupied by the baseband image, and zero at higher frequencies to remove the shifted images. Ideally, this reconstruction filter would be a perfect lowpass filter: a rectangular pulse $P(j\omega)$ in the frequency domain.

The theoretical perfect lowpass filter in the frequency domain becomes a sinc( ) function when inverse transformed to the time domain. Therefore, the reconstruction theory indicates:

$$\hat{F}(j\omega) = P(j\omega) \times \left[\frac{1}{T} \sum_{n=-\infty}^{\infty} F\left(j(\omega + \frac{n2\pi}{T}) \right) \right] = F(j\omega) \text{ if ideal LPF and no aliasing}$$

Therefore,

$$f(t) = \hat{f}(t) = \sum_{k=-\infty}^{\infty} f(kT)\, p(t-kT) = \sum_{k=-\infty}^{\infty} f(kT) \cdot \text{sinc}\left(\frac{\pi}{T}(t-kT) \right)$$

It is not possible to implement this perfect reconstruction process in a practical system, since the formula involves an infinite and non-causal summation. Moreover, the effects of sample amplitude quantization must also be considered. So, in practical DSP systems we will need to approximate this formula using various engineering tradeoffs.

⇨Exercise A: A/D and D/A Simulation via MATLAB

In order to get a feel for the effects of sampling and aliasing during the A/D conversion process, do several trials using the MATLAB scripts '**adcdemo**' and '**dacdemo**', which may be found on the class web site (http://schof.colorado.edu/~ecen4002). Use the MATLAB help command to discover some options. Before you invoke one of these demos, you must initialize each of the following undefined variables.

> **s** is a character coding the signal type. For our purposes, `s='s'`, is appropriate, although you should take a look at other types (`'c'` – chord, `'u'` – unit pulse, `'n'` – noise, `'b'` – band noise).
>
> **type** is an integer coding the type of reconstruction filter. `type=0` is none, `type=1` is an ordinary DAC or sample and hold, `type=2` is linear interpolation, and `type=3` is Shannon or sinc reconstruction.
>
> **ratio** is the upsampling ratio. You can start with `ratio=8`, but feel free to explore other values.
>
> **alpha** is the normalized frequency. `alpha=1` corresponds to $f_s/2$ (the fold-over frequency), `alpha=2` is the sampling rate (f_s), and so forth.

(1) Using a sinusoidal input with *type=3*, run **adcdemo** for several values of `alpha`. This tool demonstrates aliasing in the A/D followed by D/A conversion processes when **no analog anti-aliasing filter is used at the input**. Pay close attention to the region 0.8<`alpha`<1.2. In this region there are two competing aliases, one coming down in frequency from the spectral image centered at f_s and the other going up from the baseband image. Thus, you will get a beat frequency of 2*(1-`alpha`), normalized. Also note the aliasing behavior for `alpha` greater than one. At `alpha=2`, the alias frequency is zero. Using this tool, sketch a plot of output frequency vs. input frequency. Keep in mind that if an anti-aliasing filter was used, sinusoids of frequency greater than `alpha=1` ($f_s/2$) would simply be filtered out. Make copies of the output figure for a few cases, (say, for `alpha`=0.44, 0.92, 1.44, 2.44) for inclusion in your report and describe what is going on in each figure.

(2) The tool **dacdemo** demonstrates variations on the D/A conversion process. You can see the conversion errors in both time and frequency for the four types of reconstruction filter. You can also see the unit pulse response of the reconstruction filter by setting `s='u'`. The CODEC on our EVM board should look like `type=3`. Holding all input variables the same, except for `type`, make copies of the figures for the four reconstruction variations for inclusion in your report. Describe what you view in each of these figures.

Real Time Processing Time Limitations

In order to keep up with the non-stop arrival and departure of samples in a real time DSP system, it is necessary that all the processing be accomplished within one sample interval, *T* (at least on average). What happens if our '`process_stereo`' function requires more processing time than the sample interval? We will end up missing the arrival of the next input samples (and the departure of the next output samples) and data will be lost. This will produce gaps or roughness in the output signal, which is generally unacceptable.

The 5630x processor includes a phase lock loop (PLL) in its clock generator that allows the internal clock rate of the processor to be a multiple or a division of the external crystal rate. The rate multiple/division can be set by the DSP software itself. This allows, for example, the processor to go to a low power state by selecting a lower clock speed if the processor is idle. Take a look at the PLL description in the *DSP56300 Family Manual* in the online documentation.

Examine the `pass1.asm` program (remember: different for the '303 and the '307) and find the instruction that sets the PLL control register (`x:M_PCTL`). This is probably being set to something like $040xxx , where the '4' sets the PLL enable bit, and 'xxx' (12 bits) is the multiplication factor applied to the external clock. For example, the '307 EVM boards have a 12.288MHz external clock crystal, so the internal core clock rate is (12.288MHz)*(xxx) . Keep in mind that the chip can't actually run at an arbitrarily fast clock rate: the max is something between 80 and 100 MHz (see the chip data sheet).

⇨Exercise B: Determine how many instructions "fit" in one sample period

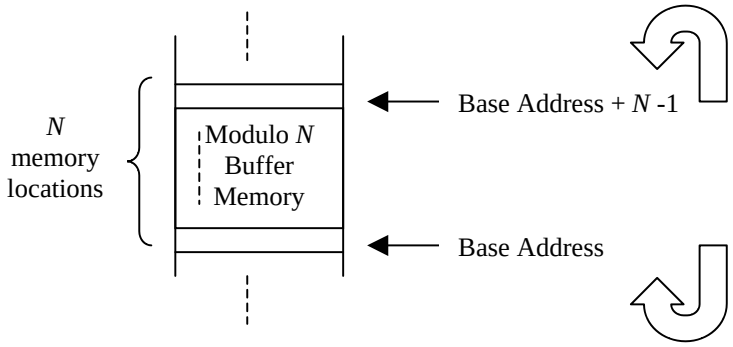
The point of this exercise is to figure out how many instructions we can place in '`process_stereo`'. Modify '`process_stereo`' by introducing some instructions (perhaps a `rep` statement or a loop) which can be used to determine how many instruction cycles are in each sampling period. After finding this practical number, use the PLL setting from your `pass1.asm` program, the 5630x Processor Family manual and the Users Guide for the EVM to find the instruction rate of the EVM board. By taking the ratio of this rate to the sampling rate, we can calculate how many instructions can be executed within a sample period. Compare this number to the number you found above and speculate on what might cause the difference. What is the internal core clock rate set in `pass1.asm`? Can you increase it to at least 80MHz?

Using the DSP to Implement Time Delay

As mentioned above, most DSP algorithms require a means to delay one digital signal with respect to another. This means that the sequence of signal samples must be stored in memory temporarily. For a typical delay buffer this means that we want a first-in, first-out (FIFO) queue, where the length of the queue is equal to the desired delay in samples.

The 5630x supports modulo address buffers. This means that we can use an address register to point to a block of memory locations, and incrementing or decrementing the address register causes the address to automatically wrap around the head and tail of the buffer. We can make a delay buffer by storing the current input sample at the location pointed to by the address register, then incrementing the address register and reading back the contents of the next location. Since the buffer is addressed in a modulo or circular fashion, the address pointer will eventually wrap around the end of the buffer and proceed back through the previously stored values.

The 5630x uses the modulo registers (M0-7) in the address generation unit to specify the length of the modulo buffer. Storing $ffffff in the modulo register sets the AGU for normal, linear addressing. This is the default state after the chip is reset. Storing zero in the modulo register causes the address to increment in "bit reversed" fashion. We will use this mode later in the course when we cover the FFT. For modulo addressing we store the integer *N*-1, where *N* is the number of memory locations we want in the modulo buffer. For example, if we want a delay buffer with 16 elements, we would store 16-1=15 in the M register.



Another important detail of the modulo addressing is that the lowest address of the modulo buffer must be on a valid "page" boundary. Specifically, this means that the base address of the modulo buffer must always be a power of 2, and further, it must have its lowest *k* bits equal to zero, where $2^k = N$, the buffer length. So, as the address register is incremented or decremented the lower *k* bits of the address register change in value while the upper 24-*k* bits stay unchanged. *Note* that we don't specify the beginning and ending of the modulo buffer directly: The upper 24-*k* bits of the address register *imply* the base address.

The 56300 assembler includes several directives to declare modulo buffers. For example, the `DSM` directive (define storage modulo) tells the assembler to advance its memory pointer to the next valid base address for the specified buffer size.

⇨Exercise C: Implement a Delay Line in Real Time Software

Begin by making a copy of the `pass1.asm` and `lab1_p.asm` programs that you used in Lab #1, and rename them something like `pass2.asm` and `lab2_p.asm`. Recall that `pass1.asm` contained a directive to include `lab1_p.asm`, and that your subroutine `process_stereo` was called for every input sample with the left channel data in accumulator A and the right channel sample in accumulator B. Modify your `pass2.asm` and `lab2_p.asm` files so that `pass2.asm` includes `lab2_p.asm`, then assemble, load, and test to make sure the pass features are working OK. Remember, work *incrementally* as you make the series of required changes.

Now modify your `pass2.asm` so that you can execute your own buffer initialization instructions: add a line

```
        jsr    my_init
```

after the existing line `jsr ada_init` .

Then edit `lab2_p.asm` and add a new subroutine (if necessary) called:

```
my_init
        …etc…etc…etc…
        rts
```

You should now modify your `process_stereo` routine so that it passes the left channel input (accumulator A) unaltered to the output, but applies a *delay* to the right channel (accumulator B).

Begin with a right channel delay of $100 samples, but try to write your code in such a way that you can change the delay amount by changing only a single EQU statement and then re-assembling the file. Your `lab2_p.asm` file will be something like:

```
YBASE       EQU   $200
BUFSIZE     EQU   $100

      org y:YBASE
delay_buf   ds    BUFSIZE

      org p:
my_init
            …initialize an index register and modulo register (e.g., r4 and m4) to use delay_buf in Y
            memory;  initialize the delay buffer contents to be all  zero, etc…
      rts

process_stereo
            …move the right sample (B) into the modulo buffer delay_buf, and retrieve the next
            delayed sample from the buffer.  Leave accumulator (A) unchanged.
      rts
```

Make sure you understand what the routines are doing, modify the skeleton as necessary, and try it out. Verify that you are getting the expected $100 sample delay between the left and right signals.

(1) What time (in seconds) does this delay of 256 samples represent? Try playing some stereo audio through your software. Is the inter-channel delay audible?

(2) Now modify (if necessary) your code and data initialization so that you can test the following delays between the left and right channel: 2 samples, 21 samples, 73 samples, 767 samples, and 1024 samples. Can you use the modulo addressing features to get a delay buffer of 1 sample? Discuss the details in your report.

(3) For the EVM you are using, determine how much on-chip Y memory you have available. Use all the available Y memory to make the longest possible Y delay line. Make sure you don't overwrite any storage used by `pass.asm`, `ada_init.asm`, etc.! Also, make sure your buffer begins on a valid modulo address boundary.

Additional exercise for Graduate Students:
Determine how much internal X and Y memory is available, and re-write your software so that the total delay consists of one delay line in X memory that feeds another, separate delay line in Y memory. Include your program and discussion in your lab report.

Non-Real Time Testing Using I/O Files and the Debugger

It is often useful to be able to test the implementation of an algorithm using a non-real time method. This allows single-stepping the program, running to a breakpoint, etc., without the difficulty of handling real time interrupts, A/D and D/A timing, and so forth. It is also useful to be able to process a known sequence of samples for the input and then to be able to collect the output samples as a computer file for later analysis. For example, by collecting the output samples for a known input file (a "test vector"), the results from the DSP can be compared bit-by-bit to a reference output example.

The Domain Technologies Debugger software provides a mechanism to transfer data from a file on the PC to the memory of the DSP, and also to transfer data from the DSP's memory back to a file on the PC.

Our plan is as follows. We will write a DSP "wrapper" program that initializes the DSP chip and allocates input and output buffers. The DSP wrapper will then commence a simulated input/output loop by reading data from the PC using calls to the Debugger, executing a process we provide, and then writing the results back to the PC.

First, consider the I/O features of the EVM30xw Debugger. At the Debugger command window prompt:

```
INPUT infile
OUTPUT outfile
```

These commands tell the Debugger to open *infile* and wait for the DSP to request data, and to open or create *outfile* and wait for the DSP to send data back. The Debugger assigns a file number (handle) to each input file and a file number to each output file as they are opened. The input and output files are regular ASCII text files, and there are options to choose hexadecimal (default), decimal, or fractional number formats. See the commands `INPUT` and `OUTPUT` in the online help for the EVM30xw Debugger. Also, consider using the Debugger's `PATH` command to set the location of your input and output files.

An example of the DSP code for file I/O looks like this:

```
;input data from PC
      move    #$200,r0        ; start address to load data ($200)
      move    #$10100,x0      ; High 8 bits indicates file number (1),
                              ;  low 16 bits indicates number of
                              ;  words to read ($100)
      move    #$8001,r1       ; $8xxx indicates input to DSP,
                              ; $xxx1 indicates X memory space
      debug                   ; invoke emulator service
```

The debug operand calls the debugger "service" that uses the values in r0, r1, and x0 to determine what action to take. In this example, the Debugger reads 256 words from input file #1 and stores them sequentially beginning at X:$200.

After the desired processing is complete, we write the results back to the *outfile* on the PC using the following instructions:

```
;output data to PC
      move    #$400,r0        ; start address to retrieve data ($400)
      move    #$10100,x0      ; High 8 bits indicates file number (1),
                              ;  low 16 bits indicates number of
                              ;  words to write ($100)
      move    #$4001,r1       ; $4xxx indicates output from DSP,
                              ; $xxx1 indicates X memory space
      debug                   ; invoke emulator service
```

In this example, the Debugger reads 256 words beginning at X:$400 and writes them to output file #1.

Once the process is complete, use the Debugger commands

```
INPUT OFF
OUTPUT OFF
```

to close the files so that you can examine the results.

The I/O framework for the "wrapper" is given below (a copy of this file should also be on the class web site). The file is `passio.asm`. Since the codec will not be used there is no need to initialize the A/D and D/A, nor is there a need to handle the codec interrupts. The program creates some small input and output buffers, then begins to read data from the input file The section "I/O block loop" calls a user-supplied subroutine `process_mono` that receives its input in the A accumulator, and returns its output A as well. The program collects the results and writes them to the output file. Note also that there is a call to the user-supplied subroutine `my_init`. You probably do *not* need to modify the `passio.asm` file, but you will

need to create the file `lab2_io.asm` containing your own memory declarations and the `process_mono` and `my_init` subroutines.

```
;*****************************************************************
;
; ECEN4002/5002 DSP Lab       Spring 2002
; Example program:  passio.asm
;
; Takes data from Debugger input file 16 words at a time, then
; loads them one by one into the A accumulator and calls user-
; supplied 'process_mono' function.  Result in A accumulator is
; sent to Debugger output file.
;
; User provides lab2_io.asm  include file containing 'my_init'
; and 'process_mono' subroutines and memory declarations.
;
; To enable data streams type from EVM30x Debugger COMMAND window:
;   INPUT data_in -fra      ;this will open input file for reading
;                           ; fractional (not exponent) data
;   OUTPUT data_out -fra    ;this will open output file for writing
;                           ; fractional data
; Do these Debugger commands BEFORE running the program.
; Remember to 'INPUT off' and 'OUTPUT off' at the end of the run.
;
;*****************************************************************
        nolist
        include 'ioequ.asm'
        list

BLOCK   EQU     $10                ; number of words in file I/O
INFILE  EQU     $10000             ; debugger:  infile #1 handle
OUTFILE EQU     $10000             ; debugger:  outfile #1 handle
XMEM    EQU     $0001              ; debugger:  x memory bit field
YMEM    EQU     $0002              ; debugger:  y memory bit field
INDSP   EQU     $8000              ; debugger:  input to dsp bit field
OUTDSP  EQU     $4000           ; debugger:  output from dsp bit field

        org     x:0

inbuf   ds      BLOCK                   ; buffer for data in/out
bufptr  ds      1                       ; temp storage for R register

        org     p:$0000                 ; Program block starts at zero
        jmp     START                   ; Skip over interrupt vectors

        org     p:$0100
START:
        movep   #$040008,x:M_PCTL       ; PLL 8 x 12.288 = 98.016MHz
        ori     #3,mr                   ; mask interrupts
        movec   #0,sp                   ; clear hardware stack pointer
        jmp     DO_INIT                 ; Jump over user-supplied code

        include 'lab2_io.asm'           ; User-supplied include file
```

```
        org     p:
DO_INIT
        jsr     my_init                 ; Call user-supplied subroutine

FILEIO:
;input data from PC
        move    #inbuf,r0               ; start address to load data
        move    #(INFILE|BLOCK),x0      ; BLOCK words from file INFILE
        move    #(INDSP|XMEM),r1        ; X mem space, input direction
        debug                           ; invoke emulator service
; data now in x:inbuf
; (Debugger will generate breakpoint if error during read)

        do      #BLOCK,endl

        move    x:(r0),a                ; move input data to A
        move    r0,x:bufptr             ; save R0 contents

        jsr     process_mono            ; Call user-supplied subroutine

        move    x:bufptr,r0             ; restore R0 contents
        nop
        nop                             ; wait for pipeline
        nop
        move    a,x:(r0)+               ; Transfer output data
                                        ;  from A to mem
                                        ;  (overwrite input)
endl    nop

; At this point, BLOCK input values have been processed and overwritten
;  to x:inbuf.  Ready to write to output file.

;output data to PC
        move    #inbuf,r0               ; start address to xfer data
        move    #(OUTFILE|BLOCK),x0     ; BLOCK words to file OUTFILE
        move    #(OUTDSP|XMEM),r1       ; X mem space, output direction
        debug                           ; invoke emulator service


        nop                           ; Debugger will generate breakpoint
                                      ;  if error during write
        jmp     FILEIO                ; Loop back and read next block.
        end
```

⇨Exercise D: Non-real time testing with file I/O

Using MATLAB, a spreadsheet, or some other method, create a text input file containing at least 64 fractional numbers between –0.5 and +0.5 . Write a simple `process_mono` subroutine that adds 0.25 to each sample and delays by one sample, i.e., y[n] = x[n-1]*0.25. Assemble and verify your program using the `INPUT` *`infile.txt`* `-fra` and `OUTPUT` *`outfile.txt`* `-fra` commands with the Debugger. What happens when the end of the input file is reached? What happens if you forget to open and close the I/O files with the Debugger? Does the output file contain the expect results? Can you import the output file into MATLAB or a spreadsheet and produce a plot of the results? Now try putting breakpoints in the code, observing memory, changing register values, and so forth. Include your code and explain your results in your lab report.

Designing Linear Phase FIR Filters with MATLAB

In order to design a digital filter, we always start with a set of specifications for the filter. Filter order, cutoff frequencies, cutoff width and ripple in both the passband and stopband are the usual specifications given a particular application. However, these specifications contain some design tradeoffs, such as that the filter order generally must increase to obtain a decrease in cutoff width. Often, the cutoff frequencies are given through some sort of desired input/output characteristics, and we try to approximate these characteristics with the best filter possible constrained by the other specifications, such as computational complexity or sensitivity to coefficient quantization. This approximation problem lies outside of the scope of this class, but is addressed in most DSP and digital filtering textbooks.

After we have determined the filter's specifications we must obtain a method for realizing/calculating the filter and then determine what hardware it should be implemented on. For the purposes of this class, our hardware has already been selected (as the 5630x EVM board). We will tend to use fairly simple realizations, with the understanding that some errors and effects can be minimized with other filter structures.

FIR Filters

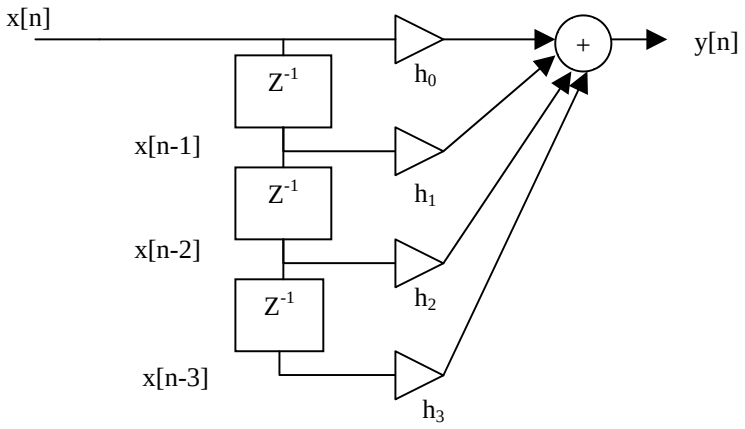
An FIR filter has a transfer function H(z) of the form

$$H(z) = \sum_{n=0}^{N-1} h(n) z^{-n}$$

This form can be seen from the structure of an FIR filter as shown in the next figure. As we can see from this transfer function, an FIR filter is a polynomial with zeros that define the characteristics of the filter. FIR filters have the following properties:

(i) Finite impulse response (hence the name FIR), as seen by using unit sample as the input.
(ii) Stability is guaranteed since they have no poles in their frequency response (only zeros as shown above).
(iii) A linear phase response is easily obtained, which implies that the filter only introduces a pure time delay at all frequencies.
(iv) To obtain a sharp cutoff filter the number of taps must be large, which increases the amount of computation.

The value of the first two properties is obvious. Linear phase is helpful in applications where frequency dispersion effects caused by a nonlinear phase response must be minimized. These applications include communication systems where the data pulse-shape and relative timing must be preserved (such as modems and ISDN networks), and hi-fidelity audio systems where the shape of the music must be preserved to prevent temporal distortion.



It can be shown that for linear phase a filter must have a *symmetrical* unit pulse response. That is, $h(N\text{-}1\text{-}k) = h(k)$.

As can be seen from the figure above, the filter coefficients for an FIR filter form the unit pulse response. Therefore, when given the filter design specifications, our goal is to find the unit pulse response of this filter. There are several methods to do this including *window* design of FIR filters and *frequency sampling* design of FIR filters. In this class, we will focus on the *window* designs, and do the actual work in MATLAB. We assume that our design is restricted to linear phase FIR filters.

Let $H_d(\theta)$ be the desired frequency response given our specifications. We want to choose an FIR filter with unit-pulse response $h(n)$ of length N, which is close to $H_d(\theta)$. In order to determine the closeness, we need some sort of criterion for measuring the distance between the spectrums. We will use the mean-square error ε between $H_d(\theta)$ and $H_1(e^{j\theta})$ on the interval $[-\pi, \pi]$. Therefore, we want to minimize

$$\varepsilon^2 = \frac{1}{2\pi}\int_{-\pi}^{\pi} \left|H_d(\theta) - H_1(\theta)\right|^2 d\theta \ .$$

Using Parseval's relation, we can express this in the form

$$\varepsilon^2 = \sum_{k=-\infty}^{\infty} \left|h_d(k) - h_1(k)\right|^2 \ .$$

From the theory of Fourier series, we can minimize ε^2 by choosing $h_1(k)$ to be the Fourier coefficients of $H_d(\theta)$ for $k_1 = k = k_2$, where the interval $[k_1, k_2]$ minimizes the error

$$\varepsilon^2 = \frac{1}{2\pi}\int_{-\pi}^{\pi} \left|H_d(\theta)\right|^2 d\theta - \sum_{k=k_1}^{k_2} \left|h_d(k)\right|^2 \ .$$

If $H_d(\theta)$ is real and even, and we take $k_2=M$, $k_1= -M$, then $H_1(e^{j\theta})$ will also be real and even but not causal. Fortunately, a causal filter having the same magnitude response and linear phase can be obtained by simply shifting $h_1(k)$ in time. Let

$$h(k) = h_1(k\text{-}M).$$

This filter is FIR of order $N=2M+1$, with linear phase.

There is one major defect in this method of design. At the points of discontinuity in $H_d(\theta)$, a characteristic overshoot in the approximation design $H_1(e^{j\theta})$ always occurs. This oscillation has been studied in connection with Fourier analysis and is known as the *Gibbs phenomenon*. Even though we have minimized the mean-square error between $H_d(\theta)$ and $H_1(e^{j\theta})$, this method of design is not really satisfactory for most applications, due to the ripple. To understand the reason behind these oscillations, consider the infinite length desired pulse response $h_d(k)$, which we truncated to length N for $h_1(k)$. This is the same as multiplying $h_d(k)$ by a window $w(k)$ as

$$h_1(k) = h_d(k)w(k) \ ,$$

where $w(k)$ is a rectangular window sequence

$$w(k) = \begin{cases} 1, & |k| \le (N-1)/2 \\ 0, & otherwise \end{cases} \ .$$

The resulting frequency response, $H_1(e^{j\theta})$, is obtained by taking the DTFT of both sides of this equation. We obtain

$$H_1\left(e^{j\theta}\right) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H_d(\phi) W\left(e^{j(\theta-\phi)}\right) d\phi \ ,$$

which is the convolution of the window's frequency response with our desired frequency response. Ideally, we want a window response that is similar to a delta function in frequency. However, no finite length window can have a unit-pulse frequency response. In fact, the frequency response for a rectangular window is

$$W\left(e^{j\theta}\right)=\frac{\sin\left(\frac{N\theta}{2}\right)}{\sin\left(\frac{\theta}{2}\right)}\ .$$

As the filter order *N* increases, the width of the main lobe of this window gets narrower, but the height of the side-lobe approaches a constant value independent of *N*. These are the two main properties of windows that are of interest in FIR filter design. The width of the main lobe is called the *resolution* of the window function and translates directly into the "smearing" that occurs at jumps in $H_d(\theta)$. Consequently, the transition width of each cutoff frequency is directly related to the width of the main lobe of the window. The height of the side lobe is sometimes called the *window leakage*. The ripple in both the pass-band and stop-band of the window is directly related to the height of the side lobe. We can reduce the ripple in $H_1(e^{j\theta})$ by using windows other than the rectangular window. However, the width of the main lobe is broadened as the side lobe height is reduced, which results in a design tradeoff between transition width and ripple. We can use the window functions in the Signal Processing toolbox supplied with MATLAB to look at the changes in $H_1(e^{j\theta})$ as the window shape changes.

MATLAB also provides a design method for linear-phase FIR filters. The function is `fir1`. Look up `fir1` in the online help for MATLAB. Note that the procedure is based on the "window" design method, and that a Hamming window is the default. Note also that the function will scale the output response so that the maximum gain is unity.

We would like to be able to design the filter and then to create a filter output file that is in a form compatible with the 56300 assembler. This way we can use an include directive to have the assembler load the filter coefficients without having to edit manually any of the files. Below is a MATLAB function `firtable` that converts a sequence of FIR coefficients (vector *h*) into a text file suitable for inclusion by the DSP assembler. A copy of this function is on the class web site.

```
function firtable(h,fname)
%
%  firtable(h,fname)
%
% Produce an assembler readable file defining the filter H(z)
% first line is the number of coefficents, n
% each following line contains one value h(k),
% from h(0) to h(n-1)
%
n=length(h);
fn=[fname,'.asm'];
fid=fopen(fn,'wt')
fprintf(fid,'; file %s\n',fn);
fprintf(fid,'; coefficients for an FIR filter with %d
coefficients\n',n);
fprintf(fid,';  line 1 contains the number of coefficients\n');
fprintf(fid,';  then come h(0), through h(%d), one line each\n',n-1);
fprintf(fid,' dc %d\n',n);
for k=1:n
   fprintf(fid,' dc %1.12g\n',h(k));
end
fclose(fid)
```

⇨Exercise E: MATLAB FIR design with the window method

Using the MATLAB function `fir1`, create a filter with the following specifications:

- Lowpass, linear phase FIR filter. Window design using a Hamming window (default for `fir1` function)
- Passband 0 Hz – 3kHz, defined by gain between +0.5dB and –0.5dB
- Stopband 7kHz – $f_s/2$, defined by gain below –(35+d), where "d" is the last digit of your student ID number. For example, if the last digit of your ID number is 7, the gain at and above 6kHz must be –42dB or less.
- Filter length *N* as small as possible while meeting the above requirements.

Note that you will probably need to iterate many times to find the optimum *N* and *Wn* when using `fir1`. The MATLAB function `freqz` may be useful, and you may want to write your own function to identify and verify the passband and stopband gain results.

Use MATLAB to plot the magnitude (dB vs. frequency) and phase of your filter and include your analysis in your report. Give sufficient results to verify your design. Also include the coefficient list and the last digit of your student ID number.

Next, use the `firtable` MATLAB function to produce an output file containing the definition of your coefficients and the filter order. You will use this file in the implementation section next.

Implementing and Testing FIR Filters

FIR filters are quite common in DSP systems, and many highly optimized implementations are available. We will use a subroutine that is compatible with the layout of the coefficient file produced in MATLAB by `firtable`. Specifically, the memory is organized so that the FIR filter's length, *N*, is stored first, followed by the *N* coefficients. Read and understand the following subroutine, including the required memory setup and register usage.

```
;************************************************************
;
; ECEN4002/5002 DSP Lab             Spring 2002
; Example program snippet:  FIR filter subroutine
;
; On entry:
;  accumulator A contains filter input
;  r2 contains address of 'filter definition block' (x memory)
;  ASSUME m2,m3,m4 set for linear addressing ($ffffff)
;  ASSUME registers do not need to be saved
;
; On exit:
;  accumulator A contains filter output
;  data buffer pointer updated in 'filter definition block'
;  registers changed:  r0,r2,r3,r4,m0,x0,y0,a
;
; Filter Definition Block
;  (element 1):  pointer to coefficient table (implicit y memory)
;  (element 2):  pointer to head of filter state (implicit x memory)
;
; Coefficient Table
;  (element 1):  number of filter taps (integer)
;  (elements 2-[2+taps]):  Fractional constant
;                          FIR coefficients 0-(taps-1)
;                          stored sequentially in y memory.
;                          Need not be modulo buffer aligned.
```

```
; Filter State
;  Delay line stored sequentially in x memory.  Base address must
;   be valid alignment for modulo buffer of length taps+1 .
;
;***********************************************************

firfilter
        move    x:(r2)+,r4      ; r4 points to start of coefficent table
        move    x:(r2),r0       ; r0 points to interior of filter state
        nop                     ; wait for pipeline
        nop
        move    y:(r4),r3       ; r3 contains number of filter taps
        move    y:(r4)+,m0      ; m0 contains number of filter taps
        nop                     ; wait for pipeline
        nop
        nop
        move    a,x:(r0)        ; copy input to filter state (overwrites
                                ;  oldest value)
        clr a   (r3)-           ; clear A, r3 contains taps-1
        move    x:(r0)+,x0      ; copy input to x0
        move    y:(r4)+,y0      ; get first coefficient into y0

        rep     r3              ; repeat next line (mac) taps-1 times
        mac     x0,y0,a  x:(r0)+,x0  y:(r4)+,y0 ; mac and zipper

        macr    x0,y0,a  r0,x:(r2)   ; do final mac,
                                     ;  round result (macr),
                                ;  and save head pointer for next time
        rts
```

You should now be ready to create an assembly language implementation of the FIR filter and to test it using the non-real time (file I/O) method. The plan is to create a text input file consisting of a unit sample function (a file with only one non-zero sample), then to observe the output file to verify that the unit sample response (impulse response) of the filter is obtained. Recall that the unit sample response of a finite impulse response filter is just that: the sequence of coefficients in the FIR filter.

⇨Exercise F: Non-real time implementation and testing of your FIR filter

Modify the `lab2_io.asm` program with an include directive for the filter coefficient file, the `firfilter` subroutine, and the `process_mono` and `my_init` subroutines. Then assemble the `passio.asm` program and load it with the Debugger. Set the `INPUT` to be a file with one non-zero value followed by at least *N* zeros, and the `OUTPUT` to be an initially empty response file. Make sure the initial state of the filter is all zero, either by using the Debugger to clear the memory (`CHANGE` command) or by including instructions in your `my_init` subroutine. Run the program, close the I/O files (`INPUT OFF` and `OUTPUT OFF`), and check the results. Proceed with debugging if the results are incorrect or if your program malfunctioned in some way. Include your code files (with COMMENTS) and the results of your non-real time testing (I/O comparisons, response plots, and so forth).

⇨Exercise G: Real time implementation and testing of your FIR filter

Now that the code is working, write new versions of your subroutines that will work in real time using a program based on a modified version of `pass1.asm` (from Lab #1), `process_stereo` subroutine, and the coefficients and `firfilter` subroutine. You must now use two filters, one for the left stereo channel

and another for the right channel. The two filters can share the same instructions (subroutine) and the same block of coefficients, but they must have separate data blocks (filter definition and filter state).

You will test the real time implementation of the filters in three different ways.

(1) Using an oscilloscope for measurement and a signal generator for input, design a means for experimentally measuring the magnitude and phase of the filter frequency response. Then verify that the filter reproduces the results expected from MATLAB. Also, determine the -3dB cutoff-frequency for your filter.

(2) Again, using the signal generator and a scope, design a means for experimentally measuring the step response of the filter. Make an accurate sketch and comment on symmetry.

(3) Now use music for an input and comment on what you hear.

Additional exercise for Graduate Students:
Repeat (1) – (3) above for a real time implementation of a high-pass filter. Choose a filter length of 65 and a reasonable cutoff frequency, then use `fir1` in MATLAB to generate the coefficients. Include your parameters and results in your report. Did you have to re-write any of your code?

Report and Grading Checklist

A: MATLAB adcdemo and dacdemo

Aliasing observations and signal plots from adcdemo; written discussion.
Reconstruction behavior observations from dacdemo; written discussion.
Comments.

B: Instructions available in one sample period

Code segment from 'process_stereo' modified to test instruction limit.
Measured maximum instruction count in one sample period and PLL setting.
Comments.

C: Delay line implementation

Code segment from lab2_p.asm, with COMMENTS.
Responses to questions (1)-(3).
Additional exercise for graduate students.
Comments.

D: Non-real time testing with PC file I/O

Code segment from process_mono, with COMMENTS.
Discussion of file I/O procedure and results.
Comments.

E: MATLAB FIR filter design

Filter design properties and MATLAB results to verify your design.
Comments.

F: Non-real time testing of FIR filter

Code segment of 'lab2_io.asm' and output impulse response.
Frequency response measurements, etc.
Comments

G: Real time testing of FIR filter

Code segment of real time filter program.
Test results (1)-(3), and discussion.
Additional exercise for graduate students.
Comments

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 and B with results and discussion.
C-	Exercise C with results and discussion.
C+	Exercise D with complete results and comments.
B	Exercises E and F with full MATLAB results verifying the design.
A	Results for Exercise G.

Note: grad student grading also requires the additional exercises from the C and G sections.