ECEN4002/5002   Digital Signal Processing Laboratory

Spring 2002

Laboratory Exercise #1

Introduction

In this first lab you will create some simple DSP programs, use the EVM boards to load, run, and debug your software, and begin to study how signal quality is affected by sampling rate, the conversion processes, and word length. For this exercise you will produce some simple versions of the 'process_stereo' subroutine within the source file "pass.asm" to enable real-time audio experiments on signal quality. Although this lab requires modifications to this source file, it is not necessary to include a listing of the entire file with your report: just plan to include listings of the code you modify or create from scratch.

There are two different Motorola EVM (evaluation module) boards available in the lab: the DSP56303EVM and the DSP56307EVM. Both boards have a Motorola 56300-series DSP microprocessor and various peripheral chips for analog and digital signal I/O. Either board is fully satisfactory for use in this course, although there are a few differences that require different initialization and control software. For this reason, just be sure to choose the correct software for the particular board you use.

The EVM software is located on the local hard drive of the lab computers. There is one folder with the skeleton and example code for the '303 (c:\evm56303) and another folder with the code for the '307 (c:\evm56307). Documentation for the boards and the DSP chips is available on the lab computers, too. Look under the Windows "Start | Programs | Motorola DSP Software Development Tools | DSP56300 SW Tools | DSP56300 Documentation" tab for the Motorola documentation.

The EVM boards use a Crystal Semiconductor (Cirrus) 16-bit multimedia audio codec for the A/D and D/A conversion. The '303 board has a CS4215 chip, while the '307 has a CS4218 chip. The CS421x is a single-chip, stereo, CMOS multimedia codec that supports CD-quality music. The A/D and D/A converters are 64×oversampled delta-sigma converters with on-chip filters, which adapt to the sample frequency selected. The sampling frequency can range from 4 kHz to 50 kHz with typical operation at 48 kHz or 44.1 kHz. Some features of the codec include programmable attenuation for the analog outputs, microphone and line-level analog inputs, headphone, speaker and line-level outputs, on-chip anti-aliasing / smoothing filters and a serial digital interface. The EVM only uses the line-level inputs and outputs along with the headphone output. You may find it useful to bring along some headphones for listening to the output of the EVM while it is executing your programs. The EVM board comes with further documentation on the codec, which can be useful for determining the timing and types of interrupts that are generated by the A/D and D/A operations. For most of this course it will not be necessary to alter the setup and initialization of the codec chip, so you will simply be able to use the standard initialization instructions provided in the "pass.asm" example program.

In this lab you will write some simple routines to adjust the amplitude (gain) of an audio signal passing through the DSP, and to alter artificially the quantized resolution and sample rate of the digital data. Along with writing these routines you will begin to learn how to work with the EVM board, the development software, and the somewhat esoteric debugging process required for real time DSP software.

This first experiment is relatively short, so be sure to take some time looking at the software provided by Motorola with the EVM board that are included in "pass.asm" for handling the A/D and D/A I/O of our audio signals. Later in the semester it may be very convenient to modify these programs, or replace them with our own. Consequently, you are encouraged always to write your code in a highly modular fashion. Plan IN ADVANCE to make your code easy to modify later: keep routines short and simple, and include comments liberally.

Getting set up and tested

In order to get started in the lab, you will need to get the hardware and software configured properly.

1. Choose a lab station and an EVM board. Get an oscilloscope, signal generator, and speakers.
2. The EVM board is attached via a serial data cable to the COM1 port on the PC.
3. The EVM requires DC power, so plug in the "wall wart" transformer and hook up the power connector.
4. Now attach audio cables to the IN and OUT jacks on the EVM. The IN signal can come from the sound card in the PC or from a signal generator. The OUT signal should go to the loudspeakers and/or the oscilloscope. ALWAYS make sure the signal levels are appropriate: start quiet and only raise the level when you are sure of the connections!

Please leave the equipment set up on the lab bench: there is no need to dismantle everything when you leave!

Next, make a copy of the software to your personal folder (Z: drive). You will need to copy the files that match the EVM board you are using (`c:\evm56303` or `c:\evm56307`). The files to copy are:

```
Ada_equ.asm      Ada_init.asm  Vectors.asm
Intequ.asm       Ioequ.asm
Pass.asm         Pass.cld
```

The `.asm` files are plain text files containing the assembly language instructions for the DSP. The `.cld` file is the assembled and linked image that is ready to be downloaded to the EVM.

Now try running the "`pass.cld`" program using the Domain Technologies EVM5630x software.

1. On the computer, go to the "Start | Programs | EVM5630x" menu and execute the EVM5630x program. If the program launch gives an error, make sure your EVM board is properly attached to COM1 and that power to the board is on.
2. Once the Debugger is running, press the STOP button (if necessary).
3. Select the "File | Load…" menu option, then browse to the `pass.cld` file that you copied to your Z: drive, and load it.
4. Now press the GREEN ARROW button to start the program.
5. Start playing some audio (use Media Player or a CD).
6. Make sure the input signal is being passed through to the output signal. If you don't hear anything, double-check the cables and signals. Then try stopping, resetting, and restarting the EVM using the Debugger controls.
7. To stop the EVM, press the STOP button.

⇨ **Exercise: System Frequency Response Measurement**

With "`pass.cld`" loaded and running, drive the EVM input with a sinusoid. Display the output on the scope and take enough data to plot the frequency response magnitude. From the frequency response, estimate the sampling frequency experimentally. (The sampling frequency can be changed by changing one of the parameters in the beginning of `pass.asm`, but we won't be doing that in this experiment). Pay attention to frequencies in the vicinity of $f_s/2$. Is there any distortion because of aliasing? Determine the amount of attenuation for sinusoids well within the pass-band, i.e., having frequencies significantly less than $f_s/2$. Finally, using a low amplitude sine wave give a rough estimate of the typical noise level generated within the EVM board and its container. Include these results and discussion in your written report.

Edit, Assemble, Load, and Run

Now that the hardware is working and you have been able to download and run the `pass.cld` module, the next step is to go through the process of editing and assembling a program.

A. Make a copy of `pass.asm`, and call it `pass1.asm`.
B. Using a plain text editor (DOS Edit, Notepad, etc.), open `pass1.asm` and make the changes as follows:

If you are using the 56303EVM version:

1. Find and then delete the lines
    ```
    process_stereo
            nop
            nop
            nop
            rts
    ```

    and replace them with a single line:
    ```
        include 'lab1_p.asm'
    ```

    The include directive will tell the assembler to insert the text from a file called `lab1_p.asm` at that point. You will then be able to create the source file (called "`lab1_p.asm`", of course) containing your own version of the `process_stereo` subroutine.

2. Next, you can change the line in `pass1.asm`:
    ```
    TONE_OUTPUT     EQU     HEADPHONE_EN+LINEOUT_EN+(4*LEFT_ATTN)+(4*RIGHT_ATTN)
    ```

    by replacing the two "4" digits with "1" digits. This will set the output gain to be maximum.

If you are using the 56307EVM version:

1. Find and then delete the two lines near the end of the file
    ```
            nop                 ; pass data straight through
            nop
    ```

    and replace them with a single line:
    ```
            jsr     process_stereo
    ```

2. Now insert a new line below the line:
    ```
            jmp     loop
    ```

    that is the *include* directive:
    ```
            include 'lab1_p.asm'
    ```

    This include directive will tell the assembler to insert the text from a file called `lab1_p.asm` at that point. You can then create the source file (called "`lab1_p.asm`", of course) containing your own version of the `process_stereo` subroutine.

❋Remember to save your modified `pass1.asm` !

C. Now use the editor to create your new source code file `lab1_p.asm`. Start by just having a simple "do nothing" subroutine in the file:

```
process_stereo
        nop
        nop
        rts
```

You will modify this shortly. But first, save the file and try assembling the program.

D. Open a command window, go to the Z: drive where you saved the source files, and run the assembler command:

```
asm56300 –a –b –g –l  pass1.asm
```

Or, you may want to make a batch file called `asm.bat` containing the single line:
```
asm56300 –a –b –g –l %1
```

This way you can use the command `asm pass1.asm` without having to remember the option flags.

The assembler should create a load file called `pass1.cld`. You may see some WARNINGS, but not any ERRORS. If possible, go ahead and fix the WARNINGS. If there are ERRORS, you need to find and fix the problem in your source files. Once the code assembles error free, use the Debugger software to download `pass1.cld` to the EVM (File | Load…). This should give the same result as the original, unaltered `pass.cld` module. Verify that it works.

Now, let's do some signal processing!

DSP Program #1: Gain Multiplier

In `pass1.asm`, the subroutine `process_stereo` (located in your `lab1_p.asm` file) is called once for every stereo sample pair (left, right). The sample from the left stereo channel is stored in the A accumulator and the sample from the right stereo channel is in the B accumulator. After doing "something" with the data, the resulting output left and right samples need to be placed into A and B, respectively, before the `rts` (return from subroutine).

⇨Exercise: Gain Program

Edit your `lab1_p.asm` file and write some assembly instructions for the `process_stereo` subroutine that change the amplitude of both the left and right signals. Write the code in such a way that it will be possible to change the gain multiplier using the Debugger. Your program will look something like:

```
        org   x:$200
gainval     dc    $7fffff      ; Gain factor (use Debugger to change
                               ;  value at location x:$200).
        org   p:
process_stereo
        move  x:gainval, x0    ; Get the gain value from its location
                               ;  in X memory and put it in multiplier
                               ;  input register x0.
        move  a, y0            ; Get the left input sample from A into
                               ;  the y0 register.
        mpy   x0,y0,a          ; Multiply the sample value by the gain
                               ;  value and put result in A.
…etc…etc…etc…
        rts
```

Note that placing `gainval` at x:$200 is not important: we just need a location in memory that is not being used for something else.

Once your program is ready, assemble it, resolve any errors, then use the Debugger to download it to the EVM. First, verify that you can alter the gain factor using the Debugger: stop, do "CHANGE …" in the command window, then resume execution. Also verify that the change in signal amplitude is as expected. Include the code listing and results in your written report.

Additional exercises for Graduate Students:

Create another program that multiplies the left signal by `gainval` and the right signal by
(`1.-gainval`). Varying the single parameter `gainval` should act like a linear "balance" control. Include the code listing and your description as part of the written report.

Next, modify the "balance" program so that `gainval` is updated automatically in the code to go linearly from 0.0 to ~1.0 and back to 0.0 repeatedly with a period of about 2 seconds. *Hint:* use your knowledge of the sample rate to determine how many increments are needed to take `gainval` from 0 to ~1.0 in 1 second, and how big the increment should be. Include the code listing and description with your report.

DSP Program #2: Word Length and Signal Quality

The DSP5630x uses fixed point arithmetic. The quantization error is normally far too small to be detected since the word-length is at least 16 bits, which is commercial audio quality. What word length is necessary to pass music and speech without noticeable degradation? This question can be answered by masking out the lower bits (least significant bits), which artificially simulates the effects of fewer bits of resolution (larger quantization steps). One can also hear the simulated "quantization error" by masking out the upper bits instead, and listening to the result.

⇨ Exercise: Software Questions and Quantization

In order to explore word-length effects, modify `lab1_p.asm` to use x:$200 to store a masking value and set it to initialize to 24 bits as follows: (Note, again, that the address $200 is not particularly important, you just need an empty place in memory somewhere reasonable to hold and modify the mask value.)

```
        org    x:$200
mask    dc     $ffffff
```

Then rewrite the subroutine `process_stereo` more-or-less along the following lines:

```
        org    p:
process_stereo
        move x:mask,x0
        and x0,a          ; (1)
        nop               ; (2)
        move a1,a         ; (3)
        and x0,b          ; (1)
        nop               ; (2)
        move b1,b         ; (3)
        rts
```

Questions:
(1) One expects to see the instructions labeled (1). How do they affect the three parts of the accumulators?
(2) Why are the nop's present?
(3) What would go wrong if the instructions labeled (3) were not there?

Unfortunately, this snippet of code will not work properly on typical audio signals. The reason is that the registers are 56 bit twos complement numbers. Consequently, truncating (masking) negative numbers has different properties than truncating positive numbers. For example, if accumulator A contained the value

(a)=$00,333333,333333 representing the value 0.399999999999999,

and the mask contained the value $F00000 then the results of the masking would be the number

(a)=$00,300000,000000 representing the value 0.375000000000000.

Now consider setting the accumulator to the bit-wise complement of $00,333333,333333,

(a)=$FF,CCCCCC,CCCCCC representing the value -0.400000000000006.

In this case, the masking—with sign extension—would produce

(a)=$FF,C00000,000000 representing the value -0.500000000000000. Note that the magnitude of the negative two's complement number *increases* due to the truncation, while for a positive number the magnitude *decreases* due to truncation.

However, if the masking is done symmetrically, then the result would be

(a)=$FF,D00000,000000 representing the value -0.375000000000000.

So, you must modify the above code so that negative numbers are handled symmetrically. Try several word lengths and listen to the effect. Use music or speech for your audio input. Include in your report the effects of the following masking values (recall that some of these values explore the "quantization error").

mask	word length
$800000	1 bit
$E00000	3 bits
$F80000	5 bits
$BFFFFF	15 bits
$8FFFFF	13 bits
$83FFFF	11 bits

Determine how you must adjust the input signal amplitude so that the signal is properly "scaled" relative to the bits used. For small amplitude signals the most significant bits, except the sign bit, may never be used. If your signal has too small of an amplitude then you will need to adjust the input signal level and redo the observations, focusing on the bits actually being used. Try changing the amplitude and see what happens to the output signal. Remember, quantization is a *nonlinear* phenomenon.

DSP Program #3: Sample Rate and Aliasing

The EVM board prevents aliasing because the cutoff frequency of the input analog anti-aliasing filter in the CS421x codec is automatically set to $f_s/2$. Thus, we cannot hear the effects of aliasing unless we introduce sampling artificially in software. Consider a system that passes the signal through at a rate of f_s samples per second without modification, and further assume that this hypothetical system has no anti-aliasing filter. Suppose we introduced a sinusoidal input. What would we hear as we vary the sample rate and listen to the effect?

With a low sample rate and no anti-aliasing filter you will be able to hear the images that result from sampling. In this part, we want to learn to recognize this sound particularly with music as the input. One of the most useful skills you can have as a DSP programmer is the ability to recognize program bugs from the sound they produce. Add this one to your repertoire now, because you may encounter this sound again when you are developing longer more complicated programs. You may add a few too many instructions and push the execution time beyond one sample period. In that case, the DSP might ignore every other interrupt from the A/D resulting in halving the sample rate. Think about how difficult it would be to identify a problem like this if you can't recognize the sound of aliasing.

⇨**Exercise: Decimation and Aliasing**
We will artificially decrease the sampling rate, in software, by the process of decimation. Modify the subroutine 'process_stereo' by using instructions whose effect is to multiply the input sequence by a periodic sequence, which has period *m* and is zero except for every *m*-th value. In other words, your routine must pass one input sample, then zero out the next *m*-1 samples, then pass another input sample, and so forth. The effective sample rate is now f_s/m. The analog lowpass filter does not know this however, and you are guaranteed to get aliasing.

Conduct an experiment with the signal generator and verify that folding occurs at $f_s/(2m)$ Hz. Include in your report some examples of your observed input and output frequencies while using a sinusoidal input for some "good" value of *m*. As you did with `gainval` and `mask` in the previous programs, write your code so that the rate constant *m* is in X memory where it can be changed by the debugger. This allows you to change the value without reassembling. Initialize *m* with the value of 1 (no subsampling). Increase the value of *m* by 1 each time. Play music through the system and listen to the aliasing using different values of *m*. When *m* is sufficiently high you will start to hear the high frequency artifacts, because the very low frequencies will alias up to the effective sample rate f_s/m. The signal will also have amplitude 1/m times the original (can you figure out why?), so you will have to crank up the amplifier gain a bit. In your report, describe the sound made by aliasing.

Report and Grading Checklist

Pass Program:
        Empirically estimated sample rate.
        Frequency response plot.
        Mid-band gain.
        Noise level.
        Comments

Program #1: Gain modifier
        Code listing for Program #1 (`lab1_p.asm` with COMMENTS!)
        Discussion.
        [Grad students: include your additional listings and comments]

Program #2: Quantization
        Answers to questions 1-3.
        Code listing for Program #2
        Comments on the effects of quantization.

Program #3: Aliasing
        Code listing for Program #3.
        Observations of aliasing (measured input vs. output frequencies, predicted image frequencies, etc.)
        Comments on audible effects of aliasing.

Grading Guidelines
        F   Anything less than what is necessary for a D.
        D   Pass program results, no code listings.
        C-   Two programs; incomplete or bogus results.
        C+   Two programs; mostly complete discussion and results.
        B   All three programs; incomplete results or discussion.
        A   All three programs; complete discussion and results.

        Note: grad student grading also requires the additional exercises in the Program #1 section.