Utah State University
Real-Time Digital Signal Processing Laboratory
ECE 5640

Lab 4: Adaptive FIR Filtering.

Objective

A powerful advantage of the digital filter is that the filter can be changed while it is being used. The purpose of this lab is to gain experience working with an adaptive FIR filter. To implement the filter in real time requires careful attention to timing. Thus, this lab will require the use of circular buffering and linear assembly to gain maximum speed.

Procedure

1. Implement the LMS adaptive filter using the equations given above. Be sure to write the entire processing loop in linear assembly code. (You may use C to set up the OMAP card and sampling codec parameters, then enter an assembly loop for processing. Assume a filter length of 256, and set the initial value of the filter weights to a delta function. Use a sampling rate of 44.1 kHz.

   Keep in mind the following points:

   (a) The circular buffering features of the 'C67 require that the arrays containing the coefficients and delay register must be on proper address boundaries (See Sec. 2.8.3 in the *TMS320C674x DSP CPU and Instruction Set Reference Guide*.) One way to place an array at an absolute location is given by the following example.

   In the example, we use an array named 'array' that provides 1024 floating-point locations. The array is absolutely located at the beginning of the shared RAM block, 0x80000000.

   i. All of the C files wishing to use 'array' must declare it as an extern, as in `file1.c`, shown below:

```
extern float array[1024];

main(ac,av)
{
    int i;

    for(i=0 ; i<1024 ; i++){
        array[i] = 0.0;
    }

            .
            .
            .

}
```

Background

One class of adaptive filters is based on the theory of optimal filtering. The purpose of the filter is to minimize the mean-square error between a *reference*, $x(n)$, and *desired*, $d(n)$, input. This goal gives the filter the name of an *Least Mean Square (LMS) adaptive filter*.

We will not concentrate in this lab on the details of LMS adaptive filtering theory. The general block diagram for the structure used in the filter is given in Figure 1. For a noise

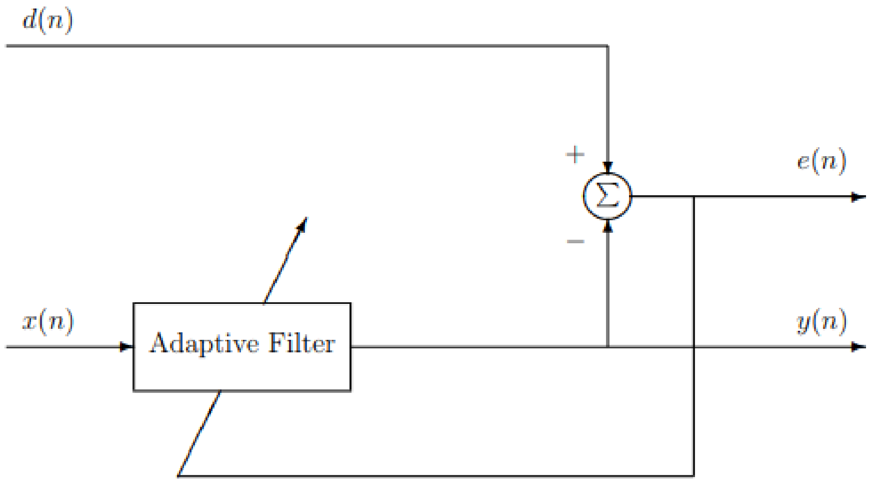

Figure 1: General LMS adaptive structure.

canceller, the desired input $d(n)$ contains the desired signal plus additive noise. The reference input $x(n)$ contains the noise sampled from a different location. The FIR filter attempts to "learn" the noise part of the input signals (the part that is similar in both channels) and remove it, leaving the desired signal as the error signal $e(n)$.

The filter itself is just a simple FIR filter. The weights of the filter, however, are changed over time to cause the filter to "learn." The adaptation algorithm is given as follows. Let the past N values of the reference signal $x(n)$ be used to create a vector $\mathbf{x}(n) = [x(n), x(n-1), x(n-2), \ldots, x(n-N+1)]^t$. The weights of the filter can also be used to create the vector $\mathbf{h}(n) = [h(0), h(1), h(2), \ldots, h(N-1)]^t$. The update for $\mathbf{h}$ is given as:

$$\mathbf{h}(n+1) = \mathbf{h}(n) + \beta e(n)\mathbf{x}(n), \tag{1}$$

where β is a constant that determines the rate of convergence.

The filter is thus used as follows. First, update the signal vector $\mathbf{x}(n)$ by putting the current reference input, $x(n)$, in the first element of the vector after shifting the past inputs down by one. (Note this can be done using a circular buffer.) Next, filter $x(n)$ using the filter weights $\mathbf{h}$ as given by

$$y(n) = \mathbf{x}^t(n)\mathbf{h}(n). \tag{2}$$

Before the next input is received, use the output of the filter, $y(n)$, and the desired signal, $d(n)$, to compute the error

$$e(n) = d(n) - y(n). \tag{3}$$

Finally, the weights are updated using (1). You might want to start with a β of 2^{-30}.

ii. The symbol and space to be used by ‘array’ should be declared in an `.asm` (or `.sa`) file. The section for ‘array’ should be unique. In the example below it is named ‘mysect’ (file: `array.sa`).

```
        .global _array
_array  .usect "mysect", 4096, 4
```

The arguments to the `.usect` directive are the name of the section, the number of bytes, and the alignment (4 bytes per element).

iv. For our current `'C6748` processor, the shared RAM is *not* cached by default. This means that there are several wait states for every memory reference. In order to get your 256-tap filter to run at 44.1 kHz, you *must* place your data arrays in L2 RAM.

v. The files `array.sa` and `file1.c` should be compiled and linked using `CCS`. (Note that the assembly code in `array.sa` can be included with other linear assembly code and does not need to be in a separate file.) The linker will use the location `0x11800000` in the L2 RAM for the address of the label `_array`.

vi. The program ‘file1’ is ready to be loaded and executed.

(b) There are a few things to remember when using the array in an assembly program. The linear assembly examples below can be put into a procedure called from your C program before you enter the processing loop. The set-up for the circular buffer needs to be called only once.

i. You must be sure to declare the array as `.global` in every assembly file that uses it (as shown above).

ii. To use the arrays, you can refer to them directly by using the label as follows:

```
mvkl    _array,A5 ; Move address of array into A5. (Lower half-word)
mvkh    _array,A5 ; Move address of array into A5. (Upper half-word)
```

iii. Note that you can refer to the starting address of any function (such as an interrupt service routine) in a similar manner.

iii. The linker command file should be modified to place the new unique section 'mysect' at the desired address. To do this, you must replace the `.cmd` file generated by *Code composer Studio*. A template `linker_dsp.cmd` file is included on the Canvas website. The steps are as follows:

A. Select the project set up for this lab.

B. Delete the `OMAPL138.cmd` file that was generated when you created the project.

C. Add the `linker_dsp.cmd` template file to your project.

D. To add section for the array, go to the "MEMORY" and "SECTIONS" part in the `linker_dsp.cmd` file and add the respective lines as shown below.

```
MEMORY
{
    SHDSPL2ROM    o = 0x11700000  l = 0x00100000
    ARRAY_DATA_MEM   o = 0x11800000  l = 0x00000400  /*<== ADDED */
    SHDSPL2RAM    o = 0x11800400  l = 0x0003FC00     /*<== MODIFIED */
    SHDSPL1PRAM   o = 0x11E00000  l = 0x00008000
    SHDSPL1DRAM   o = 0x11F00000  l = 0x00008000
                .
                .
                .
    EMIFACS0      o = 0x40000000  l = 0x20000000
    EMIFACS2      o = 0x60000000  l = 0x02000000
    EMIFACS3      o = 0x62000000  l = 0x02000000
    EMIFACS4      o = 0x64000000  l = 0x02000000
    EMIFACS5      o = 0x66000000  l = 0x02000000
    SHRAM         o = 0x80000000  l = 0x00020000
    EXT_DDR2      o = 0xC0000000  l = 0x08000000
}
SECTIONS
{
   .text       > SHDSPL2RAM
   .const      > SHDSPL2RAM
   .bss        > SHDSPL2RAM
   .far        > SHDSPL2RAM
   .switch     > SHDSPL2RAM
   .stack      > SHDSPL2RAM
   .data       > SHDSPL2RAM
   .cinit      > SHDSPL2RAM
   .sysmem     > SHDSPL2RAM
   .cio        > SHDSPL2RAM
   .vecs       > SHDSPL2RAM


.EXT_RAM     > EXT_DDR2
          .
          .
          .
 mysect      > ARRAY_DATA_MEM    /* <== ADDED to locate mysect */

}
```

Please note that this example places the array in the L2 RAM. If you want, you could also modify the `.cmd` file to place the array in the shared RAM at 0x80000000.

(c) You need to set up the Addressing Mode Register (AMR) to use circular buffering. The following code sets up the AMR to use a block size of 16 *bytes* with the circular buffer pointer in register B5:

```
mvk     0x0400,B0 ; Set bit 10 for B5 as circular buffer pointer
mvklh   0x0003,B0 ; Set the BK0 field to 3 (block size of 16 bytes).
mvc     B0,AMR    ; Move the word to the AMR
```

(d) When performing arithmetic operations on addressing registers, you must remember that the `ADDAH/ADDAW/ADDAB` and `SUBAH/SUBAW/SUBAB` instructions should be used instead of the `ADD` and `SUB` instructions. The latter instructions do not update the addresses in a circular manner.

(e) There are a few assembler optimizer directives that are useful. These are given in Table 1. For further detail and additional directives, consult the *TMS320C6000 Optimizing Compiler User's Guide*.

Directive	Description	Restrictions
.mdep [*symbol1, symbol2, ...*]	Indicates a specific memory dependence	Valid only within procedures.
.no_mdep	No memory aliases	Valid only within procedures.
.reserve [$register_1, register_2, ...$]	Reserve register use	Valid only within procedures.
.circ [$symb_1/reg_1, symb_2/reg_2, ...$]	Declares circular addressing	Valid only within procedures. Does not set up the buffer.

Table 1: Linear Assembly Directives

Sometimes the assembly optimizer removes code that modifies registers which are not used in the procedure. If the registers are used *outside* of the procedure (such as in an ISR), the **.reserve** directive can be useful to prevent the assembly optimizer from removing the code.

(f) This lab requires you to use very efficient code, enable the program cache, fully optimize, and place the filter weights and delay buffer in the on-board RAM to be able to compute 256 taps in the adaptive filter. Some ideas for improving the speed of your code are:

- Interleave the filter and filter update equations so that the delayed samples and coefficients are fetched from memory only once.
- Use double-word loads `LDDW` to get two samples at a time. The `LDDW` command requires that words are aligned on 4-byte boundaries (i.e. 0x10, 0x14, etc). If the words are not aligned on 4-byte boundaries (i.e. 0x12, 0x16, etc) which commonly happens when the C compiler allocates memory for variables, the `LDDW` command will load parts of your words (i.e. `LDDW` loads bytes 0x10-0x18 but your actual words were 0x0E-0x11, 0x12-0x15, 0x16-0x19). This leads to very confusing problems. The best way to fix this is to manually allocate memory space for your variables in the linker command (`.cmd`) file.
- Place the arrays to eliminate memory fetch conflicts.

(You may not need to do any or all of these.)

2. Test your adaptive filter using the following signals. You will be provided with a stereo file (`LMS-300Hz_sine_350Hz_square.wav`) with two signals, one in the left channel, and one in the right channel. The left channel will contain just the "noise" signal (a 350 Hz square wave), and the right channel will contain the noise signal and the desired signal (a 300 Hz sine wave).

3. Download your program and watch the spectrum analyzer. Does the output change to show mostly the 300 Hz sine wave? What are the magnitudes of the 300 Hz sine wave and the fundamental of the 350 Hz square wave? What is the reduction in magnitude of the 350 Hz signal? Plot the spectrum.

4. Repeat the previous procedure with $\beta = 2^{-25}$ and $\beta = 2^{-35}$. How does the time for convergence compare?

5. Repeat procedures 2 and 3 with a 2 kHz sine and 1.9 kHz square wave found in the file `LMS-2kHz_sine_1.9kHz_square.wav`.

6. To see the effectiveness of the cache, place your filter and sample arrays in the shared RAM (`0x80000000`). How low must the sampling rate be to get your filter to work?

7. A test file with white noise obscuring a desired signal is found in the file `LMS-white-noise.wav`. Filter this signal with your adaptive filter and determine if you can hear the desired signal.

Questions

1. Comment on the advantages and disadvantages of circular buffering in C verses in linear assembly.

2. Did your filter "learn" the noise (square wave) signal and filter it out?

3. What can you say about the effect of the parameter β on the rate of convergence?

4. Did the change in frequencies change the ability of the filter to adapt?

5. What is the effect of the cache (on-board RAM) on performance?

6. Were you able to hear the desired signal buried in noise?