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

Lab 3: IIR Filtering

Objective

The purpose of this lab is to compare the implementation of an infinite impulse-response (IIR) filter using different structures.

A second purpose of this lab is to learn to use mixed C and linear assembly in the filter program.

References

1. *Digital Signal Processing Principles, Algorithms, and Applications* by Proakis and Manolakis, Section 9.3.
2. *OMAP-L138 Datasheet*, Texas Instruments.
3. *OMAP L138 Technical Reference Manual*, Texas Instruments.
4. *TMS320C674x DSP CPU and Instruction Set Reference Guide*, Texas Instruments.
5. On-line documentation for *Code Composer Studio*.

Required Equipment

The following equipment is necessary to complete the lab:

1. The Texas Instruments OMAP-L138 Experimenter Kit.
2. An Analog Discovery soft instrumentation board.
3. Cables for connecting the system to the Analog Discovery board.

Required Software

The following software is necessary to complete the lab:

1. The Texas Instruments *Code Composer Studio*
2. The OMAP-L138 Experimenter board support library (BSL): `evmomapl138_bsl.lib`.
3. The `.zip` archive file `template2.zip`. This is a new set of template files that use interrupt-based timing. You should look at the difference between this and the polling method used in the previous labs.
4. `Matlab`.

Background

IIR filtering can be performed using various filtering structures. Among these are the cascaded direct form II and lattice-ladder structures. Each has advantages and disadvantages when considering number of arithmetic operations, memory requirements, and round-off error.

There are several aspects of the 'C67 you must be familiar with to successfully complete this lab. They are briefly described in the following:

1. To make your filter routine callable from C, you must use the register convention of the C compiler. Table 1 illustrates the registers used for passing arguments into and out of the routine. In this table the memory between even registers is used to pass double-precision values. Argument 1 and the return value is stored in `A4`, the return address is stored in `B3`, `DP` is the base pointer which points to the beginning of the `.bss` section, and the `SP` is the stack pointer that points to local variables.

Register	Function Preserved By	Special Uses	Register	Function Preserved By	Special Uses
A0	Parent	–	B0	Parent	–
A1	Parent	–	B1	Parent	–
A2	Parent	–	B2	Parent	–
A3	Parent	Structure register (pointer to a returned structure)	B3	Parent	Return register (address to return to)
A4	Parent	Argument 1 or return value	B4	Parent	Argument 2
A5	Parent	Argument 1 or return value with A4 for doubles, longs and long longs	B5	Parent	Argument 2 with B4 for doubles, longs and long longs
A6	Parent	Argument 3	B6	Parent	Argument 4
A7	Parent	Argument 3 with A6 for doubles, longs, and long longs	B7	Parent	Argument 4 with B6 for doubles, longs, and long longs
A8	Parent	Argument 5	B8	Parent	Argument 6
A9	Parent	Argument 5 with A8 for doubles, longs, and long longs	B9	Parent	Argument 6 with B8 for doubles, longs, and long longs
A10	Child	Argument 7	B10	Child	Argument 8
A11	Child	Argument 7 with A10 for doubles, longs, and long longs	B11	Child	Argument 8 with B10 for doubles, longs, and long longs
A12	Child	Argument 9	B12	Child	Argument 10
A13	Child	Argument 9 with A12 for doubles, longs, and long longs	B13	Child	Argument 10 with B12 for doubles, longs, and long longs
A14	Child	–	B14	Child	Data page pointer (DP)
A15	Child	Frame pointer (FP)	B15	Child	Stack pointer (SP)
A16-A31	Parent	C6400, C6400+, and C6700+ only	B16-B31	Parent	C6400, C6400+, and C6700+ only
ILC	Child	C6400+ and C6740 only, loop buffer counter	NRP	Parent	
IRP	Parent		RILC	Child	C6400+ and C6740 only, loop buffer counter

Table 1: Registers Used for Passing Arguments. (Source: Texas Instruments)

Note that the C routine calling the assembly function automatically stores the values in the registers labelled "Parent," and the assembly routine is responsible for storing any values overwritten in the registers labelled "Child."

2. Linear assembly is a Texas Instruments language which is very similar to assembly, but which does not require you to specify functional units, registers, or `NOP`s. It is very useful because it allows near-handcoded execution speeds, but is faster and simpler to write than true assembly. Since it is not the goal of this lab to teach programming skills that are highly specific to the TI VelociTI architecture, we will use linear assembly to learn assembly programming, but avoid issues related to the 'C6x functional units and pipeline.

   An example of a **fixed-point** dot-product routine, taken from [1, p. 163], is given in Figure 1. The C prototype declaration for the dot-product is given by

   ```
   short dotp(short *m, short *n, short count);
   ```

   The main points to learn from this example are:

   (a) Linear assembly filenames should end in the extension `.sa`.

   (b) The code for this procedure in this example will be linked into the section `code`. This section must have a corresponding entry in the linker command file. Remember that the string in the quotes must match the string used in the linker command file. In your program, it is sufficient to link into the `.text` section, which is a default section always provided by the linker.

   (c) The `.global` directive states that this is a globally defined function.

   (d) The `.proc` directive states the registers used for passing values into the procedure. In the example, `A4` is used for the address of the first array, `B4` is used for the address of the second array, and `A6` is used for the number of points in the dot product. The `B3` register, referred to as a preserved register, is not to be modified, and contains the return address of the procedure. Preserved registers must be specified in both the input and output arguments even when they are not used within the procedure.

   (e) The `.endproc` directive states the registers used for passing values out of the procedure.

   (f) The `.reg` directive declares the variables used in the procedure.

   (g) The `.trip` directive is used for the assembly optimizer to generate pipelined code. Pipelined code will not be generated without it. It must be the minimum iteration count used in the procedure.

   (h) The result of the procedure is returned, by convention, in register `A4`.

3. There is a temptation to use register names (e.g. `A4` or `B4`) in the assembly instructions. This is not incorrect, but it restricts the linear assembly optimizer from freely using registers in an optimal way. Note that in Figure 1, the register names are used only to get the values passed into the routine and put values into the return register. Variables used instead of register names allow the optimizer to re-use registers if it will speed up execution.

```
        .title "dotp.sa"
        .def   _dotp
        .sect  "code"
        .global _dotp
_dotp:  .proc  A4, B4, A6, B3
        .reg   p_m, m, p_n, n, prod, sum, count
        mv     A4, p_m          ;p_m now has the address of m
        mv     B4, p_n          ;p_n now has the address of n
        mv     A6, count        ;count = the number of iterations
        mvk    0, sum           ;sum=0

loop:   .trip  40               ;minimum of 40 iterations through loop
        ldh    *p_m++, m        ;load element of m, postincrement pointer
        ldh    *p_n++, n        ;load element of n, postincrement pointer
        mpy    m, n, prod       ;prod=m*n
        add    prod, sum, sum   ;sum += prod
[count] sub    count, 1, count  ;decrement counter
[count] b      loop             ;branch back to loop

        mv     sum, A4          ;store result in return register A4
        .endproc A4, B3

        b       B3              ;branch back to address stored in B3
        nop     5
```

Figure 1: Linear assembly code for integer dot product

4. At this point you are ready to experiment with the C/assembly optimizer. The typical development flow for real-time applications follows the following order. A flow chart of these ideas is given in Figure 2.

   (a) Write and debug the application for functional correctness. The code should be able to execute correctly and provide the appropriate results. This step is usually done with the C/assembly optimizer set with *no optimization*. This is accomplished by using the `Debug` configuration.

   (b) Turn on the optimization and check for correct functionality. This is the stage where you are trying to meet real-time processing constraints. If the application does not run in real-time, increase the optimization level and recheck. This is accomplished by using the `Release` configuration.

   (c) Write critical timing sections in (linear) assembly. You can take advantage of hardware such as circular buffers, cache configurations, bit-reversed indexing, block or instruction repeats, etc., to increase execution speed.

   (d) If the application will still not run in real-time, you must resort to hand-tuned assembly, taking particular care to efficiently use the pipeline.

   In our labs, we will not resort to hand-tuned assembly and pipelining, but you should always use C/assembly optimization in the compiler once the code is functionally correct.

Procedure

1. To become more familiar with the assembly language of the 'C67 processor, convert and write down the dot product routine given in Figure 1 to a *floating-point* dot product. (Do *not* assemble or debug the routine.) This will help you become familiar with the assembly instructions for floating-point operations.

2. Create an IIR digital filter using `Matlab`. The filter should be a Type I bandpass Chebyshev filter of order 8, implemented in cascaded second-order sections (4 second-order sections). Make the passband 3 kHz centered at 7 kHz, the passband ripple to be 0.5 dB, and the sampling frequency 48 kHz. Use the `Matlab` tool `fdatool` to design the weights for the filter. You must then save the weights to a file for download to the 'C67.

   The order of the floating-point values in the file are:

   $$b_{01}, b_{02}, b_{03}, ..., b_{0L}, ..., b_{21}, b_{22}, b_{23}, ..., b_{2L}, a_{01}, a_{02}, a_{03}, ..., a_{0L}, ..., a_{21}, a_{22}, ..., a_{2L}$$

   for an L-stage filter. Each stage is defined as

   $$H_k(z) = \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2}}, \tag{1}$$

   where the a_{0k} coefficients are typically set to 1.0. Note that any scaling factor is contained in a *fifth* second-order section. (Check the output file to verify this.)

   Check the frequency response of the filter using `Matlab`. Plot the frequency response.

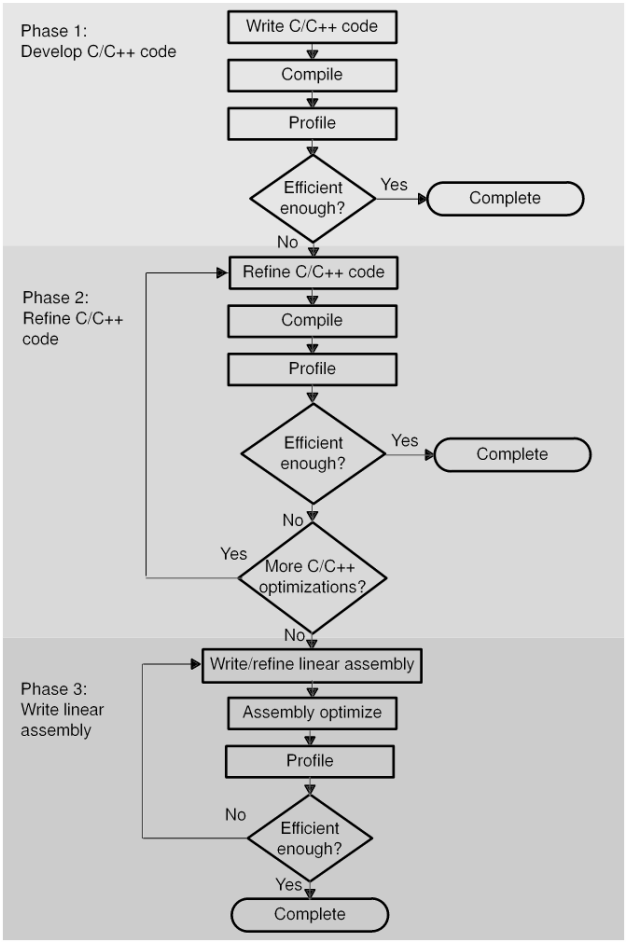

Figure 2: TI suggested code development flow. (Source: Texas Instruments)

3. Write a program that implements a direct form II IIR filter using cascaded second-order sections. Your program should be able to filter with the filter coefficients designed in procedure 2, using linear assembly code to perform the computations for the actual filtering of the data. (You may use C for the rest of the program, i.e., the data input and output.) Carefully comment your code to describe its execution. Remember to link the object files you will need.

   Please remember that you may *not* use the assembly code downloadable from the TI site, but must write your own code.

   The filter equations for the second-order section (often called a *biquad*) with the transfer function given in (1) are given by

$$y_k[n] = -a_{1k}y_k[n-1] - a_{2k}y_k[n-2] + b_{0k}x_k[n] + b_{1k}x_k[n-1] + b_{2k}x_k[n-2]. \quad (2)$$

   For direct form II, (2) is implemented using the following pair of equations:

$$\begin{aligned} d_k[n] &= x_k[n] - a_{1k}d_k[n-1] - a_{2k}d_k[n-2] & (3)\\ y_k[n] &= b_{0k}d_k[n] + b_{1k}d_k[n-1] + b_{2k}d_k[n-2]. & (4) \end{aligned}$$

   Note that the storage requirements are smaller for these equations since the d_k values are used twice.

   An efficient way to store the necessary values in memory is to store the biquad coefficients in an array in the following order: $a_{2k}, b_{2k}, a_{1k}, b_{1k}$, and b_{0k}. This allows you to load the d_k values only once for two operations.

   Save and print the assembly code created by the compiler for the C portion of the program (the subroutine) which calls your assembly routine. Also, save and print the assembly code created by the assembly optimizer.

4. Download and boot the IIR filter. Check the frequency response of the filter by using the white noise output of the Analog Discovery as input to the filter. Observe the spectrum of the output using the Analog Discovery spectrum analyzer. Plot the frequency response. Is it what you expected?

5. Finally, design a lattice-ladder filter using coefficients computed from the filter from procedure 2. (Again, remember that you may *not* use the assembly code downloadable from the TI site, but must write your own code.) Implement it on the 'C67, using linear assembly code to perform the computations for each stage. *Remember that you need to begin with the transfer function defined as ratio of polynomials instead of a product of second-order sections.* Plot the frequency response. Is it the same as in Procedure 4?

   Save and print the assembly code created by the assembly optimizer.

Questions

1. What are the memory requirements and computational requirements of the two implementations in number of locations and number of computations?

2. How do the two frequency responses compare?

3. Which do you think is the "best" implementation? Why?

References

[1] N. Kehtarnavaz and M. Keramat, *DSP System Design: Using the TMS320C6000*. Upper Saddle River, NJ, 07458: Prentice Hall, 2001.