16

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.

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

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.

List of commands found in Appendix A of the DSP CPU and Instruction Set reference guide.

2/11/19
AP

```
        .title "dotPFloat.sa"
        .def _dotp
        .sect "code"
        .global _dotPFloat
_dotPFloat:
        .proc A4, B4, A6, B3 ;(*a,*b,count)
        .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
        ldw *p_m++, m ;load element (32bit word) of m, postincrement pointer
        ldw *p_n++, n ;load element (32bit word) of n, postincrement pointer
        mpysp m, n, prod ;prod=m*nsingle precision float
        addsp prod, sum, sum ;sum += prod      single precision float
[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
```

```
float dotPFloat(~~a~~ float *a, float *b, int count)
  float sum=0
  for(~~int i~~; count>0; count--)
     sum=*(a++) * *(b++);
  return sum;
```

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:

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

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.

2/11/19

W

18

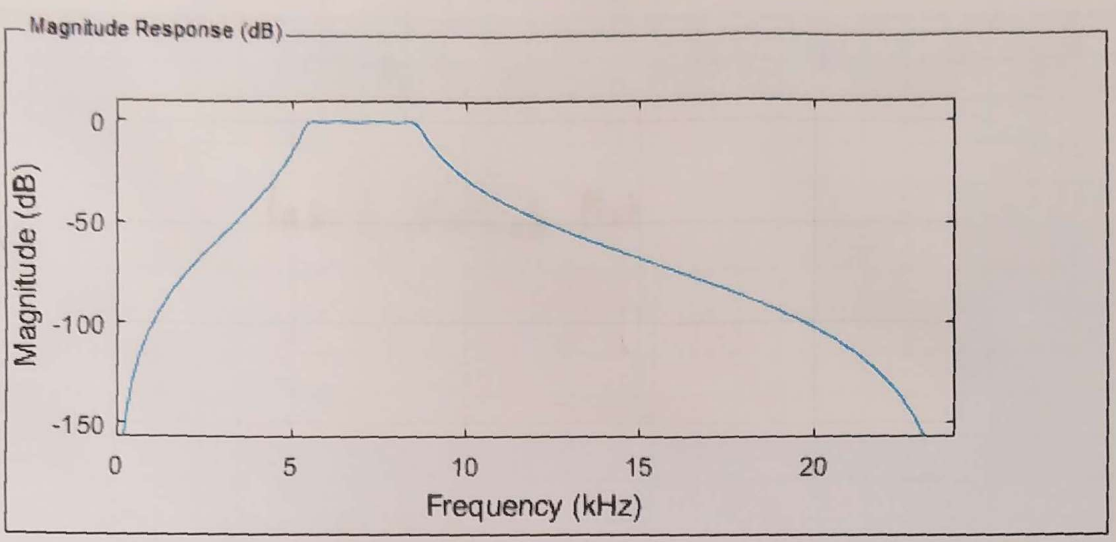


Figure 1 Matlab filter frequency response

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

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

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

$$d_k[n] = x_k[n] - a_{1k}d_k[n-1] - a_{2k}d_k[n-2] \qquad (3)$$

$$y_k[n] = b_{0k}d_k[n] + b_{1k}d_k[n-1] + b_{2k}d_k[n-2]. \qquad (4)$$

   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.

2/13/19
M

Ladder Algorithm

section $d_k[n] = x_k[n] - a_{1k} d_k[n-1] - a_{2k} d_k[n-2]$

$\Downarrow$ $y_k[n] = b_0 d_k[n] + b_{1k} d_k[n-1] + b_{2k} d_k[n-2]$

section (X, dbuff(k)d, *Filter Coef[0]

Filter section: gain, b0, b1, b2, a0, a1, a2

~~movk 3, count~~ mov ~~0~~ 4, count

~~dloop:~~ 2

~~dloo~~ mov X, d

~~floop~~

~~mpy Filter~~ ~~add Filter Coef,~~

add ~~fdw~~ Filter Coef, count, ~~p_f~~ p_a

movk 2, count,

add Filter Coef, count, p_b

store dbuff into p_d

~~floop:~~ movk ~~count~~ 2, count

mov dbuff, p_d

mov X, ~~*dbuff++~~ p_d

for (i=~~0~~ 1; i<3; i++){
filter[i]·dbuff[i]+=dbuff[0]
}

dloop:

mpy *p_a++, ~~dbuff++~~ p_d++, product

sub *dbuff, product, *dbuff

sub count, 1, count

[count] b dloop

mov dbuff, p_d movk 3, count

$Y_k[0] = 0$

yloop:

mpy *p_b++, *p_d++, product

add product, result, result

sub count, 1

[count] b yloop

mpy result, *Filter coef, result ; gain

end sec

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?

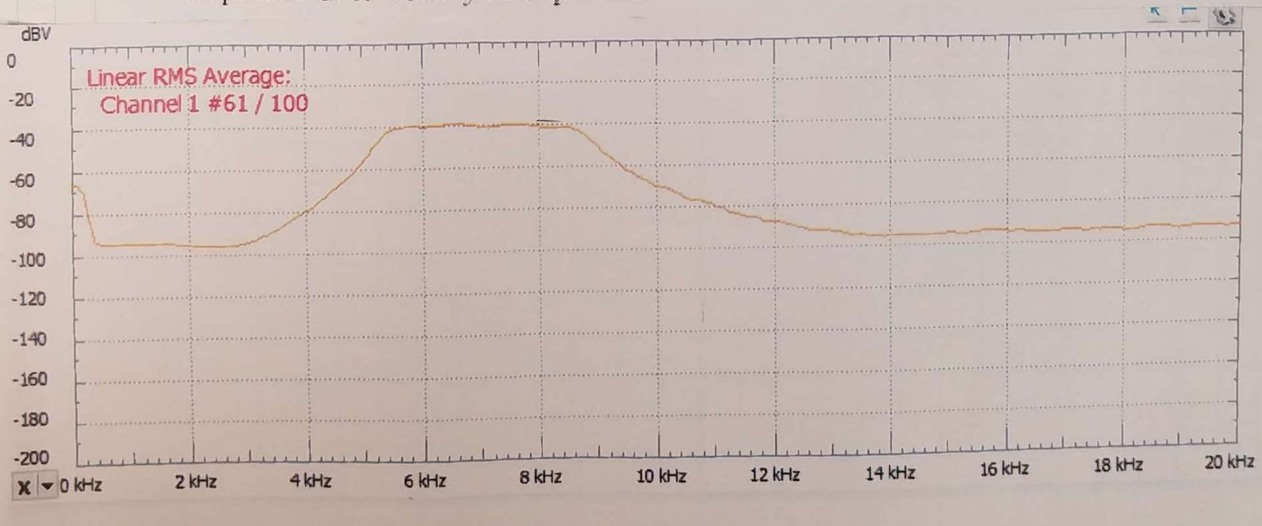


Figure 2 Cascade filter frequency response

doesn't fall off to $-\infty$ dB, but due to base noise in the system, the ~~not~~ frequency response will not fall below -100 dB

2/13/19

NY

Cascade.sa

```
        .title "cascade.sa"
        .def _cascade
        .sect ".cascade"
        .global _cascade
        .global _cascadeSection
        .global _filterSections
        .global _dBuff
        .global _dOffset
        .global _sections

_cascade: .proc A4, B3 ;cascade(x(n))
        .reg x, filter, dBuff, dOffset, sections, p_filter, p_dBuff, i, count,
product, result, gain

        ;move globals into registers
        mvk 0, i
        mv A4, x
        mvkl _dBuff, dBuff
        mvkh _dBuff, dBuff
        mvkl _filterSections, filter
        mvkh _filterSections, filter
        mvkl _dOffset, dOffset
        mvkh _dOffset, dOffset
        mvkl _sections, sections
        mvkh _sections, sections

        ldw *dOffset, dOffset
        ldw *sections, sections

;loop over every section for cascade
loop:
        mpy i, 4, count ;[i][4]
        addaw dBuff, count, p_dBuff ;&dBuff[i][0]
        mpy i,7,count ;[i][7]
        addaw filter, count, p_filter ;&filterSections[i][0]
        .call x = _cascadeSection(x,p_dBuff,dOffset,p_filter)       ;get output of
single section
        add i, 1, i
        sub i, sections, count
[count] b loop

        ;apply output gain
        mpy sections, 7, count ;[i][sections]
        addaw filter, count, p_filter ;obtain address of output gain
        ldw *p_filter, gain ;load the output gain
        mpysp x, gain, x

        mv x, A4 ;return result
        .endproc A4, B3
        b B3
```

2/21/19

M

CascadeSection.sa

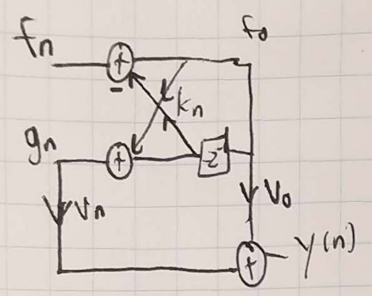
```
        .title "cascadeSection.sa"
        .def _cascadeSection
        .sect ".cascade"
        .global _cascadeSection

_cascadeSection: .proc A4, B4, A6, B6 ;cascadeSection(x(n),*dBuff(n=0),dOffset,filterCoef)
        .reg x, dBuff, filter, p_a, p_b, a, b, d, product, dresult, yresult, count, dOffset,
gain
        mvkl 0x3<<16|0x1<<8, count
        mvkh 0x3<<16|0x1<<8, count
        mvc count,AMR          ;make B4 a circular buffer of size 16(4*wordSize)
        ;move parameters into local registers
        mv A4, x
        mv B6, filter
        mv A6, dOffset
        mvk 0,yresult

        addaw filter, 4, p_a ;a0 address 4*wordLength
        addaw filter, 1, p_b ;b0 address 1*wordLength
        addaw B4, dOffset, B4 ;shift D to d(0) in circular buffer
        mv B4, dBuff ;store initial d(0) location

        ;init d_k to x_k
        ldw *p_a++, a ;load a0;
        mpysp x,a, dresult ;x(n)*a0
        addaw p_b,1,p_b
        mvk 2, count
        ;compute a,b*d_k

dLoop: ;i=1;i<3;i++
        ldw *p_a++, a  ;a(i)
        ldw *++B4, d   ;d(n-i)
        ldw *p_b++, b  ;b(i)
        mpysp d,a, product ;d(n-i)*a(i)
        subsp dresult,product,dresult

        mpysp d,b,product     ;d(n-i)*b(i)
        addsp product,yresult,yresult
        sub count, 1, count
[count] b dLoop

        ;store d[0], calculate y+=d[0]*b[0]
        stw dresult, *dBuff ;store d[0]
        ldw *+filter[1],b ;get b0
        mpysp dresult, b, product ;d[0]*b[0]
        addsp product, yresult,yresult

        ;output gain
        ldw *filter,  gain
        mpysp yresult, gain, yresult ;y*gain
        mv yresult, A4 ;return y

        .endproc A4, B3
        b B3
```

2/21/19
AM

22

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.



for m = filter_size-1, m--
{
kval = k[m]
vVal = v[m]
g = gold[m-1]
$f_{m-1} = f_m - kval \cdot g$
$g = k[m] \cdot f_{\cdot} + g$
output += vVal · g
gold[m] = g
}
gold[0] = f
output += f · v[0]
return output

had issues with output.
got strange interrupts and the code kept repeating when it shouldn't have, while trying to check values.
Ended up just letting the system run in release and it worked
No functionality was changed from the last time I ran in release

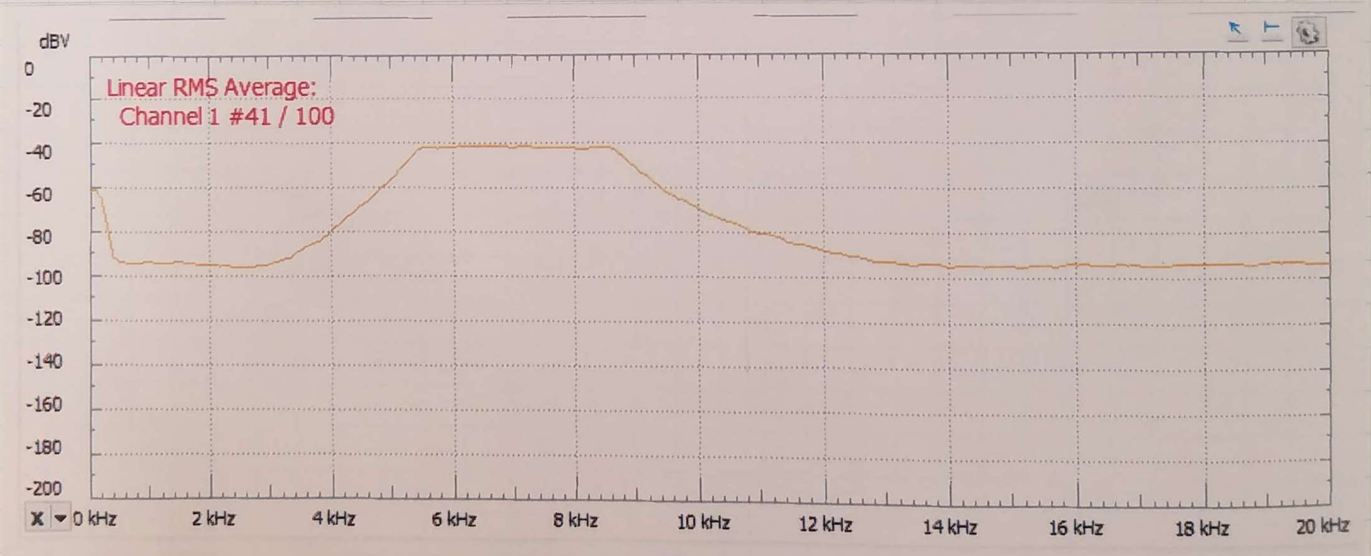


Figure 3 Lattice-Ladder filter frequency response

2/21/17

LatticeLadder.sa

```
        .title "latticeLadder.sa"
        .def _latticeLadder
        .sect ".lattice"
        .global _latticeLadder
        .global _filterLength
        .global _kVal
        .global _vVal
        .global _gOld


_latticeLadder: .proc A4,B3 ;latticeLadder(x(n))
        .reg fVal,vNew, gNew, p_gOld, gOld, p_vBuff, vVal, p_kBuff, kVal, i, m, pointer,
product, output
        ;move into local registers
        mv A4,fVal
        mvkl _filterLength, m
        mvkh _filterLength, m
        ldw *m,m
        mvkl _kVal, p_kBuff
        mvkh _kVal, p_kBuff
        mvkl _vVal, p_vBuff
        mvkh _vVal, p_vBuff
        mvkl _gOld, p_gOld
        mvkh _gOld, p_gOld
        mvk 0,output

        sub m,1,m ;filfer_size-1,for 0 index
loop:
        ;load array values
        ldw *+p_kBuff[m], kVal ;k_m
        ldw *+p_vBuff[m], vVal ;v_m
        sub m,1,m ;used for g_m-1
        ldw *+p_gOld[m], gOld ;g_m-1

        ;compute f_m-1
        mpysp kVal,gOld,product
        subsp fVal,product,fVal ;f_m-1=f_m(n)-k_m*g_m-1(n-1)

        ;compute g_m
        mpysp kVal,fVal,product
        addsp product,gOld,gNew ;g_m=k_m*f_m-1(n)+g_m-1(n-1)

        ;compute v_m
        mpysp vVal,gNew,product
        addsp product,output,output ;y+=g_m*v_m

        ;store gOld=gNew
        add m,1,product
        stw gNew,*+p_gOld[product] ;gOld[m+1]=gNew
[m]     b loop
        stw fVal,*p_gOld ;gOld[0]=f0
        ;compute final f*v0+vOld
        ldw *p_vBuff, vVal ;v0
        mpysp vVal,fVal,product
        addsp product,output, output ;output+=f0*v0
        mv output, A4
        .endproc A4, B3
        b B3
```

2/21/19

NP

Questions

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

Cascade		Lattice-Ladder	
Adds	4·sections	Adds	3·N+1
Mults	7·sections − 1	Mults	3·N+1
Memory	7·Sections +1 +3·sections	Memory	3·N
	Filter D[n-i]		k, v, gold

sections = 4

~~N=9~~ N=8

2. How do the two frequency responses compare?

The plots appear Identical

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

The Lattice Ladder seems to require less computations.
The Lattice-Ladder requires the computation of K and V, but that is during init() so during runtime, it is faster

2/21/19
AP