44

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

Lab 6: Finite Word-Length Effects

Objective

The purpose of this lab is to observe the effects of finite word-length coefficients and computations, and to learn how to use fixed-point computations.

Background

Since DSP processors must have finite word lengths for coefficients and computations, the result of a DSP operation in general will not be identical to the result using infinite precision arithmetic. The effects of rounding in computations must be understood, especially if fixed-point processors are used.

This lab will require that you *simulate* **Q-format** fractional fixed-point arithmetic. To do this, you *must* do all computations using signed integers where the required number of significant bits are the MSBs and the remaining LSBs are masked to 0. For example, if you are required to use b-bit coefficients , you must multiply the original floating-point values by 2^{31}, add $2^{(31-b)}$ (to round-to-nearest), convert to integers, and then mask the (32-b) LSBs to 0. This procedure will convert the coefficients to signed b-bit fractional fixed-point (Q-($b-1$) format).

Note that the above conversion to fixed-point must only be done once for the coefficients, and assumes the floating-point coefficients have a maximum absolute value less than one. If the coefficients a_k or b_k are such that the maximum magnitude coefficient is not less than one, the values must be normalized so the maximum coefficient is less than one. This can be done by multiplying the original floating-point values by 2^n, where n is a number of shifts that will place the first non-sign bit of the maximum coefficient in the location of bit 30 in the data word, effectively leading to a scaling of the floating-point coefficients by $2^{(n-31)}$.

For example, if the maximum magnitude coefficient is 1.5, then all the coefficients must be multiplied by 2^{30}. This causes 1.5 to be given a fixed-point representation of $0_{\triangle}11000\text{—}0$. Remember to scale the results of the first summing node in each second-order section to compensate for the *implied* scaling of $\frac{1}{2}$ caused by the conversion of 1.5 to fractional fixed-point. For more details, refer to the discussion on this subject given in class.

Input samples from the A/D converter must also be placed in the b MSBs (left justified) of the integer used to store them, and the (32-b) LSBs must be set to 0.

To simulate a signed b-bit fractional fixed-point multiply (with rounding), you must use the `SMPYH` instruction, shift the result left by 1, round, and then mask all but the b MSBs of the result to 0.

4/1/19

Procedure

1. Create an IIR digital filter using `Matlab`. The filter should be a Type I bandpass Chebyshev filter of order 10 (5 biquads). Make the passband 500 Hz centered at 5 kHz with 0.5 dB of ripple, and the sampling frequency 48 kHz. Also find the Direct Form II (non-cascaded) filter coefficients.

   Note that `Matlab` contains a nice filter design tool (`fdatool`) that will allow you to quickly design the appropriate filter. Be sure that if you use this tool, you select the result to be presented in second order sections and that you use appropriate scaling (L_2 or infinity) and order the sections in decreasing order.

2. Using the same IIR coefficients from procedure 1, filter a white noise input using a Direct Form II IIR filter made up of cascaded 2nd-order sections and 16, 12, 8, and 4-bit fixed-point arithmetic (Q-15, Q-11, Q-7, and Q-3). Remember that you will need to compensate for any normalization of the coefficients required to convert from floating-point to fixed-point, and make sure the output samples sent to the D/A are the correct format. Record the spectrum of the output. Your program should be implemented using assembly code, except for setting up the DSK, and reading in the filter coefficients.

   You should be able to write your code to make the change from one word-length to another by a few simple defines.

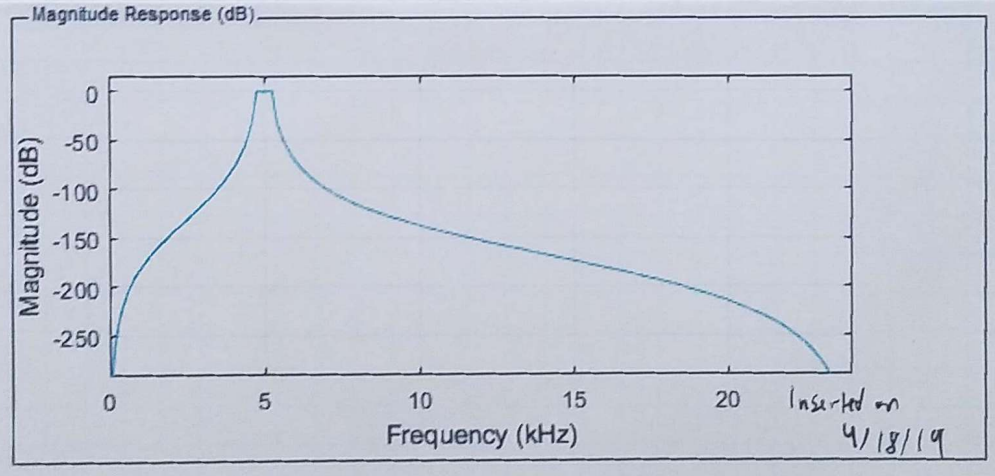


Figure 1 Type I Chebyshev of order 10. 500 Hz passband centered at 5 kHz.

4/1/19

$Bits = 15$ Float to Fixed

float num

$x = num \cdot 2^{31-n}$ n is the number of right shifts to get num < 1

$x = x + 2^{31-(Bits+1)}$ needs to include the sign

x & 0xFFFF0000 for 16 bit

Rounding

10...1 (N bits) add 00... 0 (N bits), 1 (N+1) → mask

to multiply

smpyh x, y, z

round(z)

need to rescale output of first node in each section if coefficients needed scaling

$x + a_1 d[n-1] + a_2 d[n-2]$ need to scale a_1 by 2

$\left(\frac{x}{2} + \frac{a_1}{2} d[n-1] + \frac{a_2}{2} d[n-2]\right) 2$ rescale by 2 after sum node

Need to scale input

4/17/19

AO

Do NOT scale input. use $a_0 = 1/2$ coefficient.

Input being read in wrong. Input samples currently recived ~~at~~ in the lower 16 bits of 32 bit register. Need to shift left by 15 not 16 to preserve the sign bit in the msb place.

Signal of noise 100kHz is ~~now~~ currently being filtered to only 51cHz, but it seems like there may be some error or over flow. The signal becomes noise like every few seconds then goes back to 5kHz sine wave

This causes slight bumps in dB at various frequencies other than 5kHz

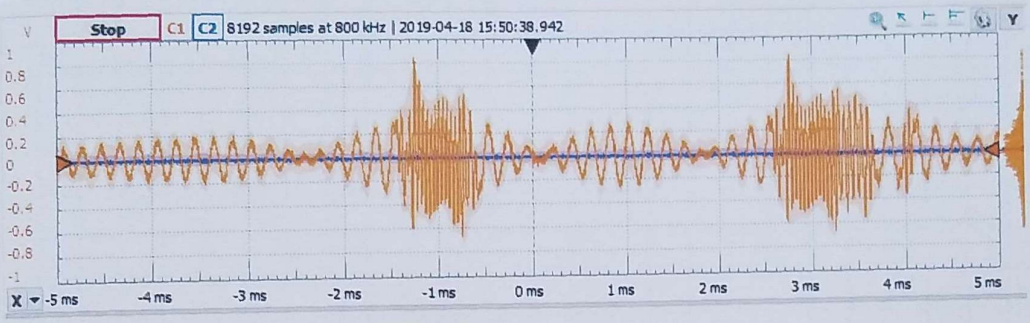


Figure 2 Filtered signal with some noise fluctuations. Q-15

The amount of noise that leaks through seems to increase as the number of bits decrease

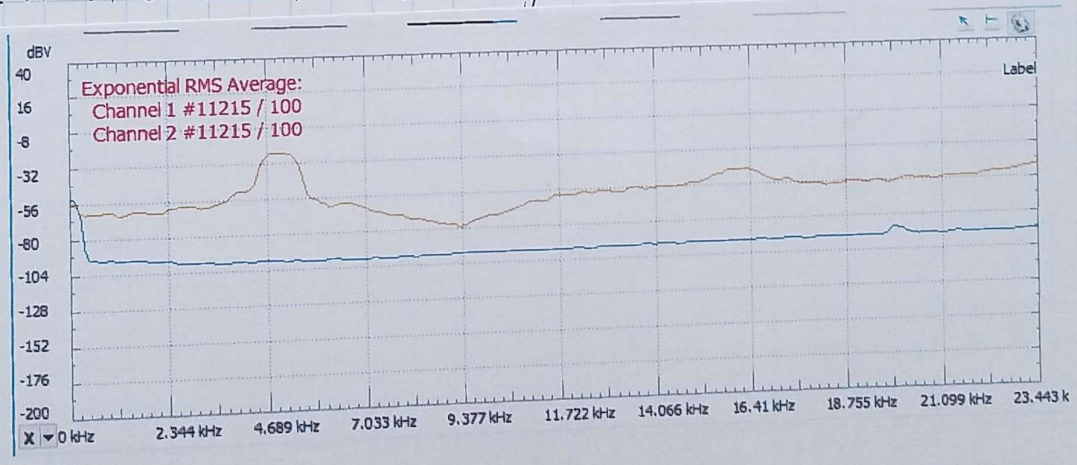


Figure 3 Q-15 FFT output.

4/18/19

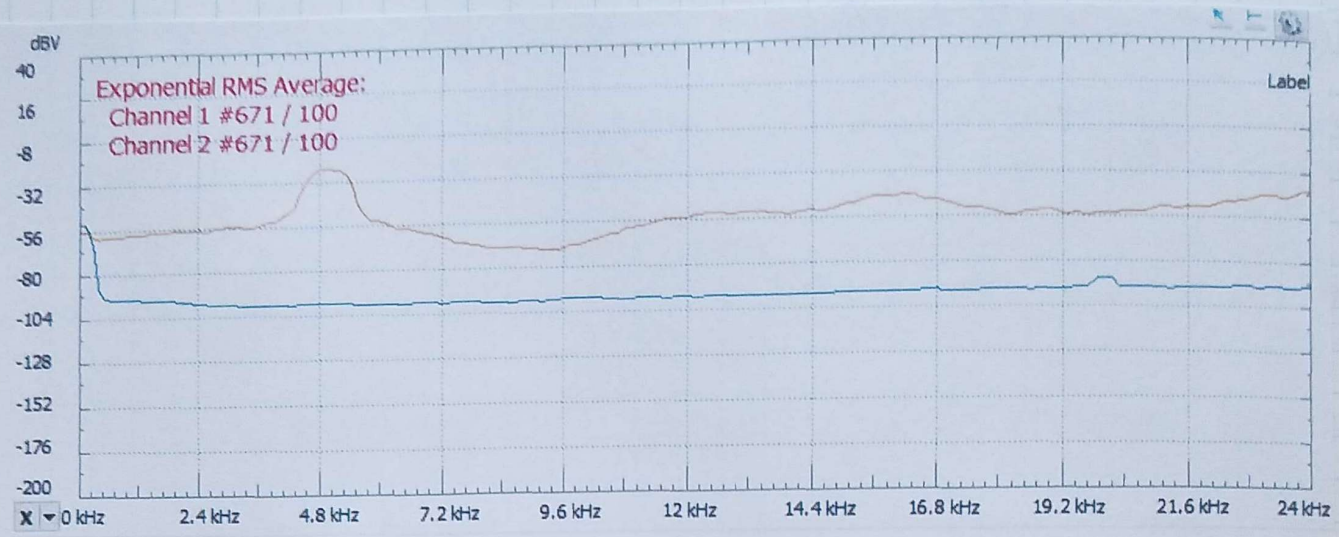


Figure 4 Q-11 FFT output.

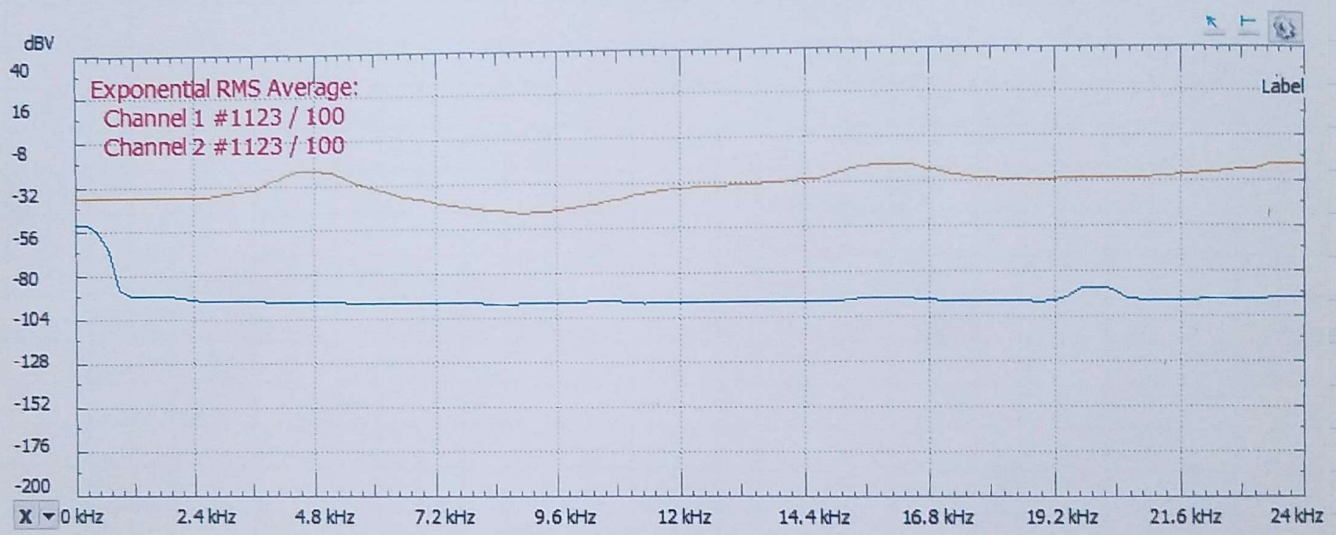


Figure 5 Q-7 FFT output. Noise nearly overpowers signal at 5kHz

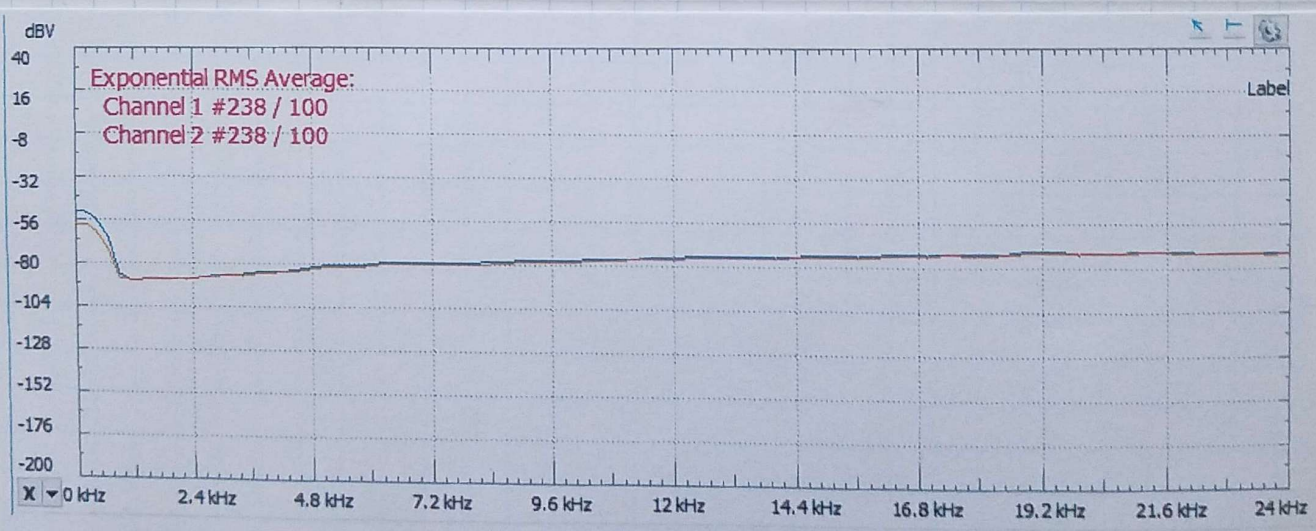


Figure 6 Q-3 FFT. Unrealizable. Not enough bits to represent the small values of the filter.

4/18/19
M

Questions

1. What did you need to do to prevent overflow in the filters?

L2 scaling was used to help prevent overflow. Some coefficients were too large so I added a shift value to my filter which specified how much the coefficient needed to be shifted to make it less than 1.

I didn't do this, but I could have added some protection to add/sub to prevent the value from getting too big and wrapping around to a negative value. I could have set a saturation max to prevent the wrapping.

2. What is the effect of the different fixed-point word sizes on the IIR frequency response?

As the word size decreased, the average noise increased. In Fig 4, the lowest noise was ~~-56 dB~~ ~-70 dB. In Fig 5, the lowest noise was ~-45 dB giving about a 15 dB increase in noise

3. Do you observe a SNR degradation? What are the evidences of this?

As seen in Fig 5, The noise nearly over powers the 5kHz signal. for Fig 5, the difference between the min and max magnitude (Δ) is ~20 dB. for Q-7

In Fig 3, the Δ is ~55 dB for Q-15

4. Compare the magnitudes of the coefficients of the IIR filter for both Direct Form II (non-cascaded) and the cascaded Direct Form II filters provided by `Matlab`. How does this affect the implementation of the filter in fixed-point?

The max value of the cascaded sections was ~1.5. The max value for the non-cascaded DFII structure is ~132

I needed to shift the cascade coefficients by 1
to I would need to shift the non-cascade coefficients by 8

This means I lose 7 more bits of data using non-cascaded sections

4/18/19

```
        .title "cascade.sa"
        .def _cascadeSection
        .def _fixedMpy
        .def _cascade
        .def _fixedRound
        .def _float2Fixed
        .sect ".fixed"
        .global _input_sample
        .global _output_sample
        .global _cascade
        .global _fixedMpy
        .global _fixedRound
        .global _dOffset
        .global _gain
        .global _filterSections
        .global _float2Fixed
_filterSections       .usect "arraySect", 160, 4    ;allocate int filterSections(5*8)
        .global _dBuffer
_dBuffer .usect "array2Sect", 80, 4                 ;allocate int dBuffer(5*3);
_BITS .set 4                                        ;total bits including sign
_sections .set 5

*****************************************
*       Interrupt to perform a cascaded filter
*       using fixed point arithmetic
*****************************************
_cascade: .cproc
        .reg x, filter, dBuff, dOffset, p_dOffset, sections, p_filter, p_dBuff, i
        .reg count, product, result, gain, mask, inputLR, shift, temp

        mvkl _dBuffer, dBuff
        mvkh _dBuffer, dBuff
        mvkl _filterSections, filter
        mvkh _filterSections, filter
        mvkl _dOffset, p_dOffset
        mvkh _dOffset, p_dOffset
        mvkl _sections, sections
        mvkh _sections, sections
        mvkl _gain, gain
        mvkh _gain, gain

        ldw *p_dOffset, dOffset
        ldw *gain, gain

        .call inputLR = _input_sample()             ;obtain 2 int_16

        mvkl 0x0000FFFF, mask                       ;mask lower 16 bits
        mvkh 0x0000FFFF, mask
        AND mask, inputLR, x                        ;right channel (chan2)
        mvk 15, shift
        shl x, shift, x                             ;shift by 15 to preserve pos sign
        .call x = _fixedRound(x)
        .call x = _fixedMpy(x,gain)                 ;input gain

        mvk 0, i
;loop over every section for cascade
loop:
        mpy i, 4, count ; dBuff[i][4], i*4
        addaw dBuff, count, p_dBuff ;&dBuff[i][0]
        mpy i,8,count ;filterSections[i][8]
        addaw filter, count, p_filter ;&filterSections[i][0]
```

9/18/19

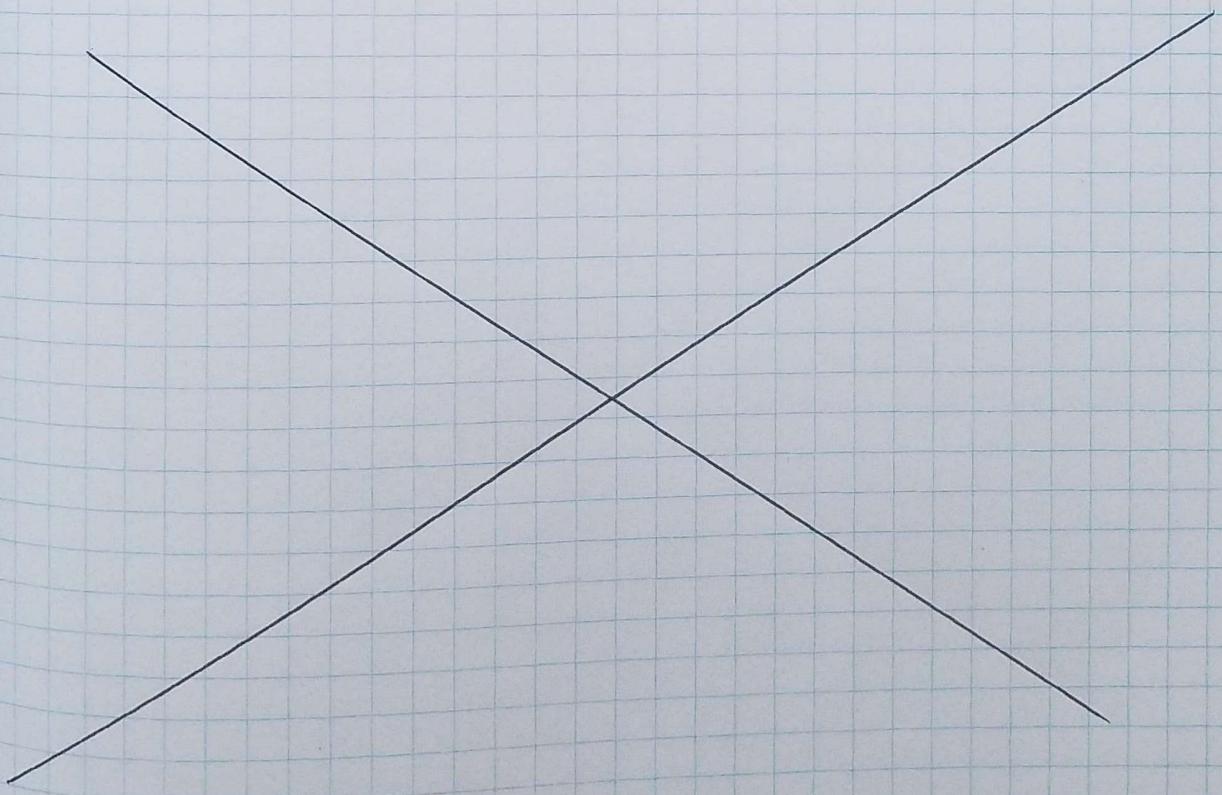
```
        .call x = _cascadeSection(x,p_dBuff,dOffset,p_filter)     ;get output of single section
        add i, 1, i
        cmplt i, sections, count
[count] b loop

        add dOffset, 1, dOffset
        mvk 0x3, mask
        AND dOffset, mask, dOffset                  ;(dOffset+1)%4
        stw dOffset, *p_dOffset
        shru x, shift, x                            ;shift into lower 16 bits
        .call _output_sample(x)                     ;output sample


        mvk 0x1, mask                               ;enable interrupt
        mvc CSR, temp
        OR temp, mask, temp
        mvc temp, CSR

        .return
        .endproc

*************************************
*       Fixed point computation of a
*       cascaded section
*       @param x: input fixed point value
*       @param B4: pointer to dBuff array, used for circular buffering
*       @param dOffset: starting location for dBuff
*       @param filter: fixed point filter with format:
*              a2k, b2k, a1k, b1k, a0k, b0k, scalarShift(int)
*************************************
_cascadeSection: .cproc x, B4, dOffset, filter
;cascadeSection(x(n),*dBuff(n=0),dOffset,filterCoef)
        .reg p_a, p_b, a, b, d, product, dresult, yresult, count, scalarShift, oldAMR, sign,
mask
        .circ dBuff/B4
        mvc AMR, oldAMR
        mvkl 0x3<<16|0x1<<8, count
        mvkh 0x3<<16|0x1<<8, count
        mvc count,AMR                               ;B4 circular buffer size 4*4

        mvk 0,yresult
        addaw dBuff, dOffset, dBuff                 ;shift dBuff to d(n)
        ldw *+filter[6], scalarShift                ;load n value to shift by

        subaw dBuff, 2, dBuff                       ;d[n-2]
        mvk 0, dresult
        mvk 2, count                                ;i=2;i>0;i--
dLoop: .trip 2
        lddw *filter++, b:a
        ldw *dBuff++, d
        .call product = _fixedMpy(d,a)
        sub dresult,product,dresult                 ;d-=d(n-i)*a(i)

        .call product = _fixedMpy(d,b)
        add product,yresult,yresult                 ;y+=d(n-i)*b(i)
        sub count, 1, count
[count] b dLoop

        lddw *filter++, b:a
        .call product = _fixedMpy(x, a)
        add dresult, product, dresult               ;d[n]=a(0)*x(n)
```

4/18/19

```
        shl dresult, scalarShift, dresult          ;undo scaling
        stw dresult, *dBuff                        ;store d[0]
        .call product = _fixedMpy(dresult,b)       ;d[0]*b[0]
        add product, yresult,yresult               ;y+=d[0]*b0

        mvc oldAMR, AMR
        .return yresult

        .endproc

**********************************
*       Computes the fixed point multiplication.
*       Rounds the result to _BITS
*       @param a: first fixed point argument
*       @param b: second fixed point argument
**********************************
_fixedMpy .cproc a,b
        .reg product
        smpyh a,b,product                          ;do fixed multiply
        .call product = _fixedRound(product)
        .return product
        .endproc

***********************************
*       Rounds the given value to a fixed _BITS
*       @param value: fixed value to be rounded
***********************************
_fixedRound .cproc value
        .reg val, mask, product,bits
        mvkl 0x80000000, val                       ;1 in the MSB
        mvkh 0x80000000, val
        mvk _BITS, bits
        shru val, _BITS, val                       ;shift 1 to the bits+1 place
        add value, val, product                    ;add 0b[0...0]1 where the 1 is at bits+1

        mvkl 0xFFFFFFFF, mask
        mvkh 0xFFFFFFFF, mask
        mvk 32, val
        sub val,bits,bits
        shl mask, bits, mask                       ;mask upper b bits
        AND product, mask, product

        .return product
        .endproc

*************************************
*       converts a floating point number to a
*       fixed point signed number
*       @param floatNum: floating point number to convert
*       @param bits: number of bits for the desired fixed point number
*              (bits includes the sign bit)
*       @param maxval: max value used to scale the output
*************************************
_float2Fixed .cproc floatNum, bits, maxVal
        .reg floatTemp, float1, n, boolVal, div2, npow, temp, mask

        mvkl 0x3f800000, float1                    ;float 1.0f
        mvkh 0x3f800000, float1
        mvk 31, n
        mvk 2, div2                                ;number to shift by (2^n)
```

4/18/19
MP

```
        intsp div2, div2
        rcpsp div2, div2                                ;float 1.0f/2

        ;find the 2^-n value to scale below 1
GTOneLoop:
        cmpgtsp maxVal, float1, boolVal
[boolVal]       sub n, 1, n                             ;divides float by 2 until less than 1
[boolVal]       mpysp maxVal, div2, maxVal              ;decrement n until max<1.0f
[boolVal]       b GTOneLoop

        ;multiply by 2^(31-n) where n causes scaling
        sub n, 1, n                                     ;exponent format E=0b1001 1110 for 2^31
        mvk 0x80, temp                                  ;0b1000 0000
        OR n, temp, n                                   ;get into required exponent format
        shl n, 23, npow                                 ;shift n into exponent of float
        mpysp floatNum, npow, floatNum                  ;mpy 2^(31-n) where n makes result < 1

        ;add 2^(31-b)
        mvk 0x9E, n                                     ;exponent format 0b0 1001 1110 (float 2^31)
        sub n, bits, n                                  ;31-b
        shl n, 23, n                                    ;shift n into exponent of float
        addsp floatNum, n, floatNum                     ;add 2^(31-b)
        spint floatNum, floatNum                                ;convert to int

        ;mask the (32-b) LSBs to 0
        mvkl 0xFFFFFFFF, mask
        mvkh 0xFFFFFFFF, mask
        mvk 32, n
        sub n,bits,bits
        shl mask, bits, mask                            ;mask has b+1(signed) zeros in LSB
        AND floatNum, mask, floatNum                    ;set lower b+1 bits to 0

        .return floatNum
        .endproc
```



9/18/14