RTDSP Lab 5

Yong Wen Chua (ywc110) & Ryan Savitski (rs5010)

Declaration

Declaration: We confirm that this submission is our own work. In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others. We are aware that this course is bound by penalties as set out in the College examination offenses policy.

Signed: Yong Wen Chua & Ryan Savitski

Contents

List of Figures

1 Single-pole Low Pass Filter

1.1 Derivation

The resistor has a value of $R = 1\text{k}\Omega$ and the capacitor, a value of $C = 1\mu\text{F}$. The capacitor has an impedance of $Z_c = \frac{1}{j\omega C}$ and the transfer function is thus given by

$$H(j\omega) = \frac{V_{out}}{V_{in}} = \frac{\frac{1}{j\omega C}}{\frac{1}{j\omega C} + R} = \frac{1}{1 + RCj\omega}$$

The corner frequency is expected to be at $\omega_c = \frac{1}{RC} = \frac{1}{1000 \times 10^{-6}} = 1000\text{rads}^{-1} \approx 160\text{Hz}$.

The Tustin Transform is given by $s = \frac{2}{T_s}\frac{Z-1}{Z+1}$of which case we have $s = j\omega$, and $T_s = \frac{1}{8000} = 125\mu\text{s}$. Then, the Z-domain transfer function is given by

$$\begin{aligned} H(Z) &= \frac{1}{1 + RC(\frac{2}{T_s}\frac{Z-1}{Z+1})} \\ &= \frac{T_s + T_s Z^{-1}}{T_s + 2RC + (T_s - 2RC)Z^{-1}} \\ &= \frac{\frac{T_s}{T_s+2RC} + \frac{T_s}{T_s+2RC}Z^{-1}}{1 + \frac{T_s-2RC}{T_s+2RC}Z^{-1}} \end{aligned}$$

So the coefficients $b_0 = b_1 = \frac{T_s}{T_s+2RC} \approx 0.05882352941176471$, and $a_1 = \frac{T_s-2RC}{T_s+2RC} \approx -0.8823529411764705$.

1.2 Code Implementation

The code listing for the implementation of this simple filter is given in section A.2.

Because this is simply a second order IIR filter, the coefficients and buffers can be simply implemented in small arrays as below:

```
// The order of the FIR filter +1
#define N 2

// coefficients
double a = -0.8823529411764705;
double b[] = {0.05882352941176471, 0.05882352941176471};

// define the buffers
double inputBuffer[N] = {0};
double outputBuffer = 0;
```

In the ISR, after the new sample is read, the input buffer is simply shifted.

```
double sample = mono_read_16Bit();  // read

// Move the input buffer
inputBuffer[1] = inputBuffer[0];
inputBuffer[0] = sample;
```

And the output is calculated and saved to the output buffer, before writing to the output.

```
// Calculate the difference equation and save to buffer
outputBuffer = b[0]*inputBuffer[0] + b[1]*inputBuffer[1] - a*outputBuffer;

mono_write_16Bit(outputBuffer);
```

1.3 Filter Response

The theoretical frequency response estimated by Matlab is given in figure 1.1, and the measured frequency response of the implementation is given in figure 1.2. In general, the gain of the implemented filter behaved according to prediction. The phase of the implemented filter, however, has a higher slope, and appears to be linear. This is due to the group delay on the DSP, also observed in the previous lab.

1.4 Time Constant Measurement

There exists a high pass filter on the input to the board, the effect of which can be seen from the gain plot in figure 1.2, therfore, together with our LPF, we have a second order filter. To measure the time constant, a square wave of 100 Hz was chosen to be less affected by the HPF. Although this has the downside of not letting the LPF to completely settle before the next edge of the square wave.

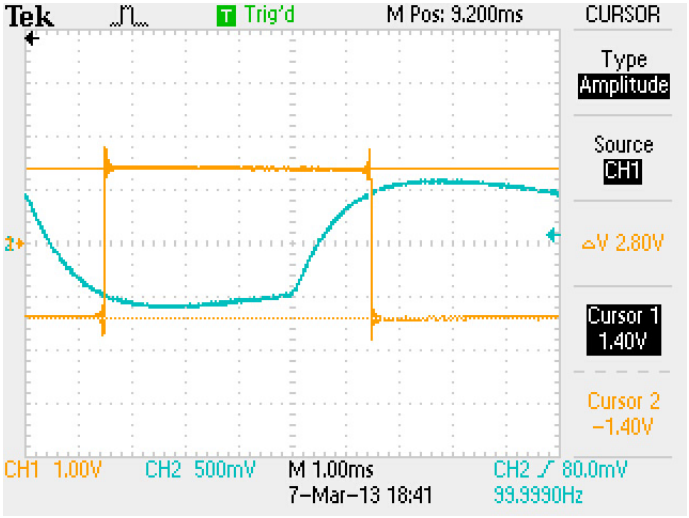

Figure 1.3: Time constant measurement. Channel 1 (yellow) shows the square wave input, while Channel 2 (turquoise) shows the output from the DSK.

The result of driving the DSK with the square wave is shown in figure 1.3. Because of the potential divider at the input port, the input to the DSK code will be halved. Thus, the output from the DSK is expected to be halved.

The capacitor in this second order filter causes the output to not rise immediately to the maximum when the square wave input changes from the minimum to the maximum. However, because insuffiient time is allowed for the capacitor in the filter to charge and discharge fully, the output from the DSK is less than what is expected, as seen in figure 1.3 where even though the scale of Channel 2 is halved of that of Channel 1, the output never reaches the level of the square wave input.

If q_+ and q_- are the maximum and minimum values of the output respectively, then if the output at $t = 0$ is $q(0) = q_-$, then while the capacitor is charging, the output at time t is given by $q(t) = q_- + (q_+ - q_-)(1 - e^{t/\tau})$ where $\tau = RC$ is the time constant. Then $q(\tau) = q_- + (q_+ - q_-)(1 - e^{-1})$.

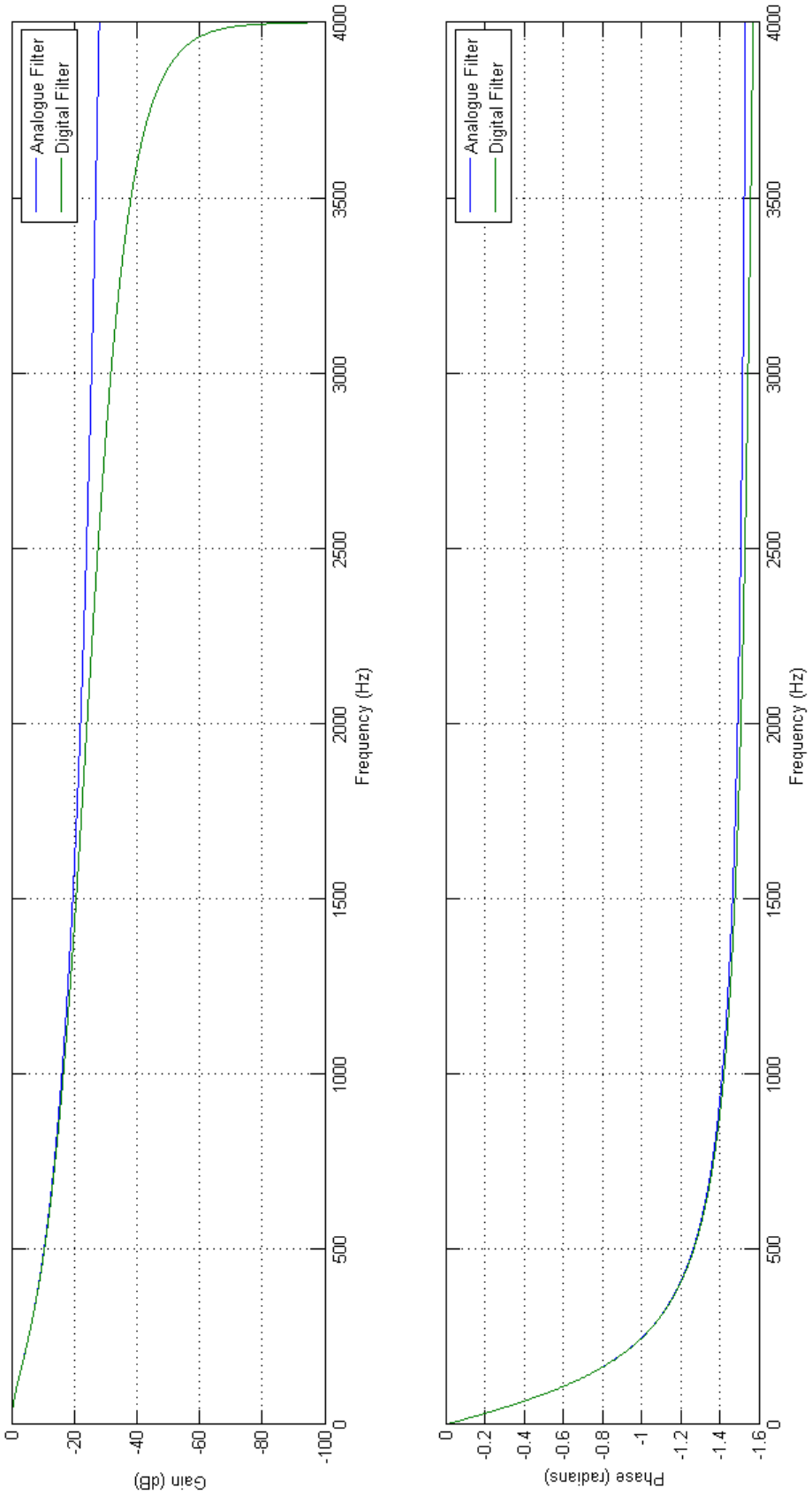

Figure 1.1: Theoretical analogue and digital filter response, calculated in Matlab using code in section A.1.

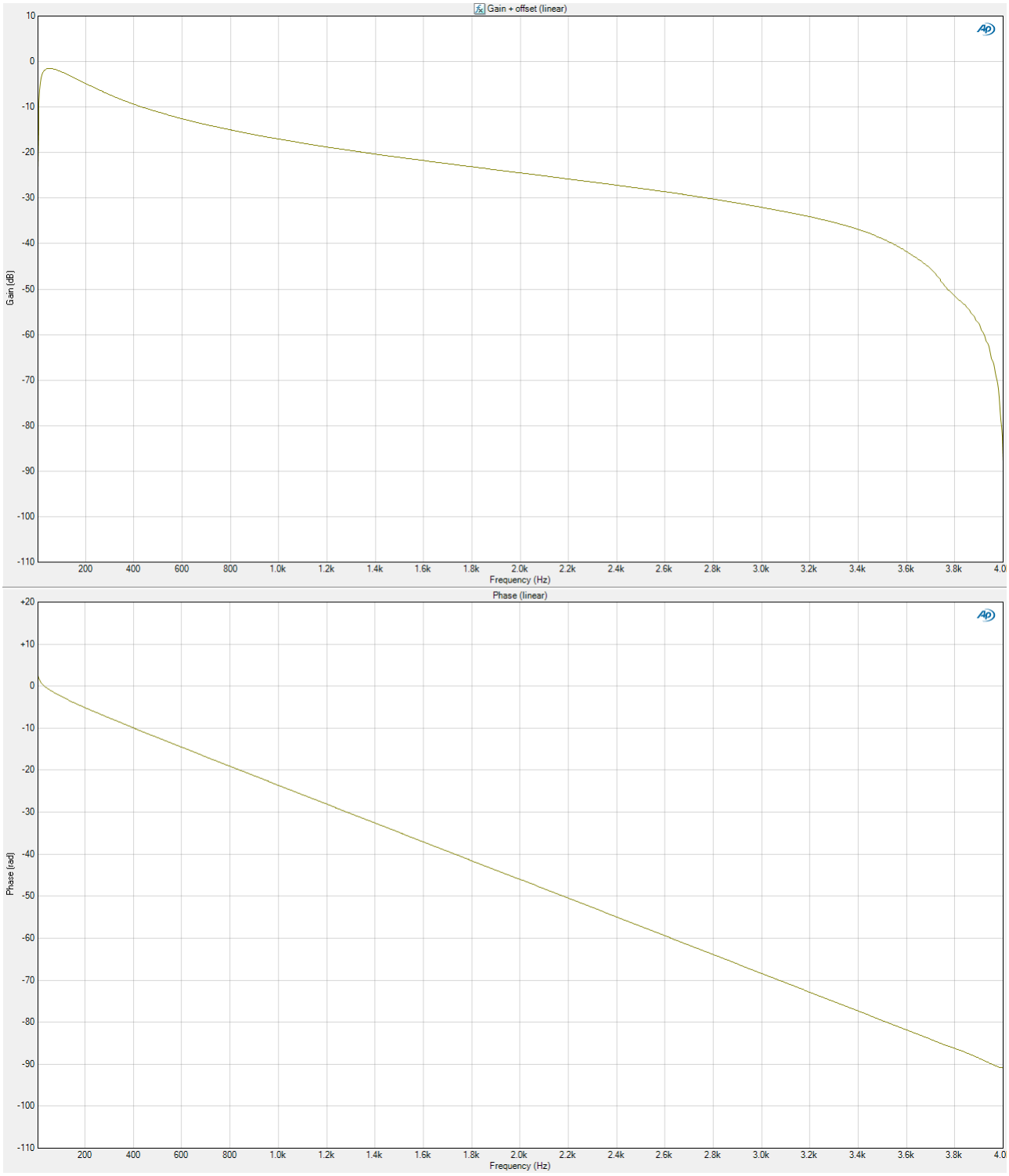

Figure 1.2: Spectrum analyser output.

In this case, for the output, at $q(0) = q_- \approx -500\text{mV}$. This is higher than the expected -700mV due to insufficient time for discharging. At $t = 0$, the square wave input to the DSK goes up to 1.4V, or 0.7Vas seen by the DSK after the potential dividers at the input. Thus, $q_+ = 700\text{mV}$.Then, $q(\tau) \approx 259\text{mV}$. From the trace, this works out to be $\tau \approx 0.9\text{ms}$, which is slightly lower than expected. The corner frequency is thus calculated to be $\omega_c = {}^1/_\tau \approx 1111\text{rads}^{-1}$.

A more accurate estimate of the corner frequency could be gained from extrapolating the gain plot's slope past the frequency range affected by the HPF.

2 Bandpass Filter

2.1 Coefficient Computation

Based on the specifications given, the coefficients were generated in Matlab using the code in section A.3. Then, the coefficients were written to a header file for inclusion later on. The generated header file is given in section A.4.

The frequency response of this filter, as predicted by Matlab is given in figure 2.1.

2.2 Direct Form II Implementation

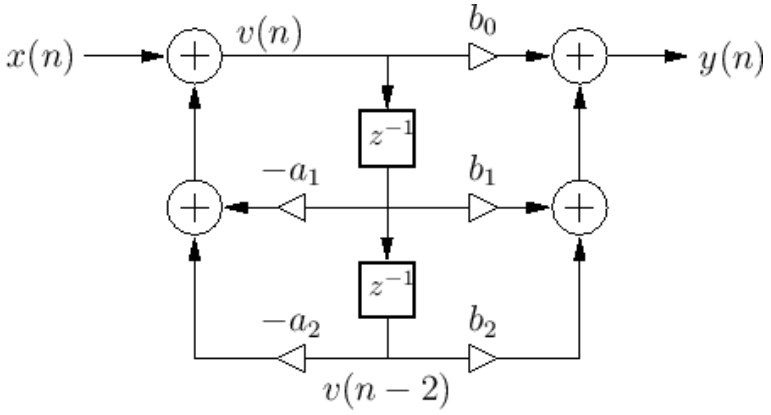

Figure 2.2: Direct Form II Signal Flow Graph. Source

Based on the signal flow graph given in figure 2.2, the following difference equations are obtained, for a filter of order N:

$$v(n) = x(n) - a_1 v(n-1) - a_2 v(n-2) - ... - a_N v(n-N) \tag{2.1}$$

$$y(n) = b_0 v(n) + b_1 v(n-1) + b_2 v(n-2) + ... + b_N v(n-N) \tag{2.2}$$

where a_n and b_n are the nth coefficients, $x(n)$ is the nth input, $y(n)$ is the nth output, and $v(n)$ is the nth entry in the delay buffer line. This implies that the Direct Form II implemenation only requires one delay buffer. The implementation below reflects these set of equations.

2.2.1 Code Operation

The complete code listing for the implementation can be found in section A.5.

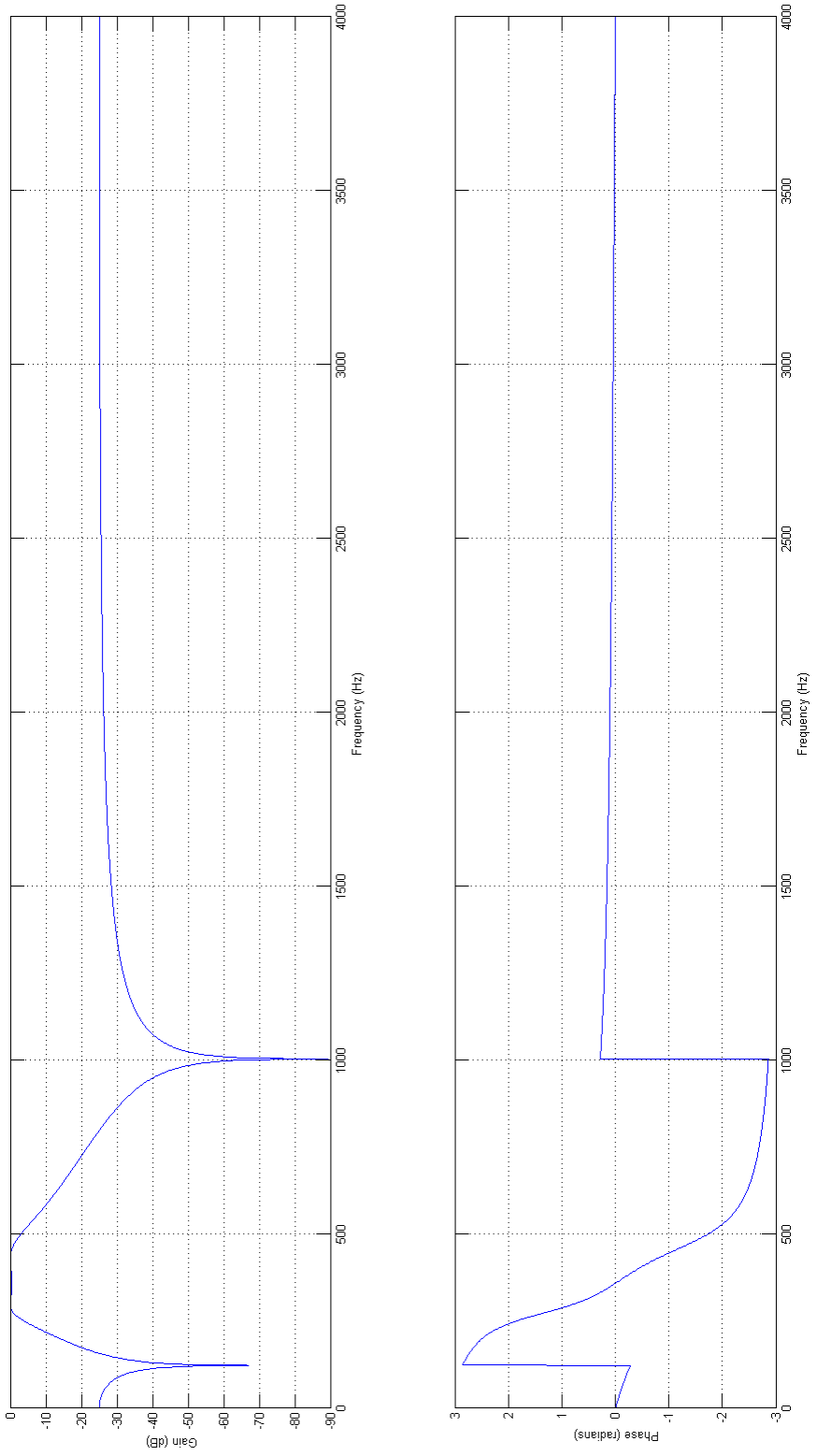

Figure 2.1: Bandpass Filter Frequency Response predicted by Matlab.

The required circular delay buffer `v` has its pointer declared in the global scope, followed by a variable `index` that is used to indicate the next entry to write to in the circular buffer.

```
double *v;  // pointer to the delay line buffer (circular)

int index = 0; // index to the circular buffer
```

The buffer is then allocated memory from the heap in `main()` using the size N that is defined in the coefficient header (see section A.4).

```
int main(void){
  // initialise the delay buffer
  v = (double *) calloc(N, sizeof(double));
  // ... other code
}
```

The `ISR` calls a function to implement the filter (as recommended in the notes). The function first initialises a series of pointers to the three arrays for use in the MAC loops. `vOffset` points to the entry in the circular buffer that is calculated in this invocation of the filter (i.e. $v(n)$) , and `vPtr` points to the one after `vOffset` (i.e.$v(n-1)$) for calculation according to equation (2.1).

```
   double* vPtr = v + index + 1;          // loop index pointer
  double* vOffset = v + index;  // current v to write to
    double* vEnd = v+N;    // one element after end of buffer
    double* aPtr = a+1;    // pointer to a1 (ao is not used for calculation)
    double* aEnd = a+N;    // one element after end of a
    double* bPtr = b;      // pointer to b0
    double* bEnd = b+N;    // one element after end of b
    double output = 0;     // output
```

The difference equation given in equation (2.1) is then calculated. The current `input`($x(n)$) is first written to the dereferenced `vOffset` ($v(n)$). The two `for` loops are then used to exploit the circular nature of the `v` buffer. This optimisation originates from Lab 4. The first `for` loop will calculate all the values in the `v` buffer up to the end of the buffer, and then the second `for` loop will handle the entries after the buffer loop pointer `vPtr` has wrapped back to the start of the buffer.

```
    *vOffset = input; // read and store

  // calculate v for the current input x
    // v(n) = x(n) - a1 * v(n-1) - a2 * v(n-2) - ...
    for (; vPtr < vEnd; ++vPtr, ++aPtr)
        *vOffset -= (*aPtr) * (*vPtr);

    for (vPtr = v; aPtr < aEnd; ++vPtr, ++aPtr)
        *vOffset -= (*aPtr) * (*vPtr);
```

The `output` $y(n)$ is then calculated according to equation (2.2) using the same circular buffer technique discussed above.

```
    // calculate output y
    // y(n) = b0*v(n) + b1*v(n-1) + b2*v(n-2) + ...
    for (; vOffset < vEnd; ++vOffset, ++bPtr)   // reuse vOffset pointer now
        output += (*bPtr) * (*vOffset);

    for (vOffset = v; bPtr < bEnd; ++vOffset, ++bPtr)   // reuse vOffset pointer now
        output += (*bPtr) * (*vOffset);
```

Finally, the `index` is decremented, and wrapped around to the end of the buffer if necessary.

```
    // decrement index
    index = (index == 0) ? N-1 : index-1;
```

2.2.2 Frequency Response

The frequency response of the implemented filter is given in figure 2.3. Compared with the expected output from Matlab given in figure 2.1, we can see that the gain output is generally similar, except for the increased attenuation on the DSK output near the Nyquist frequency. This is due to aliasing caused by the imperfect reconstruction filter on the DSK. The group delay on the output for the DSK also contributes to an increase in the gradient of the phase output, when compared with that predicted by Matlab.

2.2.3 Code Performance

Profiling the code for filters of order 4, 8, 16 with no optimisatios and o2 shows the following runtimes:

Optimisation level	Order of filter	Runtime (cycles)
none	4	604
none	8	1000
none	16	1792
o2	4	312
o2	8	436
o2	16	679

Table 2.1: Direct form II IIR filter runtimes

Solving this for a constant and per-order cycle costs, we get:

- No optimisations: $208 + 99 \times (order)$
- o2: $188 + 31 \times (order)$

No surprising results were observed with optimisations reducing both the static and per-order cycle count.

2.3 Direct Form II Transposed Implementation

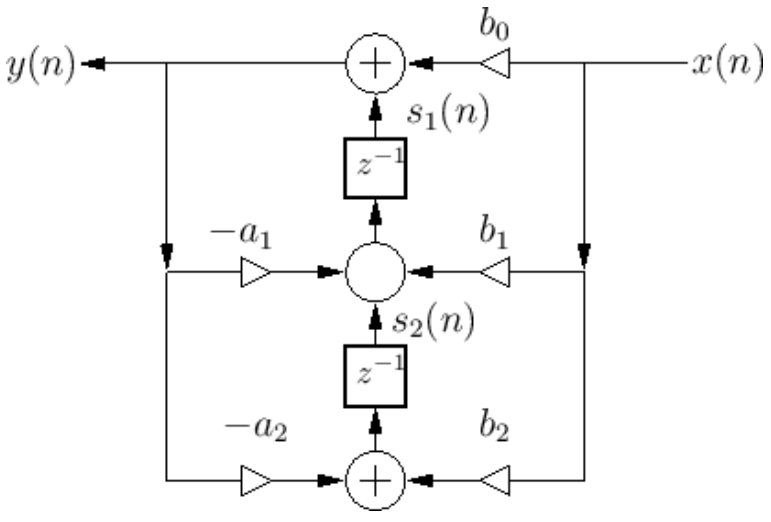

Figure 2.4: Direct Form II Transposed Signal Flow Graph. Source

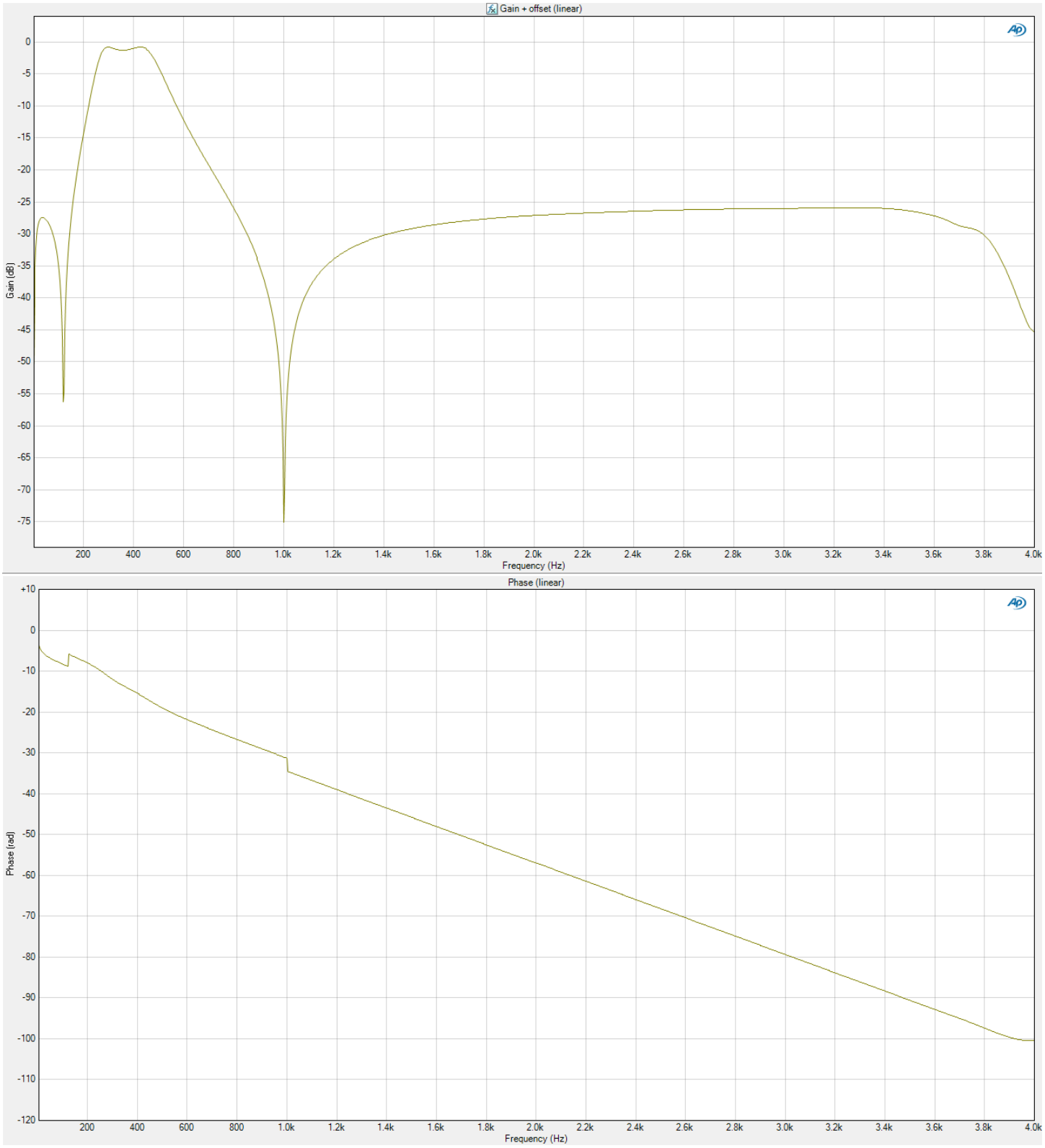

Figure 2.3: Spectrum output of the Direct Form II implementation.

According to the signal flow digram given in figure 2.4, the following equations can be derived:

$$y(n) = s_1(n) + b_0 x(n) \tag{2.3}$$

$$s_\lambda(n) = \begin{cases} s_{\lambda+1}(n-1) + b_\lambda x(n-1) - a_\lambda y(n-1) & \lambda \in [1, N) \\ b_\lambda x(n-1) - a_\lambda y(n-1) & \lambda = N \end{cases} \tag{2.4}$$

where N is the order of the IIR filter, a_λ and b_λ are the coefficients, $s_\lambda(n)$ is the λth entry in the buffer for the nth input, $x(n)$ is the nth input, and $y(n)$ is the nth output.

2.3.1 Code Operation

For the implementation, the s buffer given in equation (2.3) and equation (2.4) is renamed to `v`. The code listing can be found in section A.6.

A pointer to the buffer was declared in global context, and initialised in `main()`.

```
double *v;  // pointer to the buffer
//...
int main(void){
  // initialise the buffer
  v = (double *) calloc(N-1, sizeof(double));
  //...
}
```

The function for the filter in this case is very simple. The usual variables are first set up. The buffer values for the $n+1$th iteration is calculated during the nth iteration. The filter firsts calculates $y(n)$ according to equation (2.3).

```
    double y = 0;   // output
    int i = 0;  // loop index
  y = v[0] + b[0]*x;
```

The buffer `v` for the next iteration is then calculated according to the first case of equation (2.4).

```
    // update buffer for next iteration
    for (; i < N-2; i++)
        v[i] = v[i+1] + b[i+1]*x - a[i+1]*y;
```

Finally, the second case ofequation (2.4) is calculated.

```
    v[N-2] = b[N-1]*x - a[N-1]*y;
```

2.3.2 Frequency Response

The frequency response of the Direct Form II Transposed implementation on the DSK is given in figure 2.5. It is generally similar to the frequency response of the Direct Form II implementation given in figure 2.3.

2.3.3 Code Performance

Profiling the code for filters of order 4, 8, 16 with no optimisations and o2 shows the following runtimes.

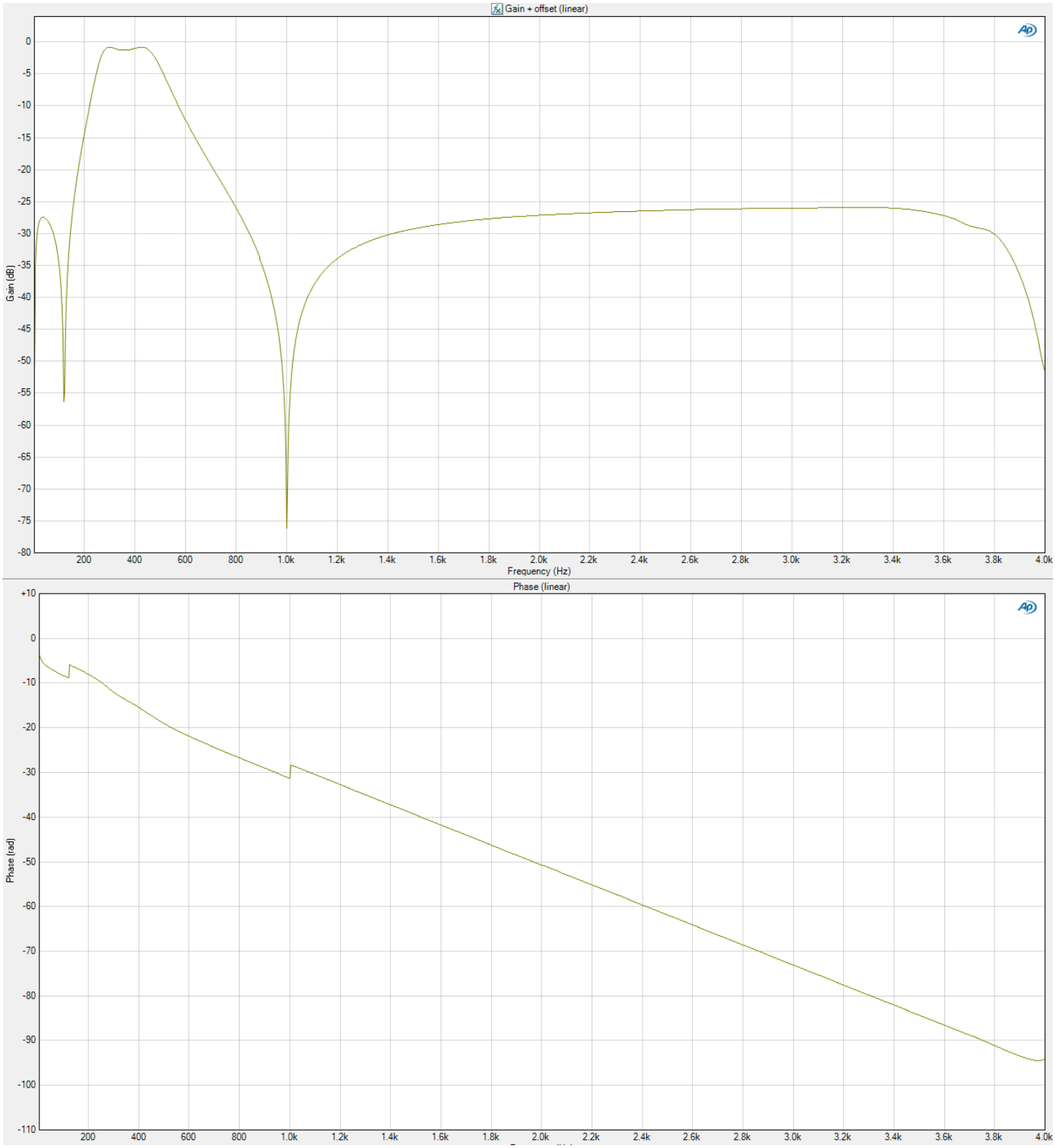

Figure 2.5: Spectrum output of the Direct Form II Transposed implementation.

Optimisation level	Filter order	Runtime (cycles)
none	4	298
none	8	530
none	16	994
o2	4	146
o2	8	172
o2	16	212

Table 2.2: Direct form II transposed IIR filter runtimes

Solving this with least squares for a constant and per-order cycle costs, we get:

- No optimisations: $66 + 58 \times (order)$
- o2: $120 + 6 \times (order)$

Therefore we see that the optimisations are making the loop more efficient but are increasing the static software pipeline setup cost (the pipeline prologue).

2.4 Performance Comparison

Comparing performance of o2 optimised versions for both the direct form II and form II transposed:

- Direct Form II: $188 + 31 \times (order)$
- Direct Form II Transposed: $120 + 6 \times (order)$

We first note that the transposed version is significantly faster and will scale to higher order filters significantly better.

The first reason is that the usual form II version has two serial mac loops (breaken into four loops in this implementation), one to accumulate the value to be delayed and the second to calculate the output:

```
    for (; vPtr < vEnd; ++vPtr, ++aPtr)
        *vOffset -= (*aPtr) * (*vPtr);

    for (vPtr = v; aPtr < aEnd; ++vPtr, ++aPtr)
        *vOffset -= (*aPtr) * (*vPtr);

    // calculate output y
    // y(n) = b0*v(n) + b1*v(n-1) + b2*v(n-2) + ...
    for (; vOffset < vEnd; ++vOffset, ++bPtr)
        output += (*bPtr) * (*vOffset);

    for (vOffset = v; bPtr < bEnd; ++vOffset, ++bPtr)
        output += (*bPtr) * (*vOffset);
```

Disassembly for one of the four loops' body is as follows:

```
0x0000D064:    020C3765            LDDW.D1T1       *A3++[1],A5:A4
0x0000D068:    021837E6  ||        LDDW.D2T2       *B6++[1],B5:B4
0x0000D06C:    03200364            LDDW.D1T1       *+A8[0],A7:A6
0x0000D070:    00004000            NOP             3
0x0000D074:    02109700            MPYDP.M1X       A5:A4,B5:B4,A5:A4
0x0000D078:    00010000            NOP             9
```

```
0x0000D07C:   0210C338            SUBDP.L1      A7:A6,A5:A4,A5:A4
0x0000D080:   00000000            NOP
0x0000D084:   2003E05A    [ B0]   SUB.L2        B0,1,B0
0x0000D088:   2FFFFC92    [ B0]   B.S2          C$L10 (PC-28 = 0x0000d064)
0x0000D08C:   00004000            NOP           3
0x0000D090:   02200274            STW.D1T1      A4,*+A8[0]
0x0000D094:   02A02274            STW.D1T1      A5,*+A8[1]
```

Therefore we see that only one C code iteration is performed per a machine level loop iteration. In addition, the setup and drain code for the loops increases the overheads.

The transposed version on the other hand uses one mac loop for operation:

```
    for (; i < N-2; i++)
        v[i] = v[i+1] + b[i+1]*x - a[i+1]*y;

    v[N-2] = b[N-1]*x - a[N-1]*y;
```

Therefore, at o2, a tight mac loop for the transposed version will scale well to higher order filters, as indeed can be seen from the disassembly of the loop body:

```
0x0000CF94:   022437E7            LDDW.D2T2     *B9++[1],B5:B4
0x0000CF98:   030C3764  ||        LDDW.D1T1     *A3++[1],A7:A6
0x0000CF9C:   00006000            NOP           4
0x0000CFA0:   02100703            MPYDP.M2      B1:B0,B5:B4,B5:B4
0x0000CFA4:   03188700  ||        MPYDP.M1      A5:A4,A7:A6,A7:A6
0x0000CFA8:   00006000            NOP           4
0x0000CFAC:   032033E6            LDDW.D2T2     *++B8[1],B7:B6
0x0000CFB0:   00006000            NOP           4
0x0000CFB4:   0318D31A            ADDDP.L2X     B7:B6,A7:A6,B7:B6
0x0000CFB8:   0000A000            NOP           6
0x0000CFBC:   0210C33A            SUBDP.L2      B7:B6,B5:B4,B5:B4
0x0000CFC0:   00000000            NOP
0x0000CFC4:   8087E058    [ A1]   SUB.L1        A1,1,A1
0x0000CFC8:   8FFFFA90    [ A1]   B.S1          C$L2 (PC-44 = 0x0000cf94)
0x0000CFCC:   00004000            NOP           3
0x0000CFD0:   022040F6            STW.D2T2      B4,*-B8[2]
0x0000CFD4:   02A020F6            STW.D2T2      B5,*-B8[1]
```

We note that the compiler fits the entire C code loop iteration into one machine level loop iteration, therefore we have much better resource utilisation and can do all of the following per iteration: three loads, two multiplies, addition, substraction.

There is still a static cost to pay for loop setup for the transposed implementation, but there are less loops so the penalty is incurred only once.

A Code Listing

A.1 Matlab Single-Pole Low Pass Filter Coefficient Calculation

```
clear;
format long e;
fs = 8000; % sampling frequency
Ts=1/fs;  % sampling period
R=1000; C=1e-6; % RC values

% S-plane coefficients
B = [0  1];
A = [R*C  1];

%plot
figure;

% plot s-plane frequency response
w = linspace(0, pi*fs, 5012);
h = freqs(B,A,w);

% Z-plane coefficients
a = [1  (Ts-2*R*C)/(Ts+2*R*C)];
b = [Ts/(Ts+2*R*C)  Ts/(Ts+2*R*C)];

% plot z-plane
[H, omega] = freqz(b,a,5012,fs);

subplot(2,1,1) , plot(w/(2*pi), mag2db(abs(h)), omega,mag2db(abs(H)));
xlim([0, fs/2]);
xlabel('Frequency (Hz)');
ylabel('Gain (dB)');
legend('Analogue Filter','Digital Filter');
grid on;
subplot(2,1,2),plot(w/(2*pi),unwrap(angle(h)), omega, unwrap(angle(H)));
xlim([0, fs/2]);
xlabel('Frequency (Hz)');
ylabel('Phase (radians)');
legend('Analogue Filter','Digital Filter');
grid on;
```

A.2 Single-Pole Low Pass Filter Code

```
/*********************************************************************************
              DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                        IMPERIAL COLLEGE LONDON

               EE 3.19: Real Time Digital Signal Processing
                  Dr Paul Mitcheson and Daniel Harvey

                                LAB 5 - Single Pole LPF
 *********************************************************************************/

/************************** Pre-processor statements ***************************/

#include <stdlib.h>
```

```
#include <stdio.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

// math library (trig functions)
#include <math.h>

// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

/******************************* Global declarations ********************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
       /*************************************************************************/
       /*   REGISTER              FUNCTION            SETTINGS         */
       /*************************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                 */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                 */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                 */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                 */\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off      */\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on      */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit               */\
    0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ               */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                  */\
       /*************************************************************************/
};


// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;


/******************************* Filter Stuff ********************************/
// The order of the FIR filter +1
#define N 2

// coefficients
double a = -0.8823529411764705;
double b[] = {0.05882352941176471, 0.05882352941176471};

// define the buffers
double inputBuffer[N] = {0};
double outputBuffer = 0;

 /******************************* Function prototypes ********************************/
void init_hardware(void);
void init_HWI(void);
```

```
71  void ISR_AIC(void);
72  Int16 convoluteNonCircular(void);
73  /********************************* Main routine *********************************/
74  void main(){
75
76
77    // initialize board and the audio port
78    init_hardware();
79
80    /* initialize hardware interrupts */
81    init_HWI();
82
83    /* loop indefinitely, waiting for interrupts */
84    while(1)
85    {};
86
87  }
88
89  /********************************* init_hardware() *********************************/
90  void init_hardware()
91  {
92      // Initialize the board support library, must be called first
93      DSK6713_init();
94
95      // Start the AIC23 codec using the settings defined above in config
96      H_Codec = DSK6713_AIC23_openCodec(0, &Config);
97
98    /* Function below sets the number of bits in word used by MSBSP (serial port) for
99    receives from AIC23 (audio port). We are using a 32 bit packet containing two
100   16 bit numbers hence 32BIT is set for  receive */
101   MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
102
103   /* Configures interrupt to activate on each consecutive available 32 bits
104   from Audio port hence an interrupt is generated for each L & R sample pair */
105   MCBSP_FSETS(SPCR1, RINTM, FRM);
106
107   /* These commands do the same thing as above but applied to data transfers to
108   the audio port */
109   MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
110   MCBSP_FSETS(SPCR1, XINTM, FRM);
111
112
113 }
114
115 /********************************* init_HWI() *********************************/
116 void init_HWI(void)
117 {
118   IRQ_globalDisable();       // Globally disables interrupts
119   IRQ_nmiEnable();         // Enables the NMI interrupt (used by the debugger)
120   IRQ_map(IRQ_EVT_RINT1,4);    // Maps an event to a physical interrupt
121   IRQ_enable(IRQ_EVT_RINT1);     // Enables the event
122   IRQ_globalEnable();        // Globally enables interrupts
123
124 }
125
126 /******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE**********************/
127
```

```
128 void ISR_AIC(void){
129   double sample = mono_read_16Bit();  // read
130
131   // Move the input buffer
132   inputBuffer[1] = inputBuffer[0];
133   inputBuffer[0] = sample;
134
135   // Calculate the difference equation and save to buffer
136   outputBuffer = b[0]*inputBuffer[0] + b[1]*inputBuffer[1] - a*outputBuffer;
137
138
139   mono_write_16Bit(outputBuffer);
140 }
```

A.3 Matlab Bandpass Filter Coefficient Calculation

```
1  fs = 8000;  % sampling frequency
2  Wp = [ 2*280/fs, 2*460/fs ];    % normalised passband frequencies
3  Rp = 0.5;   % passband ripple
4  Rs = 25;    % stopband attenuation
5  n = 4;      % 4th order
6
7
8  [z,p,k] = ellip(n/2,Rp,Rs,Wp);
9
10 [b,a] = zp2tf(z,p,k); % convert zeroes and poles form to transfer function
11 Hd=dfilt.df2(b,a); % create a discrete-time direct form II filter
12 Hdt = dfilt.df2t(b,a); % direct-form II transposed
13
14 fvtool(Hd); % filter visualisation tool
15 % plot
16 figure;
17 [H, omega] = freqz(b,a,5012, fs);
18
19 subplot(2,1,1) , plot(omega,mag2db(abs(H)));
20 xlim([0, 0.5*fs]);
21 xlabel('Frequency (Hz)');
22 ylabel('Gain (dB)');
23 grid on;
24 subplot(2,1,2),plot(omega, unwrap(angle(H)));
25 xlim([0, 0.5*fs]);
26 xlabel('Frequency (Hz)');
27 ylabel('Phase (radians)');
28 grid on;
29
30 % write coefficients
31 handle = fopen('coeff.h', 'w+');
32 fwrite(handle, 'double a[] = {');
33 fclose(handle);
34 dlmwrite('coeff.h', a, '-append', 'delimiter', ',', 'precision', 16);
35 handle = fopen('coeff.h', 'a');
36 fwrite(handle, sprintf('};\ndouble b[] = {'));
37 fclose(handle);
38 dlmwrite('coeff.h', b, '-append', 'delimiter', ',', 'precision', 16);
39 handle = fopen('coeff.h', 'a');
40 fwrite(handle, sprintf('};\n#define N sizeof(a)/sizeof(double)'));
41 fclose(handle);
```

A.4 Bandpass Filter Coefficient Header

```
double a[] = {1,-3.645104750379453,5.132966234332519,-3.305327460089965,0.8231195155333595
};
double b[] =
    {0.05747532226760528,-0.1954257629940412,0.2762188033364942,-0.1954257629940413,0.05747532226760533

};
#define N sizeof(a)/sizeof(double)
```

A.5 Bandpass Filter Direct Form II Implementation

```
/*************************************************************************
             DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                     IMPERIAL COLLEGE LONDON

              EE 3.19: Real Time Digital Signal Processing
                 Dr Paul Mitcheson and Daniel Harvey

                                LAB 5 - Direct form II
 *************************************************************************/

/*************************** Pre-processor statements ***************************/

#include <stdlib.h>
#include <stdio.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

// math library (trig functions)
#include <math.h>

// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

/******************************* Global declarations ********************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
       /*****************************************************************/
       /*   REGISTER              FUNCTION            SETTINGS          */
       /*****************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                 */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                 */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                 */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                 */\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off       */\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on       */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit                */\
```

```
    0x008d,  /* 8 SAMPLERATE Sample rate control            8 KHZ                */\
    0x0001   /* 9 DIGACT     Digital interface activation   On                   */\
       /*****************************************************************/
};


// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;


/******************************* Filter Stuff ********************************/
// include coefficient
#include "../PartII/coeff.h"

double *v;  // pointer to the delay line buffer (circular)

int index = 0; // index to the circular buffer

 /***************************** Function prototypes ***************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC(void);
double IIRFilter(double);
/********************************** Main routine *****************************/
void main(){
  // initialise the delay buffer
  v = (double *) calloc(N, sizeof(double));

  // initialize board and the audio port
  init_hardware();

  /* initialize hardware interrupts */
  init_HWI();

  /* loop indefinitely, waiting for interrupts */
  while(1)
  {};

}

/********************************* init_hardware() ***************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

  /* Function below sets the number of bits in word used by MSBSP (serial port) for
  receives from AIC23 (audio port). We are using a 32 bit packet containing two
  16 bit numbers hence 32BIT is set for  receive */
  MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

  /* Configures interrupt to activate on each consecutive available 32 bits
  from Audio port hence an interrupt is generated for each L & R sample pair */
  MCBSP_FSETS(SPCR1, RINTM, FRM);
```

```
103
104   /* These commands do the same thing as above but applied to data transfers to
105   the audio port */
106   MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
107   MCBSP_FSETS(SPCR1, XINTM, FRM);
108
109
110 }
111
112 /********************************* init_HWI() ************************************/
113 void init_HWI(void)
114 {
115   IRQ_globalDisable();       // Globally disables interrupts
116   IRQ_nmiEnable();         // Enables the NMI interrupt (used by the debugger)
117   IRQ_map(IRQ_EVT_RINT1,4);   // Maps an event to a physical interrupt
118   IRQ_enable(IRQ_EVT_RINT1);     // Enables the event
119   IRQ_globalEnable();        // Globally enables interrupts
120
121 }
122
123 /******************** ISR and filter code ***********************/
124
125 void ISR_AIC(void){
126     double output = IIRFilter(mono_read_16Bit());
127   mono_write_16Bit((Int16) output);
128 }
129
130 // based on difference equation at https://ccrma.stanford.edu/~jos/fp/Direct_Form_II.html
131 double IIRFilter(double input){
132     double* vPtr = v + index + 1;          // loop index pointer
133   double* vOffset = v + index;  // current v to write to
134     double* vEnd = v+N;    // one element after end of buffer
135     double* aPtr = a+1;    // pointer to a1 (ao is not used for calculation)
136     double* aEnd = a+N;    // one element after end of a
137     double* bPtr = b;      // pointer to b0
138     double* bEnd = b+N;    // one element after end of b
139     double output = 0;     // output
140
141     *vOffset = input; // read and store
142
143   // calculate v for the current input x
144     // v(n) = x(n) - a1 * v(n-1) - a2 * v(n-2) - ...
145     for (; vPtr < vEnd; ++vPtr, ++aPtr)
146         *vOffset -= (*aPtr) * (*vPtr);
147
148     for (vPtr = v; aPtr < aEnd; ++vPtr, ++aPtr)
149         *vOffset -= (*aPtr) * (*vPtr);
150
151     // calculate output y
152     // y(n) = b0*v(n) + b1*v(n-1) + b2*v(n-2) + ...
153     for (; vOffset < vEnd; ++vOffset, ++bPtr)   // reuse vOffset pointer now
154         output += (*bPtr) * (*vOffset);
155
156     for (vOffset = v; bPtr < bEnd; ++vOffset, ++bPtr)   // reuse vOffset pointer now
157         output += (*bPtr) * (*vOffset);
158
159     // decrement index
```

```
160     index = (index == 0) ? N-1 : index-1;
161
162   return output;
163 }
```

A.6 Bandpass Filter Direct Form II Transposed Implementation

```
1  /*************************************************************************
2               DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
3                       IMPERIAL COLLEGE LONDON
4
5                EE 3.19: Real Time Digital Signal Processing
6                   Dr Paul Mitcheson and Daniel Harvey
7
8                              LAB 5 - Direct form II Transposed
9   *************************************************************************/
10
11 /*************************** Pre-processor statements ***************************/
12
13 #include <stdlib.h>
14 #include <stdio.h>
15 //  Included so program can make use of DSP/BIOS configuration tool.
16 #include "dsp_bios_cfg.h"
17
18 /* The file dsk6713.h must be included in every program that uses the BSL.  This
19    example also includes dsk6713_aic23.h because it uses the
20    AIC23 codec module (audio interface). */
21 #include "dsk6713.h"
22 #include "dsk6713_aic23.h"
23
24 // math library (trig functions)
25 #include <math.h>
26
27 // Some functions to help with writing/reading the audio ports when using interrupts.
28 #include <helper_functions_ISR.h>
29
30 /*************************** Global declarations ***************************/
31
32 /* Audio port configuration settings: these values set registers in the AIC23 audio
33    interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
34 DSK6713_AIC23_Config Config = { \
35        /*****************************************************************/
36        /*   REGISTER              FUNCTION            SETTINGS         */
37        /*****************************************************************/\
38     0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                */\
39     0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                */\
40     0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                */\
41     0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                */\
42     0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
43     0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off       */\
44     0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on       */\
45     0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit                */\
46     0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ                 */\
47     0x0001   /* 9 DIGACT     Digital interface activation    On                    */\
48        /*****************************************************************/
49 };
50
```

```
51
52 // Codec handle:- a variable used to identify audio interface
53 DSK6713_AIC23_CodecHandle H_Codec;
54
55
56 /******************************* Filter Stuff ********************************/
57 // include coefficient
58 #include "../PartII/coeff.h"
59
60 double *v;  // pointer to the buffer
61
62  /****************************** Function prototypes ********************************/
63 void init_hardware(void);
64 void init_HWI(void);
65 void ISR_AIC(void);
66 double IIRFilter(double);
67 /********************************* Main routine *********************************/
68 void main(){
69   // initialise the buffer
70   v = (double *) calloc(N-1, sizeof(double));
71
72   // initialize board and the audio port
73   init_hardware();
74
75   /* initialize hardware interrupts */
76   init_HWI();
77
78   /* loop indefinitely, waiting for interrupts */
79   while(1)
80   {};
81
82 }
83
84 /********************************* init_hardware() ******************************/
85 void init_hardware()
86 {
87     // Initialize the board support library, must be called first
88     DSK6713_init();
89
90     // Start the AIC23 codec using the settings defined above in config
91     H_Codec = DSK6713_AIC23_openCodec(0, &Config);
92
93   /* Function below sets the number of bits in word used by MSBSP (serial port) for
94   receives from AIC23 (audio port). We are using a 32 bit packet containing two
95   16 bit numbers hence 32BIT is set for  receive */
96   MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
97
98   /* Configures interrupt to activate on each consecutive available 32 bits
99   from Audio port hence an interrupt is generated for each L & R sample pair */
100  MCBSP_FSETS(SPCR1, RINTM, FRM);
101
102  /* These commands do the same thing as above but applied to data transfers to
103  the audio port */
104  MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
105  MCBSP_FSETS(SPCR1, XINTM, FRM);
106
107
```

```
108 }
109
110 /********************************* init_HWI() ***********************************/
111 void init_HWI(void)
112 {
113   IRQ_globalDisable();       // Globally disables interrupts
114   IRQ_nmiEnable();         // Enables the NMI interrupt (used by the debugger)
115   IRQ_map(IRQ_EVT_RINT1,4);    // Maps an event to a physical interrupt
116   IRQ_enable(IRQ_EVT_RINT1);     // Enables the event
117   IRQ_globalEnable();        // Globally enables interrupts
118
119 }
120
121 /******************** ISR and filter code ***********************/
122
123 void ISR_AIC(void){
124     double output = IIRFilter(mono_read_16Bit());
125   mono_write_16Bit((Int16) output);
126 }
127
128 // based on Matlab code given at http://ocw.mit.edu/courses/mechanical-engineering/2-161-signal-
      processing-continuous-and-discrete-fall-2008/lecture-notes/lecture_20.pdf
129 double IIRFilter(double x){
130     double y = 0;   // output
131     int i = 0;  // loop index
132   y = v[0] + b[0]*x;
133
134     // update buffer for next iteration
135     for (; i < N-2; i++)
136         v[i] = v[i+1] + b[i+1]*x - a[i+1]*y;
137
138     v[N-2] = b[N-1]*x - a[N-1]*y;
139   return y;
140 }
```