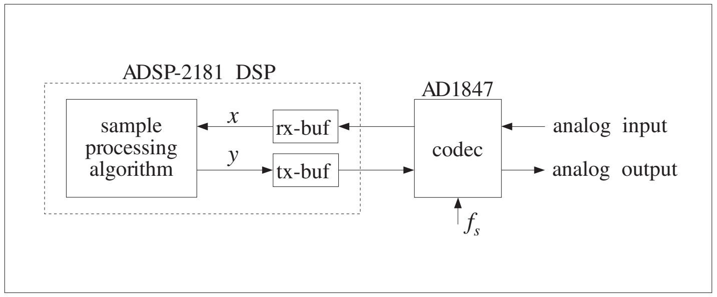
RUTGERS UNIVERSITY
The State University of New Jersey
School of Engineering
Department of Electrical and Computer Engineering
94 Brett Road, Piscataway, NJ 08854-8058
Tel. 732-445-5017, e-mail: `orfanidi@ece.rutgers.edu`

332:348 — Digital Signal Processing Laboratory

Spring 2005

ADSP-2181 Experiments

Sophocles J. Orfanidis



`http://www.ece.rutgers.edu/~orfanidi/ezkitl/ezkitl.html`

Contents

1. Introduction

This manual describes a number of hardware experiments illustrating the concrete implementation of various DSP algorithms on the ADSP-2181 chip.

The experiments include quantization and aliasing effects; the circular buffer implementation of delays, FIR, and IIR filters; the canceling of periodic interference with notch filters; wavetable generators; and several audio effects, such as comb filters, flangers and phasers, plain, allpass, and lowpass reverberators, Schroeder's reverberator, and several multi-tap, multi-delay, and stereo-delay type effects, as well as the Karplus-Strong string algorithm. All of the experiments are based on the text [1].

Our aim in these experiments is not to necessarily write the most efficient assembly code, but rather to show beginning DSP students how straightforward and fun it is to program a DSP chip and hear the algorithms in action. Thus, we have at times sacrificed efficiency for clarity.

To facilitate the programming of these applications, we have written a number of assembly code macros that closely parallel some of the C routines in the text, such as `cdelay` and `tap`, and allow the manipulation of circular delay-line buffers and the building up of more complex block diagrams. Our use of circular buffers and pointers is identical to that in the text (e.g., filter coefficients and states are stored in forward order and a delay is implemented by decrementing the buffer pointer.) A number of DOS utilities are also included, such as decimal-to-hex format converters and wavetable generators.

All of the included programs are based on the talkthru example program given in the EZ-KIT Lite Reference Manual [2]. All processor and codec initialization details have been hidden away in two include-files, `begin.dsp` and `end.dsp`, simplifying the structure of the programs and allowing the students to concentrate on the translation of their sample processing algorithm to assembly code.

Several reference resources are available. Besides the EZ-KIT Reference Manual and Data Sheets included in the kits, the following reference books may be obtained (in PDF format) from the Analog Devices web page `www.analog.com`:

1. ADSP-2100 Family User's Manual (also included in the newest kits.)
2. DSP Applications Using the ADSP-2100 Family, vols. 1 & 2.

2. Getting Started

2.1. Running DSP Programs

The process of running a signal processing algorithm on the ADSP-2181 chip of the EZ-KIT Lite board consists of three stages:

1. Creating a text source file containing the assembly code implementation of the algorithm. The default filename extension is .dsp. The DOS edit or emacs text editors may be used to edit the source file. They can be invoked by the DOS commands:

```
edit  filename.dsp
emacs filename.dsp
```

2. Compiling and linking the source file using the assembler and linker programs `asm21.exe` and `ld21.exe`. These operations have been automated into the DOS batch file `ezk.bat`, which can be invoked as follows:

   ```
   ezk filename
   ```

   where the source is `filename.dsp`. The result of these operations is the executable file `filename.exe`. (This is executable on the 2181 chip, not under DOS.)

3. Loading the executable file onto the chip via the serial connection using Dwight Elvey's public-domain loader program `ezld.com` found in [3]. The operation has again been automated by a batch file, `ezl.bat`, and is invoked from DOS as follows:

   ```
   ezl filename
   ```

   This loads `filename.exe` onto the processor and begins execution immediately. One can also load the program and enter a menu-oriented terminal program by the command:

   ```
   ezl filename t
   ```

The above utilities are in the path and can be executed from any subdirectory. The doskey synonym `dsp` changes to the directory `c:\adi_dsp\examples` below which are the subdirectories of the various examples.

There are a number of other useful utilities available. The batch file `ezs.bat` will compile and link a source code file and then run the simulator `sim2181.exe`:

```
ezs filename
```

where the full name `filename.dsp` is assumed. Some simulator example programs are in the examples subdirectory `sim`. The examples include quantization and downsampling, an order-3 delay, an FIR, and an IIR filter implemented using circular buffers. In the simulator, one can step through every instruction in the program and observe the contents of the delay-line buffer registers as they change from one input sample to the next. There is also the DOS batch file `mkezk.bat` with usage:

```
mkezk filename
```

It generates the file `filename.dsp` by copying into it a template file, `template.dsp`, from the `macros` subdirectory. The template file includes all the necessary DSP processor and codec initializations.

The student can then enter his/her choice of sampling rate f_s and insert a signal processing algorithm at the appropriate places inside the file. Several sampling rates may be selected, such as $f_s = 8, 16, 32, 44.1, 48$ kHz, and more.

2.2. Decimal to Hex Format Converters

The two DOS programs `dec2hex.exe` and `hex2dec.exe` allow the conversion from decimal format to 1.15 (or, in general, $a.b$) hexadecimal format and vice versa. Their C source code uses the routines `adc.c` and `dac.c` from [1] and is included in the Appendix.

Both programs can receive their input from `stdin` or from an input file containing the numbers to be converted (separated by spaces, tabs, or newlines). They return their output to `stdout` or an output file. Typical usage examples are:

```
dec2hex 1.15 < data.dec > data.hex
dec2hex 2.14 < data.dec > data.hex
dec2hex 1.15
```

where `data.dec` is a file containing the decimal numbers to be converted to the 1.15 format (or, the 2.14 format in the second case.) and the file `data.hex` contains the corresponding hex numbers. In the third case, the user must enter each decimal number (or, a group of numbers separated by spaces) in the command line followed by `<RET>` and enter a `<CTRL-Z>` or `<CTRL-C>` after the last number has been processed. Similarly, the usage of `hex2dec` is as follows:

```
hex2dec 1.15 < data.hex > data.dec
```

For example, the following column of decimal numbers fed into `dec2hex` 1.15 format produces the second column of hex numbers. In turn, the hex numbers fed into `hex2dec` 1.15 format will produce the third column of decimal numbers, which are the original numbers rounded to 16-bit accuracy:

1.00000	0x7fff	0.999969482421875	(largest positive number)
0.50000	0x4000	0.500000000000000	
0.40000	0x3333	0.399993896484375	
0.00003	0x0001	0.000030517578125	(smallest positive number)
-0.00003	0xffff	-0.000030517578125	(smallest negative number)
-0.40000	0xcccd	-0.399993896484375	
-0.50000	0xc000	-0.500000000000000	
-1.00000	0x8000	-1.000000000000000	(largest negative number)

The overall range of representable numbers depends on the format. For example, the above column of hex numbers fed into the `hex2dec 2.14` format will produce the third column scaled up by a factor of two. To determine which $a.b$ format to use, recall that all numbers to be converted must lie within the format's range:

$$-2^{a-1} \le x \le 2^{a-1} - 2^{-b}$$

Thus, for example, the ranges for the 1.15, 2.14, and 3.13 formats will be:

$$-1 \le x \le 1 - 2^{-15}$$

$$-2 \le x \le 2 - 2^{-14}$$

$$-4 \le x \le 4 - 2^{-13}$$

In general, using B total bits so that $a + b = B$, the $a.b$ format is simply a scaled up version of the standard $0.B$ two's complement binary representation $(b_1, b_2, \dots, b_B)$:

$$x = (-b_1 2^{-1} + b_2 2^{-2} + b_3 2^{-3} + \cdots + b_B 2^{-B}) \qquad (0.B \text{ format})$$
$$x = (-b_1 2^{-1} + b_2 2^{-2} + b_3 2^{-3} + \cdots + b_B 2^{-B})\, 2^a \qquad (a.b \text{ format})$$

where $a = 0, 1, \dots, B$. Alternatively, we may think of the $a.b$ format as the scaled down version of the $B.0$ two's complement integer format. Writing $a = B - b$ in the above expressions, we have:

$$x = (-b_1 2^{B-1} + b_2 2^{B-2} + b_3 2^{B-3} + \cdots + b_B) \qquad (B.0 \text{ format})$$
$$x = (-b_1 2^{B-1} + b_2 2^{B-2} + b_3 2^{B-3} + \cdots + b_B)\, 2^{-b} \qquad (a.b \text{ format})$$

where $b = 0, 1, \dots, B$. Fig. 2.1 shows the bit weighting factors and the placement of the fractional point for the four formats 0.16, 1.15, 2.14, and 3.13.

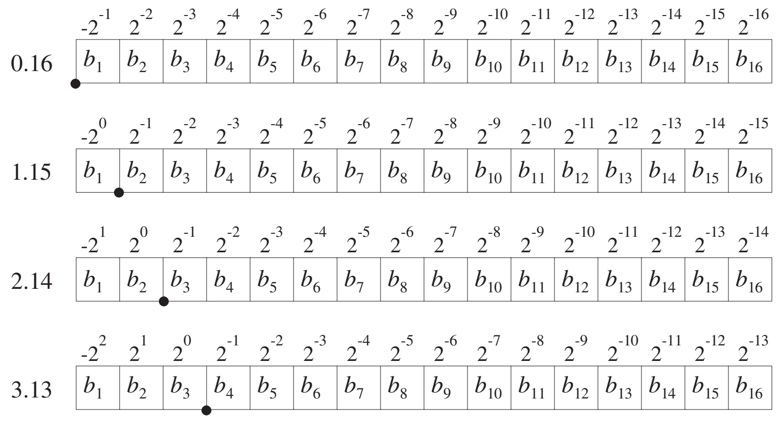


Fig. 2.1 Bit weights for 0.16, 1.15, 2.14, and 3.13 formats.

The programs `dec2hex` and `hex2dec` can be applied to any B which is a multiple of 4. For example, they can convert to/from the 1.7, 1.23, or 1.31 formats (which are 8-bit, 24-bit, and 32-bit formats) and their variants, such as the 2.6, 3.21, or 2.30 formats.

2.3. Wavetable Generator Programs

The DOS programs `sinetbl.exe`, `squartbl.exe`, `trapztbl.exe` are wavetable generators, generating *one period* of sinusoidal, square wave, and trapezoidal waveforms. Their C source code is included in the Appendix. Just typing the name of any of these utilities on the DOS command line followed by <RET> will display their usage. These programs can be used in conjunction with the `wavgen.dsp` macro to generate waveforms of different frequencies. Some examples of their usage are as follows:

```
sinetbl 0 1 2000 > sin.dec
sinetbl 1 1 2000 > cos.dec

squartbl 1 -1 1000 2000 1 > square1.dec
squartbl 1 -1 1000 2000 0 > square2.dec
```

The first generates one period of length 2000 samples of a unit-amplitude sinusoid, and the second a period of a cosinusoid, that is,

$$x_1(n) = \sin\left(\frac{2\pi n}{D}\right), \quad x_2(n) = \cos\left(\frac{2\pi n}{D}\right), \qquad n = 0, 1, \dots, D-1$$

with $D = 2000$. The third and fourth examples generate one period of the square waves:

$$x_3(n) = [\underbrace{1, 1, \dots, 1}_{\text{1000 ones}}, \underbrace{-1, -1, \dots, -1}_{\text{1000 minus-ones}}]$$

$$x_4(n) = [0, \underbrace{1, 1, \dots, 1}_{\text{999 ones}}, 0, \underbrace{-1, -1, \dots, -1}_{\text{999 minus-ones}}]$$

The square wave $x_3(n)$ jumps discontinuously from level 1 to -1, whereas $x_4(n)$ has the value 0 at the discontinuities. See Section 4.15 for more on this.

2.4. Subdirectory Structure

The default installation directory of the EZ-KIT Lite software is `c:\adi_dsp`. The various batch files, DOS utilities, and DSP macros for this lab are in the subdirectory `macros`. The various examples are in the subdirectory `examples`. The detailed contents of these directories are listed below:

```
c:\adi_dsp\macros:

    ezk.bat       ezl.bat       ezs.bat       mkezk.bat
    dec2hex.exe   hex2dec.exe   sinetbl.exe   squartbl.exe  trapztbl.exe
    uran.exe      dec2hex.c     hex2dec.c     sinetbl.c     squartbl.c
    trapztbl.c    uran.c        begin.dsp     ccan.dsp      cdelay.dsp
    cdir.dsp      cfir.dsp      dot.dsp       dspmac.dsp    end.dsp
    tap.dsp       tapin.dsp     wavgen.dsp    zero.dsp      template.dsp

c:\adi_dsp\examples:

    allpass   comb       delay      fir      flanger    guitar
    lowpass   multidel   multitap   notch    plain      quantize
    revdel    reverb     sim        stereo   wavtable
```

3. Instruction Set Tutorial

The ADSP-2181 processor has a powerful instruction set, which is summarized in the Quick Programmer's Reference section of the EZ-KIT Lite Reference Manual [2].

However, to get started writing simple filtering signal processing algorithms, one needs only a small subset of the full set.

Temporary variables, such as delay-line buffers and multiplier coefficients, can be stored in data memory (DM) or program memory (PM) from where they can be transferred back and forth to the processor's computational units where all the arithmetic operations are carried out.

3.1. Computational Units

The three computational units are the multiplier accumulator (MAC), the arithmetic logic unit (ALU), and the shifter. The MAC, ALU, and shifter registers that we will be using in our examples are the following:

```
mx0, mx1, my0, my1, mr, mr0, mr1, mr2     (MAC)
ax0, ax1, ay0, ay1, ar                    (ALU)
sr, sr0, sr1, si, se                      (shifter)
```

The wordlength of these registers is as follows:

```
16 bits:  mx0, mx1, my0, my1, mr0, mr1, ax0, ax1, ay0, ay1, ar, sr0, sr1, si
40 bits:  mr  (consists of mr2, mr1, mr0)
32 bits:  sr  (consists of sr1, sr0)
 8 bits:  mr2, se
```

The 40-bit accumulator register `mr` of the MAC consists of the three registers `mr0`, `mr1`, `mr2`. The result of multiplying two 16-bit numbers is a 32-bit number placed in the registers `mr1`, `mr0`, where `mr1` contains the 16 MSB bits and `mr0` the 16 LSB bits. The 8-bit register `mr2` is used for overflow bits. The subset of MAC operations that we will be using is:

```
mr = 0;                        (clear MAC accumulator)
mr = x * y (ss);               (multiply with 32-bit precision)
mr = x * y (rnd);              (multiply, round to 16 bits, and put result in mr1)
mr = mr + x * y (ss);          (multiply/accumulate with 32-bit precision)
mr = mr + x * y (rnd);         (multiply/accumulate and round to 16 bits)
mr = mr - x * y (ss);
mr = mr - x * y (rnd);
if mv sat mr;                  (saturate if overflow occurs)
```

where `x` and `y` may be one of the following registers:

```
x:  mx0, mx1, mr0, mr1, mr2, ar, sr0, sr1
y:  my0, my1
```

Note that `x` must always be written to the left of `y`. The `(ss)` qualifier treats the operands as signed 1.15 numbers and the MAC operation is carried out to full double-precision. The qualifier `(rnd)` causes the double-precision 32-bit result to be rounded off to its 16 most significant bits, with the result residing in `mr1`. The last, `if mv sat mr`, instruction saturates `mr` to its largest (positive or negative) value whenever the overflow flag `mv` is raised. The subset of ALU operations that we will be using is:

```
ar = x + y;
ar = x - y;
ar = y - x;
```

where the permissible registers for `x` and `y` are:

```
x:  ar, ax0, ax1, mr0, mr1, mr2, sr0, sr1
y:  ay0, ay1
```

The shifter instructions that we will be using are the arithmetic shifts:

```
sr = ashift x by exp (hi);     (multiply x by 2^exp with MSB-result in sr1)
sr = ashift x (hi);            (get exp from se register)
```

where `x` and `exp` are:

```
  x:  si, sr0, sr1, ar, mr0, mr1, mr2
exp:  any signed integer, such as, ±1, ±2, ...
```

The effect of this instruction is to scale `x` by the factor 2^{exp} and place the result in `sr`, with `sr1` containing the 16 MSB bits. In the second `ashift` instruction, the value of the exponent `exp` has been preloaded into the 8-bit exponent register `se` and is read from there.

3.2. Data Transfers

Any of the above 16-bit computational registers can be loaded with a numerical value (in 1.15 format), or transferred back and forth to another such register, or to data memory DM or program memory PM. For example, suppose we have declared some constants and some variables in data memory DM:

```
.const a = 0x6000;          {a = 0.75 in decimal format}
.const D = 3;
.var/dm w[D+1];             {4-dimensional linear buffer w[i], i=0,1,2,3}
.var/dm x, y;               {temporary variables}
```

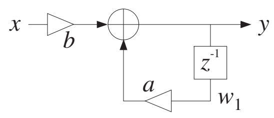
Then, the following examples of data transfers are all executable in one cycle each:

```
mr1 = 0;             {load mr1 with zero}
my1 = a;             {load my1 with the constant a}
ax1 = 0x4000;        {load ax1 with the value 0x4000 = 0.50 in decimal}
ar = sr1;            {load ar with content of sr1}
mx1 = ay0;           {load mx1 with content of ay0}
mr1 = dm(x);         {load mr1 with content of DM location x}
dm(y) = my1;         {load DM location y with the content of my1}
mr1 = dm(w);         {load mr1 with content of buffer location w[0]}
mr1 = dm(w+1);       {load mr1 with content of buffer location w[1]}
dm(w+2) = mr1;       {load buffer location w[2] with content of mr1}
```

As a signal processing example, consider the implementation of the first-order filter:

$$y(n) = ay(n-1) + bx(n), \qquad \text{where } a = 0.75,\ b = 0.25$$

Introducing the internal state $w_1(n) = y(n-1)$, and noting that the next state is $w_1(n+1) = y(n)$, we may draw a block diagram realization and write the sample processing algorithm as follows:



```
for each input sample x do:
    $y = aw_1 + bx$
    $w_1 = y$
```

To implement this example on the EZ-KIT Lite board, we must be able to read input samples from the codec and write the output samples back to the codec.

In the sample talkthru program from the EZ-KIT Lite Manual [2], this is accomplished by using the so-called *autobuffering* technique of the ADSP-2181 serial ports, which allows the samples from the stereo codec's analog input to be written to DM memory at a predefined receive-buffer location, called `rx_buf`. Samples from the left and right channels are stored in `rx_buf+1` and `rx_buf+2`.

After both stereo samples have been written to DM, a receive-interrupt is issued to the DSP, which then initiates an interrupt service routine that implements the DSP sample processing algorithm to be applied to the input samples. The sampling rate f_s (selected through the codec's format register) determines the rate at which these interrupts are being issued to the DSP, and therefore, the rate at which the sample processing algorithm is repeatedly executed, generating the output samples.

The processed left/right output samples are then written back to some predefined transmit-buffer location in DM, called `tx_buf`, from where they are passed on to the DSP's serial port and on to the codec to be converted to analog output. The overall sequence of operations is shown in Fig. 3.1.

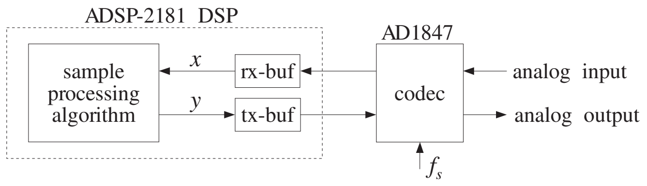


Fig. 3.1 Sample by sample processing on the EZ-KIT Lite board.

For our simple example, we start with the variable declarations and initializations:

```
.const a = 0x6000;              {a = 0.75}
.const b = 0x2000;              {b = 0.25}
.var/dm w1;                     {filter's internal state}

ax0 = 0;
dm(w1) = ax0;                   {initialize w1 to zero}
```

Then, the sample processing algorithm (for the right channel only) will be:

```
my0 = dm(rx_buf + 2);           {get right input from codec}
```

```
mx0 = b;                        {read filter coefficient b into my0}
mr = mx0 * my0 (ss);            {mr = b * x}
mx0 = a;                        {read filter coefficient a}
my0 = dm(w1);                   {get internal state from DM}
mr = mr + mx0 * my0 (rnd);      {mr = y = a * w1 + b * x = output sample}
dm(w1) = mr1;                   {update state, w1 = y}

dm(tx_buf + 2) = mr1;           {send right output to codec}
```

Note that after the final MAC, only the rounded 16-bit MSB part of `mr`, contained in `mr1`, is sent to the codec and saved into the internal state memory location.

3.3. DAG Registers and Circular Buffers

Another way to transfer data back and forth from DM or PM memory is indirectly via the two sets of data address generator (DAG) registers. The address pointers or *index* (I) registers `i0,i1,i2,i3` are associated with DAG1 and can point only to DM memory, whereas `i4,i5,i6,i7` are associated with DAG2 and can point either to DM or PM memory. Associated with these address pointers, we also have the corresponding *modify* (M) registers `m0,m1,m2,m3` and `m4,m5,m6,m7`, and the corresponding *length* (L) registers `L0,L1,L2,L3` and `L4,L5,L6,L7`. They are all 14-bit registers. Note that `i0` and `i1` are reserved in the EZ-KIT Lite.

The data memory location pointed to by an I-register can be accessed by the instruction `dm(I,M)`, where after the access (read or write), the address is incremented or decremented automatically by an amount specified by the value of the M-register. For example, the instruction

```
mr1 = dm(i2, m2);
```

writes into `mr1` the contents of the DM memory location pointed to by i2 and then changes `i2` by an amount `m2`. If `m2=1`, then the new `i2` will point to the next memory location, and if `m2=-1`, it will point to the previous location. This post-modify scheme applies also to writing into memory. For example, the instruction

```
dm(i2, m2) = mr1;
```

writes the contents of `mr1` into the memory location pointed to by `i2` and then changes `i2` by an amount `m2`. A useful related instruction is the `modify` instruction, which allows `i2` to be changed by a desired amount `m2` without reading or writing data:

```
modify(i2, m2);
```

The DAG registers can keep track of the memory locations for both linear and circular buffer arrays. Thus, they are especially convenient for implementing circular delay-line buffers.

Fig. 3.2 shows how the DAG pointer `i2` points into a circular buffer, just like the usual pointer p of the text [1]. The value of `m2` specifies by how much `i2` is to move. As `i2` moves up or down within the buffer locations, it wraps around automatically if it exceeds the upper or lower bounds.

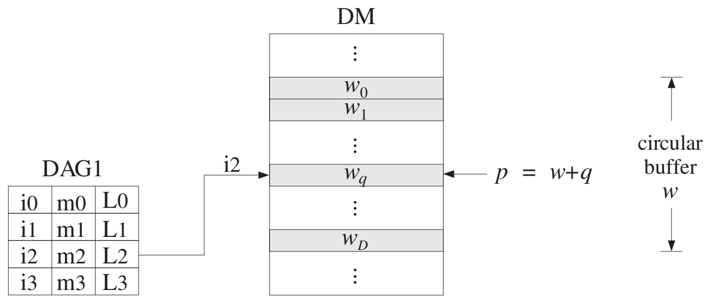


Fig. 3.2 DAG pointer into a circular delay-line buffer.

3.4. Defining and Initializing a Delay

To implement a delay of some maximum amount, say $D = 100$ samples, one must declare it as a circular $(D+1)$-dimensional array, then set one of the I-registers to point to the beginning of the buffer, and define the corresponding L-register to contain the length of the buffer:

```
.const D = 100;
.var/dm/circ w[D+1];        {placed in DM memory}

i2 = ^w;                    {i2 initially points to beginning of w}
L2 = %w;                    {L2 is set equal to the length of w}
```

The buffer may be initialized to zero by the do-loop:

```
m2 = 1;                          {post-increment i2 by one}
cntr = L2;
do loop until ce;                {repeat until counter expires}
   loop:  dm(i2, m2) = 0;        {put 0 in dm(i2,m2) and point to next i2}
```

The do-loop iterates `L2` times and therefore, because the buffer is circular, the pointer `i2` will wrap around completely and, upon exit, it will be pointing again at the beginning of the buffer `w`. This initialization loop has been made into a macro, `zero.dsp`, and is invoked by

```
zero(i2, m2, L2);
```

3.5. Reading and Writing a Delay-Line Buffer

Once the circular buffer `w` and pointer `i2` have been defined, the delay-line's tap outputs can be obtained by accessing the buffer entries relative to `i2`. For example, the value contained in the d-th tap, can be obtained by the instructions:

```
m2 = d; modify(i2, m2);          {go to location pointed to by i2 + d}
m2 =-d; mr1 = dm(i2, m2);        {put its content in mr1, then restore i2}
```

These steps have been put into a macro, `tap.dsp`, with usage:

```
tap(i2, m2, d, mr1);
```

where d is in the range $d = 0, 1, \ldots, D$, and `mr1` can be replaced by any other 16-bit computational register. Fig. 3.3 illustrates these operations. One caveat on the use of `tap` is that it cannot be used as described when the delay d is variable and is passed through a register (because we cannot set `m2 = -dreg`). See Section 4.16 for further discussion of this point and a remedy.

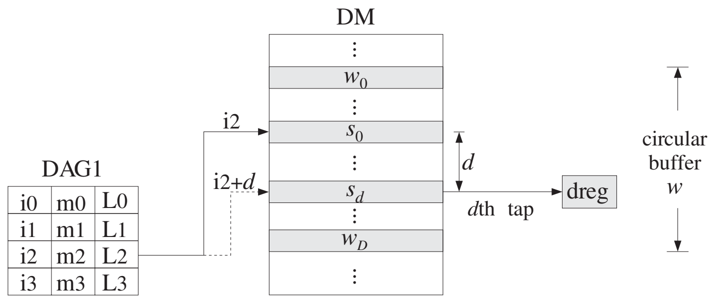


Fig. 3.3 Reading the dth tap of a circular delay-line buffer.

In filtering operations, we must put a new input sample into tap-0 before updating the delay line. Such operation can be accomplished by

```
m2 = 0; dm(i2, m2) = mx1;
```

which puts the value of `mx1` into the location pointed to by `i2` without post-modifying `i2`. These steps have been automated into another macro, `tapin.dsp`, with usage:

```
tapin(i2, m2, mx1);
```

where the `mx1` register can be replaced by any other 16-bit register. Fig. 3.4 illustrates this operation.

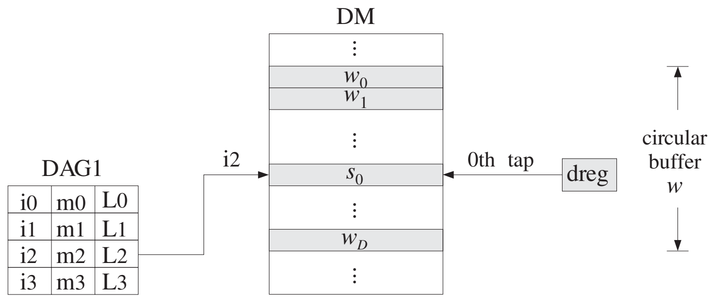


Fig. 3.4 Writing into tap-0 of a circular delay-line buffer.

3.6. Updating a Delay

Once the input to the delay-line buffer is in, we may update the delay by simply backshifting the pointer `i2`:

```
m2 = -1; modify(i2, m2);
```

Fig. 3.5 illustrates this operation. These two instructions have been placed into the macro `cdelay.dsp` with usage:

```
cdelay(i2, m2);
```

In summary, the following macros in the directory `c:\adi_dsp\macros` facilitate the operations required in circular delay lines:

```
zero(i2, m2, L2);          {clear delay line to zero}
tapin(i2, m2, mx1);        {put mx1 into tap-0 of delay line}
tap(i2, m2, d, mr1);       {get the d-th tap and put in mr1}
cdelay(i2, m2);            {update delay line}
```

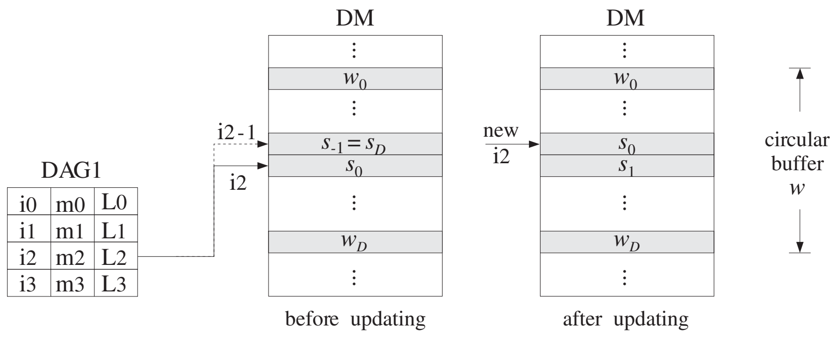


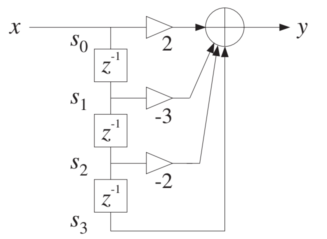
Fig. 3.5 Updating a circular delay-line buffer.

3.7. Multifunction Instructions and FIR Filters

Using the above circular buffer tools, we present a final example on implementing an order-3 FIR filter and we also discuss multifunction-type instructions. The I/O equation is:

$$y(n) = 2x(n) - 3x(n-1) - 2x(n-2) + x(n-3)$$

The block diagram and circular buffer version of its sample processing algorithm are shown below:



```
for each input x do:
    $*p = s_0 = x$
    $s_1 = \text{tap}(3, w, p, 1)$
    $s_2 = \text{tap}(3, w, p, 2)$
    $s_3 = \text{tap}(3, w, p, 3)$
    $y = 2s_0 - 3s_1 - 2s_2 + s_3$
    $\text{cdelay}(3, w, \&p)$
```

Since the range of the filter coefficients is $[-4, 4]$, we must convert them with the 3.13 format, resulting in the hex numbers:

$$\mathbf{h} = [2, -3, -2, 1] \rightarrow \texttt{[0x4000, 0xa000, 0xc000, 0x2000]}$$

Equivalently, these are the filter coefficients scaled down by 4 in order to fit within the 1.15 format, namely

$$\frac{1}{4}[2, -3, -2, 1] = [0.50, -0.75, -0.50, 0.25] \rightarrow \texttt{[0x4000, 0xa000, 0xc000, 0x2000]}$$

Because the MAC performs its multiplications in the 1.15 format, after the final output y is computed, it can (optionally) be scaled up by a factor of 4 with the help of the shifter. The following instructions declare the delay and filter coefficient circular buffers and initialize them:

```
.const M = 3;               {filter order}
.var/dm/circ w[M+1];        {delay-line buffer placed in DM}
.var/pm/circ h[M+1];        {filter coefficient buffer placed in PM}

.init h: 0x4000, 0xa000, 0xc000, 0x2000;        {coefficient values}

i2 = ^w;  L2 = %w;          {delay-line buffer pointer and length}
i4 = ^h;  L4 = %h;          {coefficient buffer pointer and length}

zero(i2, m2, L2);           {clear delay-line buffer to zero}
```

Assuming as in the previous example that the codec inputs and outputs come from receive and transmit buffers in DM, the sample processing algorithm can be translated to assembly language as follows:

```
mx1 = dm(rx_buf + 2);              {read right input from codec}

tapin(i2, m2, mx1);                {put mx1 into tap-0 of delay line}

m2 = 1; m4 = 1;
mr = 0, mx0 = dm(i2,m2),  my0 = pm(i4,m4);                  {s0,h0, next s1,h1}
mr = mr + mx0 * my0 (ss), mx0 = dm(i2,m2), my0 = pm(i4,m4);  {s1,h1, next s2,h2}
mr = mr + mx0 * my0 (ss), mx0 = dm(i2,m2), my0 = pm(i4,m4);  {s2,h2, next s3,h3}
mr = mr + mx0 * my0 (ss), mx0 = dm(i2,m2), my0 = pm(i4,m4);  {s3,h3, next s0,h0}
mr = mr + mx0 * my0 (rnd);                                   {mr = y}
if mv sat mr;
```

```
    cdelay(i2, m2);                    {update delay}

    sr = ashift mr1 by 2 (hi);         {scale output by factor of 2^2 = 4}

    dm(tx_buf + 2) = sr1;              {write right output to codec}
```

The four code lines involving a MAC operation and fetching data from DM and PM memories, are examples of *multifunction instructions* which are executable in one cycle, that is, 30 nsec. These instructions are separated by commas instead of semicolons.

The first multifunction instruction clears the MAC accumulator `mr` to zero, fetches the values of s_0, h_0 into the registers `mx0`, `my0`, and then post-increments the buffer pointers to point to the next buffer entries, that is, s_1, h_1. (All of that is done in one cycle.)

The next code line calculates the partial product `mr` $= h_0 s_0$, fetches s_1, h_1, and points to s_2, h_2. The next line updates the partial sum `mr` $= h_0 s_0 + h_1 s_1$, fetches s_2, h_2, and points to s_3, h_3. The next line updates the partial sum `mr` $= h_0 s_0 + h_1 s_1 + h_2 s_2$, fetches s_3, h_3, and wraps around to point to the beginning of the circular buffers, namely, to s_0, h_0. The next line performs the final accumulation `mr` $= h_0 s_0 + h_1 s_1 + h_2 s_2 + h_3 s_3$, and rounds the result to its 16 most significant bits, contained now in `mr1`. The result is saturated if overflow is detected.

Then, the delay line is updated by backshifting its pointer `i2`. Note that the argument `m2` of `cdelay(i2,m2)` is set internally to `m2=-1` in the macro.

Then, the computed output in `mr1` is scaled up by a factor of 4 by the shifter and the scaled result is placed in `sr1`, and finally sent out to the codec.

The repeated multifunction instructions can be replaced by a do-loop, which effectively performs the dot-product of the internal states with the filter coefficients:

```
    m2 = 1; m4 = 1;
    mr = 0, mx0 = dm(i2,m2), my0 = pm(i4,m4);
    cntr = M;                                         {M = filter order}
           do dotloop until ce;
    dotloop:   mr = mr + mx0 * my0 (ss), mx0 = dm(i2,m2), my0 = pm(i4,m4);
    mr = mr + mx0 * my0 (rnd);
    if mv sat mr;
```

These instructions have been collected into a macro, `dot.dsp`, with usage:

```
    dot(M, i4, m4, i2, m2);            {result returned in mr1}
```

Fig. 3.6 illustrates the dot product operation. Note that `i4`, pointing to the filter coefficients, cycles back to the beginning of the buffer, and `i2`, pointing to the filter states, cycles back to the 0th state.

In summary, the sample processing algorithm can be simplified to the three operations of: (a) reading the input sample into the delay line, (b) computing the dot product output, and (c) updating the delay line:

```
    mx1 = dm(rx_buf + 2);              {read right input from codec}

    tapin(i2, m2, mx1);                {put mx1 into tap-0 of delay line}
```

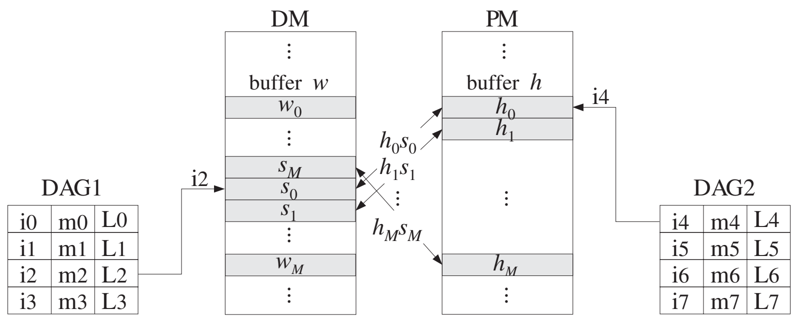


Fig. 3.6 Dot product $y = \mathbf{h}^T \mathbf{s}$ of state vector with filter vector.

```
    dot(M, i4, m4, i2, m2);            {compute output into mr1}
    cdelay(i2, m2);                    {update delay}

    sr = ashift mr1 by 2 (hi);         {scale output by factor of 2^2 = 4}

    dm(tx_buf + 2) = sr1;              {write right output to codec}
```

The three macros `tapin`, `dot`, and `cdelay`, have been combined into another macro, `cfir.dsp`, which implements an FIR filter. The above example would read in this case:

```
    mx1 = dm(rx_buf + 2);              {read right input from codec}

    cfir(M, i4, m4, i2, m2, mx1);      {input from mx1, output in mr1}

    sr = ashift mr1 by 2 (hi);         {scale output by factor of 2^2 = 4}

    dm(tx_buf + 2) = sr1;              {write right output to codec}
```

3.8. Linear Delay-Line Buffers

The DAG registers can also point to linear buffers. For example, the following instructions define the buffer and its pointer:

```
    .const D = 100;
    .var/dm w[D+1];                {placed in DM memory}

    i2 = ^w;                       {i2 points to beginning of w}
    L2 = 0;                        {L2 must be set to 0 for a linear buffer}
```

The post-modify feature is still present, except `i2` does not automatically wrap around upon reaching the end of the buffer. The simulator example `delex1.dsp` in the subdirectory `examples\sim` illustrates the operation of a linear delay line.

3.9. Concatenated Circular Buffers

Finally, we mention concatenated circular buffers, which can be used to store the denominator and numerator coefficients of an IIR filter. The following declaration defines two buffers each of length $M+1$ in PM:

```
.const M = 100;
.var/pm/circ a[M+1], b[M+1];

i4 = ^a;
L4 = 2*(M+1);
```

This declaration defines an extended circular buffer of double-length $2(M+1)$. The DAG pointer i4 will traverse both buffers a and b before wrapping around to the beginning of a. See Section 5.7 for more details.

4. Experiments

4.1. Sampling and Quantization

This section contains three types of experiments: (a) sampling and immediate playback (based on the EZ-KIT Lite's talkthru program), (b) input quantization effects, and (c) demonstration of aliasing effects by downsampling.

Talkthru

This experiment illustrates the sampling and immediate playback of an input audio signal. The following program, thru.dsp, is simply a copy of the template program, template.dsp:

```
{thru.dsp - talkthru program - may be used as template}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on mic2out.dsp from Analog Devices FTP site and the sample talkthru
 program of the EZ-KIT Lite Reference Manual.}

{--- define sampling rate in kHz: ---------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc85b; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- start processing input samples -------------------------------------}

wait: idle; jump wait;            {wait for interrupt and loop forever}
                                  {interrupt service routine starts here}
input_samples: ena sec_reg;       {enable secondary register set}
```

```
{--- read input samples from codec --------------------------------------}

   ax1 = dm(rx_buf + 1);           {left input sample}
   mx1 = dm(rx_buf + 2);           {right input sample}

{--- write output samples to codec --------------------------------------}

   dm(tx_buf + 1) = ax1;           {left output sample}
   dm(tx_buf + 2) = mx1;           {right output sample}

{--- return from interrupt ----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

It is the same as the example talkthru program of the EZ-KIT Lite Reference Manual [2]. The input samples from the codec are deposited in the receive-buffer locations in DM, rx_buf+1 and rx_buf+2, and then copied into the registers ax1 and mx1 for the left and right channels.

Then, without any further processing, the input samples are written back into the transmit-buffer locations tx_buf+1 and tx_buf+2 from where they are read by the codec and sent to the analog output.

Lab Procedure

Change directory into c:\adi_dsp\examples\quantize by the DOS commands:

```
dsp
cd quantize
```

Compile, link, and load this program with the DOS commands:

```
ezk thru
ezl thru
```

Speak into the mike.

Quantization Effects

The program quantize.dsp illustrates input quantization effects:

```
{quantize.dsp - quantization to B bits}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{--- choose sampling rate in kHz: ---------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc85b; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}
```

```
{--- define constants, variables, and buffers ----------------------------}

.const B = 6;                     {quantization bits per sample}
.const L = 16 - B;                {L least-significant bits thrown away}

{--- start processing input samples -------------------------------------}

wait: idle; jump wait;            {wait for interrupt and loop forever}
                                  {interrupt service routine starts here}
input_samples: ena sec_reg;       {enable secondary register set}

{--- read input samples from codec --------------------------------------}

   mr0 = dm(rx_buf + 1);          {left input sample}
   mr1 = dm(rx_buf + 2);          {right input sample}

{--- sample processing algorithm ----------------------------------------}

   sr = ashift mr0 by -L (hi);    {shift right by L bits}
   sr = ashift sr1 by  L (hi);    {shift left by L bits}
   mr0 = sr1;                     {requantized left sample}

   sr = ashift mr1 by -L (hi);    {shift right by L bits}
   sr = ashift sr1 by  L (hi);    {shift left by L bits}
   mr1 = sr1;                     {requantized right sample}

{--- write output samples to codec --------------------------------------}

   dm(tx_buf + 1) = mr0;          {left output sample}
   dm(tx_buf + 2) = mr1;          {right output sample}

{--- return from interrupt ----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

The input samples from the codec are available in 16-bit resolution. In this experiment, each input sample is re-quantized to B-bit resolution, where $1 \le B \le 16$, and then it is sent out to the codec. The number of least-significant bits that are thrown away are $L = 16 - B$.

The re-quantization operation is done conveniently by the shifter by first shifting to the right by L bits and then shifting to the left by L bits. This has the effect of filling the last L bits with zeros. The simulator program `quantex1.dsp` in the examples subdirectory `sim` steps through such a re-quantization operation.

Lab Procedure

a. Still in the `c:\adi_dsp\examples\quantize` directory, compile and run the above program with an initial choice of $B = 6$ bits. Speak into the mike and listen to the quantization noise.

   Repeat the above procedure by editing the program and choosing the successive values $B = 16, 15, \dots, 1$. For the higher values of B you will probably

   not hear any quantization noise. The noise should become more and more evident as you decrease B, especially below 8 bits. Make sure you listen to 1-bit and 2-bit speech.

b. The quantization operation is an example of a nonlinear memoryless operation on each input sample. Other examples of such nonlinear operations can be tried out.

   As a distortion example, modify the above program so that instead of quantizing each sample, it squares it, that is, $y(n) = x^2(n)$, and then it sends $y(n)$ to the output. Compile and run this program. Can you understand the types of distortions you are hearing in the frequency domain? Repeat when each input sample is cubed, that is, $y(n) = x^3(n)$. (You may need to scale up the output $y(n)$ using the shifter.)

Aliasing by Downsampling

The following program `dnsample.dsp` implements a downsampling operation where the sampling rate is reduced by a factor of M:

```
{dnsample.dsp - downsampling and aliasing by decimation}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{--- choose sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc85c; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const M = 6;                     {decimation ratio; fs_low = fs/M}
.var/dm/circ s[M];                {low-rate sampling pulse train}
.const A = 0x7fff;                {A=1, amplitude of sampling pulse}

i5 = ^s; L5 = %s; m5 = 1;         {low-rate sampling pulse train}

cntr = M;                         {initialize pulse train to zero}
do zero_s until ce;               {s = [0, 0, ..., 0]}
   zero_s:  dm(i5, m5) = 0;       {may also call zero(i5, m5, L5)}

ax0 = A;                          {set s[0] = A}
dm(s) = ax0;                      {s = [A, 0, ..., 0] = A and M-1 zeros}

{--- start processing input samples -------------------------------------}

wait: idle; jump wait;            {wait for interrupt and loop forever}
                                  {interrupt service routine starts here}
input_samples: ena sec_reg;       {enable secondary register set}

{--- read input samples from codec --------------------------------------}
```

```
   mr0 = dm(rx_buf + 1);           {left input sample}
   mr1 = dm(rx_buf + 2);           {right input sample}

{--- sample processing algorithm ----------------------------------------}

   ar = dm(i5, m5);                {get sampling pulse}
   ar = pass ar;                   {forces updating of AZ flag}
   if ne jump output;              {if AR != 0, write output}

   mr0 = 0;                        {else, write zero outputs}
   mr1 = 0;

{--- write output samples to codec --------------------------------------}

   output:

#if 1                                           {optional amplification}
   sr = ashift mr0 by 1 (hi);  mr0 = sr1;
   sr = ashift mr1 by 1 (hi);  mr1 = sr1;
#endif

   dm(tx_buf + 1) = mr0;                        {left output sample}
   dm(tx_buf + 2) = mr1;                        {right output sample}

{--- return from interrupt ----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

The re-sampling operation is accomplished by multiplying the input samples $x(n)$ at the initial rate f_s by a periodic re-sampling pulse train $s(n)$ consisting of unit impulses every M of the original samples and has zeros otherwise, that is,

$$s(n) = [1, \underbrace{0, 0, \dots, 0}_{M-1}, 1, \underbrace{0, 0, \dots, 0}_{M-1}, 1, \underbrace{0, 0, \dots, 0}_{M-1}, \dots]$$

The re-sampled input signal will be:

$$x'(n) = s(n)x(n)$$

It is nonzero only every M samples. These nonzero samples are separated by a sampling time interval $T' = MT$, resulting in the reduced sampling rate:

$$f_s' = \frac{1}{T'} = \frac{1}{MT} = \frac{1}{M}f_s$$

If M becomes too large, aliasing effects will become audible as the spectral images due to sampling overlap more and more. The aliasing will arise from the spectral components of $x(n)$ that lie outside the reduced Nyquist interval $[-f_s'/2, f_s'/2]$. A more mathematical discussion of downsampling effects may be found in Section 12.5 of the text [1].

In the above program, instead of actually multiplying by $s(n)$, we store one period of $s(n)$ into a circular buffer `s`, that is, we store the numbers $[1, 0, 0, \dots, 0]$, and periodically cycle over them with the help of the DAG pointer `i5` with `m5=1`.

Every M samples, we encounter a nonzero value in `s` and then we output that input sample to the codec. Otherwise, we output zero to the codec. The following three instructions implement this logic:

```
ar = dm(i5, m5);
ar = pass ar;
if ne jump output;
```

The first reads the current value of $s(n)$ from the circular buffer and saves it in `ar`. The second passes `ar` back into `ar` and its only effect is to force the updating of the flag bits of the ALU, and in particular, the flag `AZ`, which is `AZ=1` whenever `ar=0`, and `AZ=0` whenever `ar` is nonzero. The third instruction tests whether `ar` is nonzero and if so it outputs the current sample, otherwise it outputs zeros to the codec. To examine these operations in more detail, look at the simulator program `dnsamp1.dsp` in the examples subdirectory `sim`.

The following program, `dnsamp2.dsp`, is an alternative way to demonstrate aliasing. It generates internally a sinusoid of frequency f using a sinusoidal wavetable generator, and then it outputs every Mth sample of that sinusoid. The resulting sinusoid will be heard as having frequency $f_a = f \bmod f_s'$, where $f_s' = f_s/M$.

Wavetables are discussed in Section 8.1.3 of Ref. [1], and also in Section 4.14 of this manual and in the Appendix. The wavetable stores one period of length D samples of a sinusoid and it is stepped at increments of every c samples. The resulting sinusoid will have frequency $f = cf_s/D$.

```
{dnsamp2.dsp - aliasing of a sinusoid by decimation}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{--- define sampling rate in kHz: ---------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-----------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ---------------------------}

.const M = 10;                     {downsampling ratio; fs_low = fs/M}
.var/dm/circ s[M];                 {low-rate sampling pulse train}

.const D = 2000;                   {basic frequency increment fs/D = 4 Hz}
.var/dm/circ sine[D];              {sinusoidal wavetable, from sinetbl.hex}

.const FS = 8000;                  {fast sampling rate in samples/sec}
.const c = 250;                    {f = c * fs / D = c * 4 = 1000 Hz}
.const T = 4;                      {desired total duration in seconds}
.var/dm N;                         {N = fs * T = no. of samples in T sec}
```

```
ax1 = T * FS;                      {N = T * FS = duration in samples}
dm(N) = ax1;                       {save N in DM}

i5 = ^s; L5 = %s; m5 = 1;          {low-rate sampling pulse train}
i6 = ^sine; L6 = %sine; m6 = c;    {sine wavetable with increment c}

zero(i5, m5, L5);                  {initialize pulse train s(n) to zero}
ax0 = 0x7fff;                      {set s[0] = 1}
dm(s) = ax0;                       {s = [1,0,0,...,0] = one and M-1 zeros}

.init sine: <sinetbl.hex>;         {load sinusoidal wavetable}
                                   {from file sinetbl.hex}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- sample processing algorithm -----------------------------------------}

   ay0 = dm(N);
   ar = ay0 - 1;                   {decrement N -> N-1  until zero}
   dm(N) = ar;                     {save new N}
   if eq jump stop;                {do not stop until N iterations}

   mr1 = dm(i6, m6);               {mr1 = sinusoidal input sin(2*pi*f*t)}

   ar = dm(i5, m5);                {get sampling pulse}
   ar = pass ar;                   {forces updating of AZ flag}
   if ne jump output;              {if AR != 0, write output}

   mr1 = 0;                        {else, output is zero}

   output:

   dm(tx_buf + 1) = mr1;                         {left output sample}
   dm(tx_buf + 2) = mr1;                         {right output sample}

   rti;

stop:

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

Lab Procedure

a. The constant A represents the unit amplitude of the re-sampling pulse train $s(n)$. Set $A = 0$ and $M = 1$, and recompile and run. You should hear nothing because $s(n)$ becomes identically zero, so that `ar` remains zero causing only zeros to be sent to the codec. For the next part, reset A to unity.

b. With an initial choice of $f_s = 48$ kHz, choose successive downsampling ratios of $M = 1, 2, 3, 4, 6, 8, 12, 16, 24, 48, 96$, corresponding to sample rates of $f_s' = 48, 24, 16, 12, 8, 6, 4, 3, 2, 1, 0.5$ kHz. In each case, recompile and run the program and listen to the output. This experiment works better if you run it on a music CD than on your speech.

c. For the program `dnsamp2.dsp`, the sampling rate is $f_s = 8$ kHz and the frequency of the generated sinusoid $f = 1$ kHz. If the sampling rate is decimated down by a factor of $M = 10$ to the new sampling frequency of $f_s' = 800$ Hz, then the frequency $f = 1$ kHz will be aliased with $f_a = 1000 - 800 = 200$ Hz, which lies in the new Nyquist interval $[-400, 400]$ Hz.

To hear this effect, first you must generate the 2000-long period of the wavetable. The program reads the sinusoidal table values in 1.15 hex format from the file `sinetbl.hex`. These values can be generated by the following DOS command, which pipes the output of `sinetbl.exe` into `dec2hex.exe`:

```
sinetbl 0 1 2000 | dec2hex 1.15 > sinetbl.hex
```

The program plays the output for 4 seconds. No microphone input is required. The codec keeps sampling the input and, therefore, interrupts the processor at the sampling rate f_s. However, the samples that are loaded automatically into the receive register `rx_buf` are not used. Instead, the interrupt service routine reads a wavetable sample and sends it to the output every Mth time, while at other times, it sends zero to the output.

In this experiment, first choose $M = 1$, compile, and run the program to hear the 1 kHz sinusoid. Then, set $M = 10$, recompile and run to hear the 200 Hz aliased version.

4.2. Delays

The program `delay.dsp` implements a maximum delay of duration $T_D = 0.75$ sec. At an 8 kHz sampling rate, the total number of samples in the delay will be:

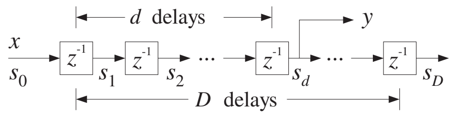
$$D = \frac{T_D}{T} = f_s T_D = 8000\text{ Hz} \times 0.75\text{ sec} = 6000$$

Thus, the transfer function of the delay will be $H(z) = z^{-D} = z^{-6000}$. Its implementation requires a 6001-dimensional circular buffer:

```
{delay.dsp - plain delay by TD sec}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on cdelay.c of Introduction to Signal Processing, p.177.
 I/O equation: y(n) = x(n-d), range of delay d=1, 2, ..., D.
 Sample processing algorithm:
       for each input x do:
           y = tap(D, w, p, d)            get d-th tap
           *p = x                         put input x into tap-0
           cdelay(D, w, &p)               update delay
}

{--- define sampling rate in kHz: ---------------------------------------}
                   {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
```

```
                     {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850;  {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                     {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                     {0xc85e = 9.6    | 0xc85f = 6.615                     }
{---------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const D = 6000;            {max delay = TD = D/fs = 6000/8000 = 3/4 sec}
.const d = 2000;            {d-th tap, d = 1,2,...,D}
.var/dm/circ w[D+1];        {define delay-line buffer}

i2 = ^w;  L2 = %w;          {buffer pointer and length}

zero(i2, m2, L2);           {clear delay line}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- read input samples from codec ---------------------------------------}

   ax1 = dm(rx_buf + 1);           {left input sample}
   mx1 = dm(rx_buf + 2);           {right input sample}

{--- sample processing algorithm --- process right channel only ----------}

   tap(i2, m2, d, my1);            {d-th tap output into my1}
   tapin(i2, m2, mx1);             {input from mx1 into tap-0}
   cdelay(i2, m2);                 {update delay}

{--- write output samples to codec ---------------------------------------}

   dm(tx_buf + 1) = my1;           {left output sample}
   dm(tx_buf + 2) = my1;           {right output sample}

{--- return from interrupt -----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

The initial version of the program outputs the 2000th tap, that is, the signal $y(n) = x(n-d)$, corresponding to a time delay in seconds: $T_d = dT = d/f_s = 2000/8000 = 0.25$ sec. The sample processing algorithm in the notation of the text [1] is:



```
for each input x do:
    $y = \text{tap}(D, w, p, d)$
    $*p = x$
    $\text{cdelay}(D, w, \&p)$
```

Lab Procedure

a. Go to the directory `c:\adi_dsp\examples\delay` and compile, link, and load this program by the following DOS commands:

```
dsp
cd delay
ezk delay
ezl delay
```

Give the system an impulse by lightly tapping the table with the mike, and listen to the impulse response. Then, speak into the mike.

Bring the mike near the speaker and then give the system an impulse. You should hear repeated echoes. If you bring the mike too close to the speakers the output goes unstable. Draw a block diagram realization that would explain the effect you are hearing. Experimentally determine the distance at which the echoes remain marginally stable, that is, neither die out nor diverge. (Technically speaking, the poles of your closed-loop system lie on the unit circle.)

b. Change the sampling rate to 16 kHz, recompile and reload keeping the value of d the same, that is, $d = 2000$. Listen to the impulse response. What is the duration of the delay in seconds now?

c. Reset the sampling rate back to 8 kHz, and this time change d to its maximum value $d = D = 6000$. Recompile, reload, and listen to the impulse response. Experiment with lower and lower values of d and listen to your delayed voice until you can no longer distinguish a separate echo. How many milliseconds of delay does this correspond to?

d. Set $d = 0$, recompile and reload. It should correspond to no delay at all, that is, $y(n) = x(n)$. But what do you hear? Can you explain why? Can you fix it by changing the program? Will your modified program still work with $d \neq 0$? Is there any good reason for structuring the program the way it was originally?

4.3. FIR Filters

General FIR Filter Program

The implementation of an FIR filter is accomplished with the help of the macro `cfir`, which is the assembly code equivalent of the text routines `cfir.c` and `cfir2.c`. The following is a general FIR filtering program:

```
{fir.dsp - FIR filter experiment}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on cfir.c or cfir2.c of Introduction to Signal Processing}

{--- define sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
```

```
                     {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850;  {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                     {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                     {0xc85e = 9.6    | 0xc85f = 6.615                     }
{----------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ------------------------------}

.const M = 206;                    {from firlp.m with fc=200 Hz, Df=100 Hz}
.var/dm/circ w[M+1];               {circular delay-line buffer in DM}
.var/pm/circ h[M+1];               {filter coefficient buffer in PM}

i2 = ^w; L2 = %w;                  {delay-line buffer pointer and length}
i4 = ^h; L4 = %h;                  {coefficient buffer pointer and length}

zero(i2, m2, L2);                  {clear delay line}

.init h: <fir.hex>;                {read coefficients in 1.15 hex format}
                                   {from file fir.hex}
{--- start processing input samples ----------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- read input samples from codec -----------------------------------------}

   ax1 = dm(rx_buf + 1);                  {left input sample}
   mx1 = dm(rx_buf + 2);                  {right input sample}

{--- sample processing algorithm --- process right input only ------------}

   cfir(M, i4, m4, i2, m2, mx1);          {input from mx1, output in mr1}

{--- write output samples to codec -----------------------------------------}

   dm(tx_buf + 1) = mr1;                  {left output sample}
   dm(tx_buf + 2) = mr1;                  {right output sample}

{--- return from interrupt ------------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

The filter coefficients are read from the hex file `fir.hex`. The value of the filter order M must be edited into the program. The delay-line buffer is in DM memory and the filter coefficient buffer in PM.

The following MATLAB file `firlp.m` implements a Kaiser design of a lowpass filter with prescribed cutoff frequency and transition width. The passband and stopband attenuations are assumed to be 0.1 dB and 40 dB, respectively. It uses the functions `dtft.m` and `klh.m` of Chapter 10 of the text [1]:

```
% firlp.m - lowpass FIR design
```

```
% Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996
%
% the filter coeff file, lp.dec, in decimal format must be postprocessed
% by dec2hex to convert it to the 1.15 format hex file lp.hex, that is,
% dec2hex 1 15 < lp.dec > lp.hex
%
% to run it on the EZ-KIT Lite,
% copy lp.hex into fir.hex, recompile and load fir.dsp
% (you may also need to edit the designed filter order M into fir.dsp)

fs = 8000;                          % 8 kHz rate

Apass = 0.1;                        % 0.1 dB
Astop = 40;                         % 40 dB, stopband ripple: delta = 0.01

fc = input('cutoff frequency fc (in Hz) =  ');      % e.g., fc = 200 Hz
Df = input('transition width Df (in Hz) =  ');      % e.g., Df = 100 Hz

fpass = fc - Df/2;
fstop = fc + Df/2;

h = klh(1, fs, fpass, fstop, Apass, Astop)';              % Kaiser design

save lp.dec h /ascii                          % save in decimal format

[N, N1] = size(h); M = N-1                    % filter order M

NF = 400;                                     % number of frequencies
w = (0:NF-1) * pi / NF;                       % NF freqs over positive Nyquist
H = 20 * log10(abs(dtft(h', w)))';            % magnitude response in dB

save lpmag.dat H /ascii
```

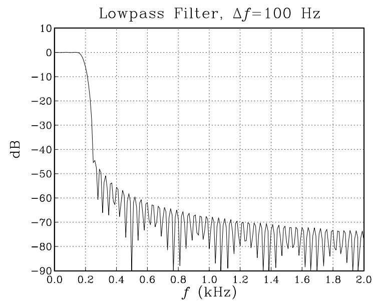
Using this program, we have designed two FIR filters with a sampling rate of 8 kHz and cutoff frequencies and widths:

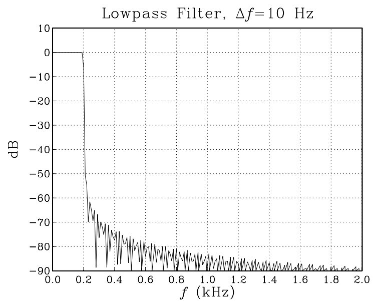
f_c	Δf
200 Hz	100 Hz
200 Hz	10 Hz

These specifications imply that the passband, transition, and stopband frequency ranges are in Hz (where the stopband extends to the Nyquist frequency $f_s/2 = 4000$ Hz):

passband	transition	stopband
0–150	150–250	250–4000
0–195	195–205	205–4000

Thus, within the passband, the attenuation must remain less than 0.1 dB and within the stopband, it must be greater than 40 dB. The magnitude responses $|H(f)|$ of the two filters are plotted in units of dB, that is, $20\log_{10}|H(f)|$, in the following figures:





The designed filters have orders $M = 206$ and $M = 2054$, respectively, and their impulse responses are in the data files `lp1.dec` and `lp2.dec` in decimal format (as generated by `firlp.m`.)

As discussed in Ch.10 of the text [1] that the filter order M of an FIR filter is essentially inversely proportional to its transition width Δf, and therefore, it should be no surprise that the second filter has 10-times greater order than the first.

FIR Filtering in Stereo

If we want to process the input samples in stereo, we must apply the filter to the two channels separately. The following program `fir2.dsp` independently processes the left and right stereo channels. It uses two circular delay-line buffers in DM and a common filter coefficient buffer in PM:

```
{fir2.dsp - FIR filter experiment - with separate stereo channels}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on cfir.c or cfir2.c of Introduction to Signal Processing}

{--- define sampling rate in kHz: ---------------------------------------}
                   {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                   {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                   {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                   {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-----------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ---------------------------}

.const M = 206;                 {designed by firlp.m or firhp.m}
.var/dm/circ wL[M+1];           {left delay-line buffer in DM}
.var/dm/circ wR[M+1];           {right delay-line buffer in DM}
.var/pm/circ h[M+1];            {filter coefficient buffer in PM}

i2 = ^wL; L2 = %wL;             {delay-line buffer pointer and length}
i3 = ^wR; L3 = %wR;             {delay-line buffer pointer and length}
i4 = ^h;  L4 = %h;              {coefficient buffer pointer and length}

zero(i2, m2, L2);               {clear left delay line}
```

```
zero(i3, m3, L3);                 {clear right delay line}

.init h: <fir.hex>;               {read coefficients in 1.15 hex format}
                                  {from file fir.hex}
{--- start processing input samples -------------------------------------}

wait: idle; jump wait;            {wait for interrupt and loop forever}
                                  {interrupt service routine starts here}
input_samples: ena sec_reg;       {enable secondary register set}

{--- read input samples from codec --------------------------------------}

   ax1 = dm(rx_buf + 1);                 {left input sample}
   mx1 = dm(rx_buf + 2);                 {right input sample}

{--- sample processing algorithm --- process right input only ------------}

   cfir(M, i4, m4, i2, m2, ax1);         {input from ax1, output in mr1}
   ay1 = mr1;                            {left output}
   cfir(M, i4, m4, i3, m3, mx1);         {input from mx1, output in mr1}
   my1 = mr1;                            {right output}

{--- write output samples to codec --------------------------------------}

   dm(tx_buf + 1) = ay1;                 {left output sample}
   dm(tx_buf + 2) = my1;                 {right output sample}

{--- return from interrupt ----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;    {wrapup}
```

FIR Filtering of Sinusoids

Finally, we consider a third version of the FIR filtering program, `fir3.dsp`, that uses the concept of wavetable generators from Section 8.1.3 of the text [1].

```
{fir3.dsp - FIR filtering of sinusoids generated by circular wavetable}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based of cfir.c and Sec. 8.1.3 of Introduction to Signal Processing}

{--- define sampling rate in kHz: ---------------------------------------}
                   {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                   {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                   {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                   {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-----------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ---------------------------}

.const FS = 8000;                 {sampling rate in samples/sec}
```

```
.const M = 206;                   {from firlp.m with fc=200 Hz, Df=100 Hz}
.var/dm/circ w[M+1];              {circular delay-line buffer in DM}
.var/pm/circ h[M+1];              {filter coefficient buffer in PM}

.const D = 800;                   {basic frequency increment fs/D = 10 Hz}
.var/dm/circ s[D];                {sinusoidal wavetable, from sinetbl.hex}
                                  {see Section 8.1.3 of text}
.const c1 = 10;                   {f1 = c1 * fs / D = c1 * 10 = 100 Hz}
.const c2 = 40;                   {f2 = c2 * fs / D = c2 * 10 = 400 Hz}
.const A1 = 0x4000;               {A1 = 0.5 = amplitude of sinusoid 1}
.const A2 = 0x4000;               {A2 = 0.5 = amplitude of sinusoid 2}

.const T = 4;                     {desired total duration in seconds}
.var/dm N;                        {N = fs * T = no. of samples in T sec}

ax1 = T * FS;                     {N = T * FS = duration in samples}
dm(N) = ax1;                      {save N in DM}

i2 = ^w; L2 = %w;                 {delay-line buffer pointer and length}
i4 = ^h; L4 = %h;                 {coefficient buffer pointer and length}

i5 = ^s; L5 = %s; m5 = c1;        {sinusoid f1, wavetable increment c1}
i6 = ^s; L6 = %s; m6 = c2;        {sinusoid f2, wavetable increment c2}

zero(i2, m2, L2);                 {clear delay line}

.init h: <fir.hex>;               {read coefficients in 1.15 hex format}
                                  {from file fir.hex}
.init s: <sinetbl.hex>;           {load sinusoidal wavetable}
                                  {from file sinetbl.hex}

{--- start processing input samples -------------------------------------}

wait: idle; jump wait;            {wait for interrupt and loop forever}
                                  {interrupt service routine starts here}
input_samples: ena sec_reg;       {enable secondary register set}

{--- sample processing algorithm ----------------------------------------}

   ay0 = dm(N);
   ar = ay0 - 1;                         {decrement N until zero}
   dm(N) = ar;
   if eq jump stop;                      {do not stop until N iterations}

   mx1 = A1;
   my1 = dm(i5, m5);                     {my1 = s1 = sin(2*pi*f1*t)}
   mr = mx1 * my1 (ss);                  {mr = A1 * s1}

   mx1 = A2;
   my1 = dm(i6, m6);                     {my1 = s2 = sin(2*pi*f2*t)}
   mr = mr + mx1 * my1 (rnd);            {mr = x = A1 * s1 + A2 * s2}

#if 1                                    {use 0 to skip filtering}
   cfir(M, i4, m4, i2, m2, mr1);         {input from mr1, output in mr1}
#endif

{--- write samnples to codec --------------------------------------------}
```

```
      dm(tx_buf + 1) = mr1;               {left output sample}
      dm(tx_buf + 2) = mr1;               {right output sample}

      rti;

   stop:

   .include <c:\adi_dsp\macros\end.dsp>;  {wrapup}
```

This program generates internally a sum of two sinusoids and filters them through the `cfir` macro. The input signal is:

$$x(t) = A_1 \sin(2\pi f_1 t) + A_2 \sin(2\pi f_2 t)$$

where the two sinusoidal terms are generated from a common wavetable which is cycled every c_1 and c_2 samples, such that

$$f_1 = c_1 \frac{f_s}{D}, \qquad f_2 = c_2 \frac{f_s}{D}$$

The wavetable is filled with one period of length D, that is, by the numbers:

$$s(n) = A \sin\Big(\frac{2\pi n}{D}\Big), \qquad n = 0, 1, \ldots, D-1$$

We use $D = 800$ so that the smallest frequency that can be generated is $f_s/D = 8000/800 = 10$ Hz, at an 8 kHz sampling rate. We took $A = 1$ for the amplitude and generated the D sinusoidal values by the DOS wavetable generator `sinetbl.exe` and converted them to hex by the DOS commands:

```
sinetbl 0 1 800 > sinetbl.dec
dec2hex 1.15 < sinetbl.dec > sinetbl.hex
```

The wavetable is loaded on the chip at run time. The wavetable is cycled over at different speeds by two independent DAG2 pointers, `i5` and `i6`, which are incremented respectively by `m5=c1` and `m6=c2`. The wavetable increments are $c_1 = 10$ and $c_2 = 40$, resulting in the frequencies $f_1 = 100$ and $f_2 = 400$ Hz. One is in the passband and the other in the stopband of the filter. Thus, the filter will remove f_2 and let f_1 pass through. The amplitudes were chosen to be $A_1 = A_2 = 0.5$. In Section 4.14, we discuss the definition and use of the macro `wavgen.dsp` which can also be used to generate the required sinusoids.

Lab Procedure

a. Go into the directory `c:\adi_dsp\examples\fir`. Then, convert the filter coefficient file `lp1.dec` into the hex file `fir.hex`. Then, compile and load the `fir.dsp` program. These operations are carried out by the commands:

```
dsp
cd fir
dec2hex 1.15 < lp1.dec > fir.hex
ezk fir
ezl fir
```

Speak into the mike or sing a do-re-mi scale. Note how the filter cuts off the higher pitches of your voice.

b. Next, test the longer filter. First edit the file `fir.dsp` so that $M = 2054$, then convert the coefficient file `lp2.dec` into `fir.hex`, recompile and reload.

Speak into the mike. The filter will cut off the higher pitches in your voice, but it also introduces a perceptible delay in the output. This delay was too short to be heard for the first filter.

Through their phase response, all filters introduce a certain amount of delay, which depends on the frequency of the input. FIR filters can be designed to have linear phase, which implies that all frequency components of the input get delayed by the same amount, and thus, the input as a whole gets delayed. For a linear phase FIR filter of order M, the amount of delay is:

$$D = \frac{M}{2} \qquad \Rightarrow \qquad T_D = \frac{M}{2}T = \frac{M}{2f_s}$$

Thus, how much is the delay in seconds for filter-1 and filter-2?

c. Next, run program `fir3.dsp` with the order-206 filter. First, listen to the unfiltered sinusoid f_1. This can be done by commenting out the `cfir` call and replacing the sinusoid amplitudes by $A_1 = 1$ (`0x7fff` in hex) and $A_2 = 0$.

Then, listen to the unfiltered component f_2, and then, to the unfiltered sum of both. Finally, uncomment `cfir` and send in the sum of the two. You will hear only the f_1 component at the output.

4.4. Comb Filters

This experiment implements the FIR comb filter given by Eq. (8.2.8) of the text [1]:

$$y(n) = x(n) + ax(n-D) + a^2x(n-2D) + a^3x(n-3D)$$

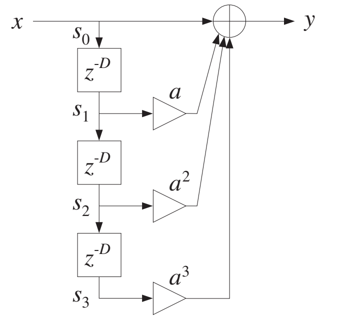
Its transfer function is

$$H(z) = 1 + az^{-D} + a^2z^{-2D} + a^3z^{-3D}$$

This filter can be implemented using the program `fir.dsp` of the previous experiment as a general FIR filter with an impulse response:

$$\mathbf{h} = [1, \underbrace{0, 0, \dots, 0}_{D-1 \text{ zeros}}, a, \underbrace{0, 0, \dots, 0}_{D-1 \text{ zeros}}, a^2, \underbrace{0, 0, \dots, 0}_{D-1 \text{ zeros}}, a^3]$$

Because of the sparseness of the impulse response, a more efficient implementation is to program the block diagram directly, in the sense of using a common delay line of order $3D$ and tapping it out at taps 0, D, $2D$, and $3D$. The block diagram realization and corresponding sample processing algorithm will be:



```
for each input x do:
    $s_0 = x$
    $s_1 = \text{tap}(3D, w, p, D)$
    $s_2 = \text{tap}(3D, w, p, 2D)$
    $s_3 = \text{tap}(3D, w, p, 3D)$
    $y = s_0 + as_1 + a^2s_2 + a^3s_3$
    $*p = s_0$
    $\text{cdelay}(3D, w, \&p)$
```

The following program `comb.dsp` is the translation to assembly language:

```
{comb.dsp - FIR comb filter}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Eq.(8.2.8) of Introduction to Signal Processing.
 I/O equation: y(n) = a0 x(n) + a1 x(n-D) + a2 x(n-2D) + a3 x(n-3D)
 Sample processing algorithm:
       for each x do:
           *p = s0 = x                                    read filter input
           s1 = tap(3*D, w, p, D)                         get tap-D
           s2 = tap(3*D, w, p, 2*D)                       get tap-2D
           s3 = tap(3*D, w, p, 3*D)                       get tap-3D
           y = a0 * s0 + a1 * s1 + a2 * s2 + a3 * s3      filter output
           cdelay(3*D, w, &p)                             update delay
}

{--- define sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const a0 = 0x7fff;               {a0 = 0.9999 = approximate unity}
.const a1 = 0x4000;               {a1 = 0.50}
.const a2 = 0x2000;               {a2 = a1^2 = 0.50^2 = 0.25}
.const a3 = 0x1000;               {a3 = a1^3 = 0.50^3 = 0.125}

.const D = 2000;                  {TD = D/fs = 2000/8000 = 1/4 sec}
.var/dm/circ w[3*D + 1];          {delay-line buffer, max delay = 3*D}

i2 = ^w;  L2 = %w;                {delay-line buffer pointer and length}

zero(i2, m2, L2);                 {clear delay line}

{--- start processing input samples -------------------------------------}
```

```
wait: idle; jump wait;              {wait for interrupt and loop forever}
                                    {interrupt service routine starts here}
input_samples: ena sec_reg;         {enable secondary register set}

{--- read input samples from codec --------------------------------------}

   ax1 = dm(rx_buf + 1);            {left input sample}
   mx1 = dm(rx_buf + 2);            {right input sample}

{--- sample processing algorithm --- process right channel only ---------}

   my1 = a0;
   mr = mx1 * my1 (ss);             {mr = a0 * x}

   tap(i2, m2, D, my0);             {my0 = s1 = D-th tap}
   mx0 = a1;
   mr = mr + mx0 * my0 (ss);        {mr = a0 * x + a1 * s1}

   tap(i2, m2, 2*D, my0);           {my0 = s2 = 2D-th tap}
   mx0 = a2;
   mr = mr + mx0 * my0 (ss);        {mr = a0 * x + a1 * s1 + a2 * s2}

   tap(i2, m2, 3*D, my0);           {my0 = s3 = 3D-th tap}
   mx0 = a3;
   mr = mr + mx0 * my0 (rnd);       {a0 * x + a1 * s1 + a2 * s2 + a3 * s3}
   if mv sat mr;                    {saturate mr if overflow}

   tapin(i2, m2, mx1);              {put input from mx1 into tap-0}
   cdelay(i2, m2);                  {update delay}

{--- write output samples to codec --------------------------------------}

   dm(tx_buf + 1) = mr1;            {left output sample}
   dm(tx_buf + 2) = mr1;            {right output sample}

{--- return from interrupt ----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

The delay D is taken to be $D = 2000$, corresponding to $T_D = 0.25$ sec, so that the total duration of the filter is $3T_D = 0.75$ sec. The parameter a is chosen to be $a = 0.5$.

Lab Procedure

a. Go to directory `c:adi_dsp\examples\comb`, compile, link, and load the program using the commands:

```
dsp
cd comb
ezk comb
ezl comb
```

Listen to the impulse response of the filter. Speak into the mike. Bring the mike close to the speakers and get a closed-loop feedback.

b. Keeping the delay D the same, choose $a = 0.2$ and run the program again. What effect do you hear? Repeat for $a = 0.1$.

c. Finally, run the program with the values $a = 1$ and $a = -1$. Note that 1 is represented approximately by `0x7fff` in 1.15 format, whereas -1 is represented exactly by `0x8000`.

d. The FIR comb filter can also be implemented as an *ordinary* FIR filter, without taking into account the sparseness of its impulse response $\mathbf{h}$. In this part, define the $(3D + 1)$-dimensional impulse response:

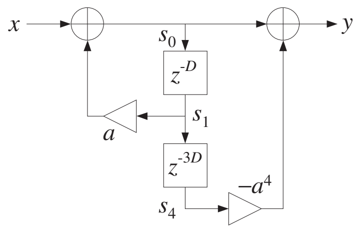
$$\mathbf{h} = [1, \underbrace{0, 0, \dots, 0}_{D-1\text{ zeros}}, a, \underbrace{0, 0, \dots, 0}_{D-1\text{ zeros}}, a^2, \underbrace{0, 0, \dots, 0}_{D-1\text{ zeros}}, a^3]$$

and assign it to a circular buffer in PM. Then, use the program `fir.dsp` of the previous section to implement this filter. Compile and run with the value $D = 2000$ so that you may compare its output with that of `comb.dsp`.

e. The FIR comb can also be implemented *recursively* using the geometric series formula to rewrite its transfer function in the recursive form:

$$H(z) = 1 + az^{-D} + a^2z^{-2D} + a^3z^{-3D} = \frac{1 - a^4z^{-4D}}{1 - az^{-D}}$$

which requires a $(4D+1)$-dimensional delay-line buffer. The canonical realization and the corresponding sample processing algorithm are shown below:



```
for each input x do:
    $s_1 = \text{tap}(4D, w, p, D)$
    $s_4 = \text{tap}(4D, w, p, 4D)$
    $s_0 = x + as_1$
    $y = s_0 - a^4 * s_4$
    $*p = s_0$
    $\text{cdelay}(4D, w, \&p)$
```

The following program `comb2.dsp` is the assembly code implementation. Using the values $D = 1600$ (corresponding to a 0.2 sec delay) and $a = 0.5$, recompile and run both the `comb.dsp` and `comb2.dsp` programs and listen to their outputs. In general, such recursive implementations of FIR filters are more prone to the accumulation of roundoff errors than the non-recursive versions. You may want to run these programs with $a = 1$ and $a = -1$ to observe this sensitivity.

```
{comb2.dsp - FIR comb filter implemented recursively}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Eq.(8.2.8) of Introduction to Signal Processing.
 I/O equation: y(n) = a0 x(n) + a1 x(n-D) + a2 x(n-2D) + a3 x(n-3D)
 Sample processing algorithm:
       for each x do:
           s1 = tap(4*D, w, p, D)                       get tap-D
           s4 = tap(4*D, w, p, 4*D)                     get tap-4D
           *p = s0 = x + a * s1
           y =  s0 - a4 * s4                            filter output
           cdelay(4*D, w, &p)                           update delay
}

{--- define sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const a0 = 0x7fff;                {a0 = 0.9999 = approximate unity}
.const a1 = 0x4000;                {a1 = 0.50}
.const a4 = 0x0800;                {a4 = a1^4 = 0.50^4 = 0.0625}

.const D = 1600;                   {TD = D/fs = 1600/8000 = 0.20 sec}
.var/dm/circ w[4*D + 1];           {delay-line buffer, max delay = 4*D}

i2 = ^w;  L2 = %w;                 {delay-line buffer pointer and length}

zero(i2, m2, L2);                  {clear delay line}

{--- start processing input samples -------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- read input samples from codec --------------------------------------}

   ax1 = dm(rx_buf + 1);           {left input sample}
   mx1 = dm(rx_buf + 2);           {right input sample}

{--- sample processing algorithm --- process right channel only ----------}

   mr = 0;
   mr1 = mx1;                      {mr = x = input}

   tap(i2, m2, D, my0);            {my0 = s1 = D-th tap}
   mx0 = a1;
   mr = mr + mx0 * my0 (rnd);      {mr = s0 =  x + a1 * s1}
   if mv sat mr;                   {saturate mr if overflow}
   sr0 = mr1;                      {sr0 = s0}
```

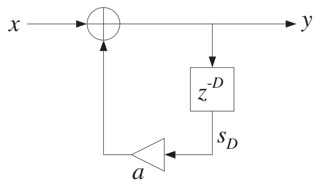
```
   mx0 = a4;
   tap(i2, m2, 4*D, my0);          {my0 = s4 = 4D-th tap}
   mr = mr - mx0 * my0 (rnd);      {mr = y = s0 - a4 * s4}
   if mv sat mr;                   {saturate mr if overflow}

   tapin(i2, m2, sr0);             {put input from sr0 into tap-0}
   cdelay(i2, m2);                 {update delay}

{--- write output samples to codec ---------------------------------------}

   dm(tx_buf + 1) = mr1;           {left output sample}
   dm(tx_buf + 2) = mr1;           {right output sample}

{--- return from interrupt -----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

4.5. Plain Reverb

A plain reverberator can be used as an elementary building block for more complicated reverberation algorithms. It is given by Eq. (8.2.12) of the text [1] and shown in Fig. 8.2.6. Its I/O equation and transfer function are:

$$y(n) = ay(n-D) + x(n), \qquad H(z) = \frac{1}{1 - az^{-D}}$$

Its sample processing algorithm using a circular delay-line buffer is given by Eq. (8.2.14) of Ref. [1]:



```
for each input sample $x$ do:
    $s_D = \text{tap}(D, w, p, D)$
    $y = x + as_D$
    $*p = y$
    $\text{cdelay}(D, w, \&p)$
```

The following program `plain.dsp` is the translation to assembly code:

```
{plain.dsp - plain reverb - IIR comb filter}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on plain.c of Introduction to Signal Processing.
 Sample processing algorithm from Eq.(8.2.14) and Fig.8.2.6:
       for each x do :
           sD = tap(D, w, p, D)                  get D-th tap
           y = x + a * sD                        filter output
           *p = y                                put y into tap-0
           cdelay(D, w, &p)                      update delay
}

{--- define sampling rate in kHz: ----------------------------------------}
```

```
                     {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                     {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850;  {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                     {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                     {0xc85e = 9.6    | 0xc85f = 6.615                     }
{--------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const a = 0x4000;               {feedback coefficient a = 0.50}
.const D = 3000;                 {TD = D/fs = 3/8 = 0.375 sec}
.var/dm/circ w[D+1];             {delay-line buffer}

i2 = ^w;  L2 = %w;               {delay-line buffer pointer and length}

zero(i2, m2, L2);                {clear delay line}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;           {wait for interrupt and loop forever}
                                 {interrupt service routine starts here}
input_samples: ena sec_reg;      {enable secondary register set}

{--- read input samples from codec ---------------------------------------}

   ax1 = dm(rx_buf + 1);         {left input sample}
   mx1 = dm(rx_buf + 2);         {right input sample}

{--- sample processing algorithm --- process right channel only ----------}

   mr = 0;
   mr1 = mx1;                    {mr = x = input}

   my1 = a;
   tap(i2, m2, D, mx1);          {mx1 = sD = D-th tap}
   mr = mr + mx1 * my1 (rnd);    {mr = x + a * sD}

   tapin(i2, m2, mr1);           {put y in tap-0}
   cdelay(i2, m2);               {update delay}

{--- write output samples to codec ---------------------------------------}

   dm(tx_buf + 1) = mr1;         {left output sample}
   dm(tx_buf + 2) = mr1;         {right output sample}

{--- return from interrupt -----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

The program `plain2.dsp` is the implementation in stereo:

```
{plain2.dsp - plain reverb - IIR comb filter - in stereo}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}
```

```
{Based on plain.c of Introduction to Signal Processing.
 Sample processing algorithm from Eq.(8.2.14) and Fig.8.2.6:
      for each x do :
          sD = tap(D, w, p, D)                  get D-th tap
          y = x + a * sD                        filter output
          *p = y                                put y into tap-0
          cdelay(D, w, &p)                      update delay
}

{--- define sampling rate in kHz: ----------------------------------------}
                     {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                     {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850;  {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                     {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                     {0xc85e = 9.6    | 0xc85f = 6.615                     }
{--------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const aL = 0x4000;              {left feedback coefficient aL = 0.50}
.const DL = 3000;                {left TD = D/fs = 3/8 = 0.375 sec}

.const aR = 0x4000;              {right feedback coefficient aR = 0.50}
.const DR = 3000;                {right TD = D/fs = 3/8 = 0.375 sec}

.var/dm/circ wL[DL+1];           {left delay-line buffer}
.var/dm/circ wR[DR+1];           {right delay-line buffer}

i2 = ^wL; L2 = %wL;              {left delay buffer pointer and length}
i3 = ^wR; L3 = %wR;              {right delay buffer pointer and length}

zero(i2, m2, L2);                {clear left delay line}
zero(i3, m3, L3);                {clear right delay line}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;           {wait for interrupt and loop forever}
                                 {interrupt service routine starts here}
input_samples: ena sec_reg;      {enable secondary register set}

{--- read input samples from codec ---------------------------------------}

   ax1 = dm(rx_buf + 1);         {left input sample}
   mx1 = dm(rx_buf + 2);         {right input sample}

{--- sample processing algorithm for left channel ------------------------}

   mr = 0;
   mr1 = ax1;                    {mr = x = left input}

   my0 = aL;
   tap(i2, m2, DL, mx0);         {mx0 = sD = D-th tap}
   mr = mr + mx0 * my0 (rnd);    {mr = y = x + a * sD = left output}

   tapin(i2, m2, mr1);           {put y in tap-0}
   cdelay(i2, m2);               {update left delay}
```

```
      dm(tx_buf + 1) = mr1;           {left output sample}

   {--- sample processing algorithm for right channel -----------------------}

      mr = 0;
      mr1 = mx1;                      {mr = x = right input}

      my0 = aR;
      tap(i3, m3, DR, mx0);           {mx0 = sD = D-th tap}
      mr = mr + mx0 * my0 (rnd);      {mr = y = x + a * sD = right output}

      tapin(i3, m3, mr1);             {put y in tap-0}
      cdelay(i3, m3);                 {update right delay}

      dm(tx_buf + 2) = mr1;           {right output sample}

   {--- return from interrupt ---------------------------------------------}

      rti;

   .include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

The plain reverberator is an IIR comb filter with frequency response shown in Fig. 8.2.7. When the filter parameter a is positive and near unity, the peak gains $1/(1-a)$ become large, causing overflows. In such cases, the input must be appropriately scaled down using the shifter before it is passed to the filter. This was not done in the above programs, but the reader should be aware that it may need to be done.

Lab Procedure

a. Go to directory `c:\adi_dsp\examples\plain`, compile, link, and run the program `plain.dsp` with the parameter values $D = 3000$ and $a = 0.5$. Listen to the impulse response of the system. Speak into the mike.

   (If you have a portable CD with you, change the jumpers on the EZ-KIT Lite board, and play a CD through it. To get better sound quality, you may want to use the stereo version `plain2.dsp`. In this case, you may also want to experiment with using different values of the left and right delays.)

b. Recompile and run the program with the new feedback coefficient $a = 0.25$. Listen to the impulse response. Repeat for $a = 0.75$. Discuss the effect of increasing or decreasing a.

c. According to Eq. (8.2.16), the effective reverberation time constant is given by

$$\tau_{\rm eff} = \frac{\ln\epsilon}{\ln a}T_D, \qquad T_D = DT = D/f_s$$

   For each of the above values of a, calculate $\tau_{\rm eff}$ in seconds, assuming $\epsilon = 0.001$ (which corresponds to the so-called 60-dB time constant.) Is what you hear consistent with this expression?

d. According to this formula, $\tau_{\rm eff}$ remains invariant under the replacements:

$$D \to 2D, \qquad a \to a^2$$

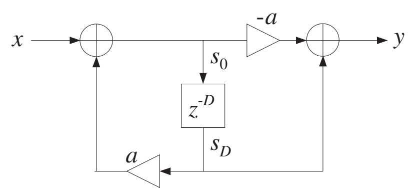
   Test if this is true by running your program and hearing the output with $D = 6000$ and $a = 0.5^2 = 0.25$ and comparing it with the case $D = 3000$ and $a = 0.5$. Repeat the comparison also with $D = 1500$ and $a = \sqrt{0.5} = 0.7071$.

4.6. Allpass Reverb

Like the plain reverberator, an allpass reverberator can be used as an elementary building block for building more complicated reverberation algorithms. It is given by Eq. (8.2.25) of the text [1] and shown in Fig. 8.2.17. Its I/O equation and transfer function are:

$$y(n) = ay(n-D) - ax(n) + x(n-D), \qquad H(z) = \frac{-a + z^{-D}}{1 - az^{-D}}$$

Its sample processing algorithm using a circular delay-line buffer is given by Eq. (8.2.14) of Ref. [1]:



```
for each input sample x do:
    $s_D = \text{tap}(D, w, p, D)$
    $s_0 = x + as_D$
    $y = -as_0 + s_D$
    $*p = s_0$
    $\text{cdelay}(D, w, \&p)$
```

The program `allpass.dsp` is the translation to assembly code:

```
{allpass.dsp - allpass reverb - canonical realization}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on allpass.c of Introduction to Signal Processing.
 Sample processing algorithm from Eq.(8.2.30) and Fig. 8.2.17:
       for each x do:
           sD = tap(D, w, p, D)          get D-th tap
           s0 = x + a * sD               input to delay line
           y = -a * s0 + sD              filter output
           *p = s0                       put input to delay line
           celay(D, w, &p)               update delay line
}

{--- define sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{------------------------------------------------------------------------}
```

```
.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const a = 0x4000;                {feedback coefficient a = 0.50}
.const D = 3000;                  {TD = D/fs = 3/8 = 0.375 sec}
.var/dm/circ w[D+1];              {delay-line buffer}

i2 = ^w;  L2 = %w;                {delay-line buffer pointer and length}

zero(i2, m2, L2);                 {clear delay line}

{--- start processing input samples -------------------------------------}

wait: idle; jump wait;            {wait for interrupt and loop forever}
                                  {interrupt service routine starts here}
input_samples: ena sec_reg;       {enable secondary register set}

{--- read input samples from codec --------------------------------------}

   ax1 = dm(rx_buf + 1);          {left input sample}
   mx1 = dm(rx_buf + 2);          {right input sample}

{--- sample processing algorithm --- process right channel only ---------}

   mr = 0;
   mr1 = mx1;                     {mr = x = input}

   my0 = a;
   tap(i2, m2, D, mx0);           {mx0 = sD = D-th tap}
   mr = mr + mx0 * my0 (rnd);     {mr = s0 = x + a * sD}

   tapin(i2, m2, mr1);            {put s0 in tap-0}
   cdelay(i2, m2);                {update delay}

   mr =  mr1 * my0 (rnd);         {mr = a * s0}
   ay0 = mx0;                     {ay0 = sD}
   ar =  ay0 - mr1;               {ar = y = sD - a * s0}

{--- write output samples to codec --------------------------------------}

   dm(tx_buf + 1) = ar;           {left output sample}
   dm(tx_buf + 2) = ar;           {right output sample}

{--- return from interrupt ----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

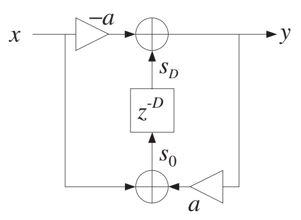
The program `allpass2.dsp` is the implementation in stereo:

```
{allpass2.dsp - allpass reverb in stereo}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on allpass.c of Introduction to Signal Processing.
 Sample processing algorithm from Eq.(8.2.30) and Fig.8.2.17:
       for each x do:
```

```
            sD = tap(D, w, p, D)           get D-th tap
            s0 = x + a * sD                input to delay line
            y = -a * s0 + sD               filter output
            *p = s0                        put input to delay line
            celay(D, w, &p)                update delay line
}

{--- define sampling rate in kHz: ---------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-----------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ---------------------------}

.const aL = 0x4000;                {left feedback coefficient aL = 0.50}
.const DL = 3000;                  {left TD = D/fs = 3/8 = 0.375 sec}

.const aR = 0x4000;                {right feedback coefficient aR = 0.50}
.const DR = 3000;                  {right TD = D/fs = 3/8 = 0.375 sec}

.var/dm/circ wL[DL+1];             {left delay-line buffer}
.var/dm/circ wR[DR+1];             {right delay-line buffer}

i2 = ^wL; L2 = %wL;                {left delay buffer pointer and length}
i3 = ^wR; L3 = %wR;                {right delay buffer pointer and length}

zero(i2, m2, L2);                  {clear left delay line}
zero(i3, m3, L3);                  {clear right delay line}

{--- start processing input samples ------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- read input samples from codec -------------------------------------}

   ax1 = dm(rx_buf + 1);           {left input sample}
   mx1 = dm(rx_buf + 2);           {right input sample}

{--- sample processing algorithm for left channel ----------------------}

   mr = 0;
   mr1 = ax1;                      {mr = x = left input}

   my0 = aL;
   tap(i2, m2, DL, mx0);           {mx0 = sD = D-th tap}
   mr = mr + mx0 * my0 (rnd);      {mr = s0 = x + a * sD}

   tapin(i2, m2, mr1);             {put s0 in tap-0}
   cdelay(i2, m2);                 {update left delay}

   mr =  mr1 * my0 (rnd);          {mr = a * s0}
   ay0 = mx0;                      {ay0 = sD}
   ar =  ay0 - mr1;                {ar = y = sD - a * s0}

   dm(tx_buf + 1) = ar;            {write left output to codec}

{--- sample processing algorithm for right channel -----------------------}

   mr = 0;
   mr1 = mx1;                      {mr = x = right input}

   my0 = aR;
   tap(i3, m3, DR, mx0);           {mx0 = sD = D-th tap}
   mr = mr + mx0 * my0 (rnd);      {mr = s0 = x + a * sD}

   tapin(i3, m3, mr1);             {put s0 in tap-0}
   cdelay(i3, m3);                 {update right delay}

   mr =  mr1 * my0 (rnd);          {mr = a * s0}
   ay0 = mx0;                      {ay0 = sD}
   ar =  ay0 - mr1;                {ar = y = sD - a * s0}

   dm(tx_buf + 2) = ar;            {write right output to codec}

{--- return from interrupt -----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

Although the overall frequency response of the allpass reverberator is unity, the intermediate stage of computing the recursive part s_0 can overflow because this part is just like the plain reverb and its peak gain is $1/(1-a)$.

The allpass reverberator can also be implemented in its transpose realization form, which avoids overflows better than the canonical form. It is depicted below together with its sample processing algorithm:



```
for each input x do:
    $s_D = \text{tap}(D, w, p, D)$
    $y = s_D - ax$
    $*p = s_0 = x + ay$
    $\text{cdelay}(D, w, \&p)$
```

The program `allpass3.dsp` is the assembly code implementation:

```
{allpass3.dsp - allpass reverb - transpose realization}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on allpass.c of Introduction to Signal Processing.
 Sample processing algorithm from Eq.(8.2.30) and Fig.8.2.17:
       for each x do:
           sD = tap(D, w, p, D)          get D-th tap
           s0 = x + a * sD               input to delay line
           y = -a * s0 + sD              filter output
```

```
           *p = s0                       put input to delay line
           celay(D, w, &p)               update delay line
}

{--- define sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const a = 0x4000;                 {feedback coefficient a = 0.50}
.const D = 3000;                   {TD = D/fs = 3/8 = 0.375 sec}
.var/dm/circ w[D+1];               {delay-line buffer}

i2 = ^w;  L2 = %w;                 {delay-line buffer pointer and length}

zero(i2, m2, L2);                  {clear delay line}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- read input samples from codec ---------------------------------------}

   ax1 = dm(rx_buf + 1);           {left input sample}
   mx1 = dm(rx_buf + 2);           {right input sample}

{--- sample processing algorithm --- process right channel only ----------}

   mr = 0;

   tap(i2, m2, D, mr1);            {mr = sD = D-th tap}

   my1 = a;
   mr = mr - mx1 * my1 (rnd);      {mr = y = sD - a * x}

   sr1 = mr1;                      {sr1 = y}

   mr = 0;
   mr1 = mx1;                      {mr = x}

   mr = mr + sr1 * my1 (rnd);      {mr = s0 = x + a * y}

   tapin(i2, m2, mr1);             {put s0 in tap-0}
   cdelay(i2, m2);                 {update delay}

{--- write output samples to codec ---------------------------------------}

   dm(tx_buf + 1) = sr1;            {left output sample}
   dm(tx_buf + 2) = sr1;            {right output sample}
```

```
{--- return from interrupt ----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

Lab Procedure

a. Go to directory `c:\adi_dsp\examples\allpass`, compile, link, and run the program `allpass.dsp` with the parameter values $D = 3000$ and $a = 0.5$. Repeat with $a = 0.9$.

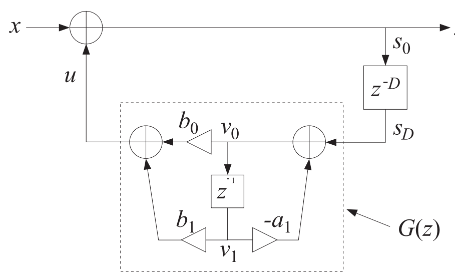
   Listen to the impulse response of the system. Speak into the mike. Compare the output sound with that of the plain reverberator.

b. As in the previous experiment, compare the 60-dB time constants of the case $D = 6000$ and $a = 0.5^2$ and the case $D = 3000$ and $a = 0.5$.

c. Repeat part (a) using the transposed form implemented by the program `allpass3.dsp`. Compare the output to that of `allpass.dsp`.

4.7. Lowpass Reverb

The lowpass reverberator of this experiment is shown in Fig. 8.2.21 of Ref. [1]. Setting $a = -a_1$, the corresponding sample processing algorithm is:



```
for each input sample x do:
    $s_D = \mathrm{tap}(D, w, p, D)$
    $v_0 = a v_1 + s_D$
    $u = b_0 v_0 + b_1 v_1$
    $y = x + u$
    $v_1 = v_0$
    $*p = y$
    $\mathrm{cdelay}(D, w, \&p)$
```

The following program `lowpass.dsp` is an implementation.

```
{lowpass.dsp - lowpass reverb}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on lowpass.c of Introduction to Signal Processing.
 Sample processing algorithm from Eq.(8.2.34) and Fig.8.2.21:
       for each input x do:
           sD = tap(D, w, p, D)                 get D-th tap
           v0 = a * v1 + sD                     input to unit delay
           u = b0 * v0 + b1 * v1                output of feedback filter
           y = x + u                            filter output
           v1 = v0                              update unit delay
           *p = y                               input to D-fold delay
           cdelay(D, w, &p)                     update D-fold delay
```

```
}

{--- define sampling rate in kHz: ---------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-----------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ---------------------------}

.var/dm v1;                        {state of first-order feedback filter}

.const a  = 0x4000;                {a  = 0.50}
.const b0 = 0x199a;                {b0 = 0.20}
.const b1 = 0x0ccd;                {b1 = 0.10}

.const D = 3000;                   {TD = D/fs = 3/8 = 0.375 sec}
.var/dm/circ w[D+1];               {circular delay line}

i2 = ^w;  L2 = %w;                 {delay-line buffer pointer and length}

ax0 = 0;
dm(v1) = ax0;                      {clear feedback delay, v1 = 0}
zero(i2, m2, L2);                  {clear delay line}

{--- start processing input samples -------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- read input samples from codec --------------------------------------}

   ax1 = dm(rx_buf + 1);                  {left input sample}
   mx1 = dm(rx_buf + 2);                  {right input sample}

{--- sample processing algorithm --- process right channel only ----------}

   tap(i2, m2, D, mr1);                   {mr1 = sD = D-th tap}

   mx0 = a;
   my1 = dm(v1);                          {state v1}
   mr = mr + mx0 * my1 (rnd);             {mr = v0 = sD + a * v1}
   if mv sat mr;
   sr0 = mr1;                             {save v0 for later updating v1}

   my0 = b0;
   mr = mr1 * my0 (ss);                   {mr = b0 * v0}
   mx0 = b1;
   mr = mr + mx0 * my1 (rnd);             {mr = u = b0 * v0 + b1 * v1}
   if mv sat mr;

   ay1 = mx1;                             {ay1 = x = input}
   ar = mr1 + ay1;                        {ar = y = x + u = output}
   dm(v1) = sr0;                       {update lowpass filter, v1 = v0}
   tapin(i2, m2, ar);                  {put y in tap-0}
   cdelay(i2, m2);                     {update delay}

{--- write output samples to codec ----------------------------------------}

   dm(tx_buf + 1) = ar;                {left output sample}
   dm(tx_buf + 2) = ar;                {right output sample}

{--- return from interrupt -----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

Lab Procedure

a. Go to the directory `c:\adi_dsp\examples\lowpass`. Using the parameters $D = 3000$, $a = 0.5$, $b_0 = 0.20$, $b_1 = 0.10$, compile and run this program. Listen to its impulse response. Speak into the mike. Notice how successive echoes get more and more mellow as they circulate through the lowpass filter.

b. Try the case $D = 20$, $a = 0$, $b_0 = b_1 = 0.49$. You will hear a guitar-like sound. Repeat for $D = 40$ and $D = 80$. What do you hear? We will encounter these cases again in making a model of a guitar string.

4.8. Schroeder's Reverb

A more realistic reverberation effect can be achieved using Schroeder's model of reverberation, which consists of several plain reverb units in parallel, followed by several allpass units in series. An example is shown in Fig. 8.2.18 of the text [1]. Its sample processing algorithm is given by Eq. (8.2.31) of [1]. The program `reverb.dsp` is an implementation:

```
{reverb.dsp - Schroeder's reverberator}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Fig.8.2.18 of Introduction to Signal Processing.
 Sample processing algorithm is given by Eq.(8.2.31) and the code is
 taken from plain.dsp and allpass.dsp.}

{--- choose sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc85b; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}
```

```
.const b1 = 0x7fff;         {b1 = 0.9999}
.const b2 = 0x7333;         {b2 = 0.90}
.const b3 = 0x6666;         {b3 = 0.80}
.const b4 = 0x599a;         {b4 = 0.70}

.const a1 = 0x70a4;         {a1 = 0.88}
.const a2 = 0x70a4;         {a2 = 0.88}
.const a3 = 0x70a4;         {a3 = 0.88}
.const a4 = 0x70a4;         {a4 = 0.88}
.const a5 = 0x70a4;         {a5 = 0.88}
.const a6 = 0x70a4;         {a6 = 0.88}

.const D1 = 1759;
.const D2 = 1949;
.const D3 = 2113;
.const D4 = 2293;
.const D5 = 307;
.const D6 = 313;

.var/dm/circ w1[D1 + 1];
.var/dm/circ w2[D2 + 1];
.var/dm/circ w3[D3 + 1];
.var/dm/circ w4[D4 + 1];
.var/dm/circ w5[D5 + 1];
.var/dm/circ w6[D6 + 1];

.var/dm x[5];        {dm(x) = input x, dm(x+i) = parallel outputs x_i}

i2 = ^w1;  L2 = %w1;
i3 = ^w2;  L3 = %w2;
i4 = ^w3;  L4 = %w3;
i5 = ^w4;  L5 = %w4;
i6 = ^w5;  L6 = %w5;
i7 = ^w6;  L7 = %w6;

zero(i2, m2, L2);
zero(i3, m3, L3);
zero(i4, m4, L4);
zero(i5, m5, L5);
zero(i6, m6, L6);
zero(i7, m7, L7);

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- read input samples from codec ---------------------------------------}

   ax1 = dm(rx_buf + 1);           {left input sample}
   mx1 = dm(rx_buf + 2);           {right input sample}

{--- sample processing algorithm --- process right channel only ----------}

   si = mx1;                       {read current input into SI}
   sr = ashift si by -3 (hi);      {scale x down by a factor of 8}
   dm(x) = sr1;                    {save x}
```

```
{--- plain-1 --------------------------------------------------------------}

   mr = 0;
   mr1 = dm(x);                     {mr = x = input}

   mx1 = a1;
   tap(i2, m2, D1, my1);            {my1 = sD1 = D1-th tap}
   mr = mr + mx1 * my1 (rnd);       {mr = x1 = x + a1 * sD1}

   tapin(i2, m2, mr1);              {put x1 into tap-0}
   cdelay(i2, m2);                  {update delay}

   dm(x + 1) = mr1;                 {save x1 for later usage}

{--- plain-2 --------------------------------------------------------------}

   mr = 0;
   mr1 = dm(x);                     {mr = x = input}

   mx1 = a2;
   tap(i3, m3, D2, my1);            {my1 = sD2 = D2-th tap}
   mr = mr + mx1 * my1 (rnd);       {mr = x2 = x + a2 * sD2}

   tapin(i3, m3, mr1);              {put x2 into tap-0}
   cdelay(i3, m3);                  {update delay}

   dm(x + 2) = mr1;                 {save x2 for later usage}

{--- plain-3 --------------------------------------------------------------}

   mr = 0;
   mr1 = dm(x);                     {mr = x = input}

   mx1 = a3;
   tap(i4, m4, D3, my1);            {my1 = sD3 = D3-th tap}
   mr = mr + mx1 * my1 (rnd);       {mr = x3 = x + a3 * sD3}

   tapin(i4, m4, mr1);              {put x3 into tap-0}
   cdelay(i4, m4);                  {update delay}

   dm(x + 3) = mr1;                 {save x3 for later usage}

{--- plain-4 --------------------------------------------------------------}

   mr = 0;
   mr1 = dm(x);                     {mr = x = input}

   mx1 = a4;
   tap(i5, m5, D4, my1);            {my1 = sD4 = D4-th tap}
   mr = mr + mx1 * my1 (rnd);       {mr = x4 = x + a4 * sD4}

   tapin(i5, m5, mr1);              {put x4 into tap-0}
   cdelay(i5, m5);                  {update delay}

   dm(x + 4) = mr1;                 {save x4 for later usage}

{--- combine parallel outputs of plain sections --------------------------}
```

```
   mr = 0;
   mx0 = b1; my0 = dm(x+1);
   mr = mr + mx0 * my0 (ss);        {mr = b1 * x1}

   mx0 = b2; my0 = dm(x+2);
   mr = mr + mx0 * my0 (ss);        {mr = b1 * x1 + b2 * x2}

   mx0 = b3; my0 = dm(x+3);
   mr = mr + mx0 * my0 (ss);        {mr = b1 * x1 + b2 * x2 + b2 * x2}

   mx0 = b4; my0 = dm(x+4);
   mr = mr + mx0 * my0 (rnd);       {x5 = b1*x1 + b2*x2 + b2*x2 + b4*x4}
   if mv sat mr;                    {mr = x5}

{--- allpass-1 ------------------------------------------------------------}

   my0 = a5;
   tap(i6, m6, D5, mx0);            {mx0 = sD5 = D5-th tap}
   mr = mr + mx0 * my0 (rnd);       {mr = s05 = x5 + a5 * sD5}

   tapin(i6, m6, mr1);              {put s05 in tap-0 of delay-D5}
   cdelay(i6, m6);                  {update delay}

   mr =  mr1 * my0 (rnd);           {mr = a5 * s05}
   ay0 = mx0;                       {ay0 = sD5}
   ar =  ay0 - mr1;                 {ar = x6 = sD5 - a5 * s05}

{--- allpass-2 ------------------------------------------------------------}

   mr = 0;
   mr1 = ar;                        {mr = x6 = input}

   my0 = a6;
   tap(i7, m7, D6, mx0);            {mx0 = sD6 = D6-th tap}
   mr = mr + mx0 * my0 (rnd);       {mr = s60 = x6 + a6 * sD6}

   tapin(i7, m7, mr1);              {put s06 in tap-0}
   cdelay(i7, m7);                  {update delay}

   mr =  mr1 * my0 (rnd);           {mr = a6 * s06}
   ay0 = mx0;                       {ay0 = sD6}
   ar =  ay0 - mr1;                 {ar = y = sD6 - a6 * s06}

{--- write output samples to codec ---------------------------------------}

   dm(tx_buf + 1) = ar;             {left output sample}
   dm(tx_buf + 2) = ar;             {right output sample}

{--- return from interrupt -----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

There are six multiple delays each requiring its own circular buffer and DAG pointer.

Lab Procedure

a. Go to the directory `c:\adi_dsp\examples\reverb`. Compile and run this program. Listen to its impulse response and speak into the mike.

b. What are the feedback delays of each unit in msec? Replace all the delays by double their values, compile, and run again. Compare the output with that of part (a). Repeat when you triple all the delays. (Note that you can just replace the constant definitions by `.const D1 = 2*291;`, etc.)

c. Repeat parts (a, b) with the following value of the feedback multiplier `a=0x7000`. (What is its decimal value?)

d. On occasion, you can hear quantization overflow effects, especially when the input gets too strong. Introduce appropriate scaling factors for each plain reverb filter and replace the allpass filters by their transposed forms. Recompile and run. Have you managed to eliminate the overflow effects?

4.9. Stereo Reverb

In several of the previous experiments, we considered processing in stereo. In those cases, the left and right channels were processed completely independently of each other. In this experiment, we allow the cross-coupling of the two channels, so that the reverb characteristics of one channel influences those of the other.

An example of such system is given in Problems 8.22 and 8.23 and depicted in Fig. 8.4.1 of the text [1]. Here, we assume that the feedback filters are plain multiplier gains, so that

$$G_L(z) = a_L, \qquad G_R(z) = a_R$$

Each channel has its own delay-line buffer and circular pointer. The sample processing algorithm is modified now to take in a pair of stereo inputs and produce a pair of stereo outputs:

```
for each input stereo pair $x_L, x_R$ do:
    $s_{LL} = \text{tap}(L, w_L, p_L, L)$
    $s_{RR} = \text{tap}(R, w_R, p_R, R)$
    $y_L = c_L x_L + s_{LL}$
    $y_R = c_R x_R + s_{RR}$
    $*p_L = s_{L0} = b_L x_L + a_L s_{LL} + d_R s_{RR}$
    $*p_R = s_{R0} = b_R x_R + a_R s_{RR} + d_L s_{LL}$
    $\text{cdelay}(L, w_L, \&p_L)$
    $\text{cdelay}(R, w_R, \&p_R)$
```

where L and R denote the left and right delays. Cross-coupling between the channels arises because of the coefficients d_L and d_R. The following program `stereo.dsp` is an implementation:

```
{stereo.dsp - stereo delay effects with cross-feedback}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Fig.8.4.1 of Introduction to Signal Processing.
 Assuming the feedback filters are plain multipliers, GL(z)=aL,
 GR(z)=aR, the sample processing algorithm is:
    for each pair xL,xR do:
       sLL = tap(L, wL, pL, L)                   L-th tap of left delay
       sRR = tap(R, wR, pR, R)                   R-th tap of right delay
       yL = cL * xL + sLL                        left output
       yR = cR * xR + sRR                        right output
       *pL = bL * xL + aL * sLL + dR * sRR       input to left delay
       *pR = bR * xR + aR * sRR + dL * sLL       input to right delay
       cdelay(L, wL, &pL)                        update left delay
       cdelay(R, wR, &pR)                        update right delay
}

{--- define sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const aL = 0x0000;              {aL = 0 - left self-feedback}
.const aR = 0x0000;              {aR = 0 - right self-feedback}

.const bL = 0x6666;              {bL = 0.8 - left delay gain}
.const bR = 0x6666;              {bR = 0.8 - right delay gain}

.const cL = 0x4000;              {cL = 0.5 - left direct path gain}
.const cR = 0x4000;              {cR = 0.5 - right direct path gain}

.const dL = 0x4000;              {dL = 0.5 - cross feedback from L to R}
.const dR = 0x4000;              {dR = 0.5 - cross feedback from R to L}

.const L = 3000;                 {TL = L/fs = 3/8 = 0.375 sec}
.const R = 3000;                 {TR = R/fs = 3/8 = 0.375 sec}

.var/dm/circ wL[L+1];            {left delay-line buffer}
.var/dm/circ wR[R+1];            {right delay-line buffer}

i2 = ^wL; L2 = %wL;              {delay-line buffer pointer and length}
i3 = ^wR; L3 = %wR;

zero(i2, m2, L2);                {clear delay line}
zero(i3, m3, L3);

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;           {wait for interrupt and loop forever}
                                 {interrupt service routine starts here}
input_samples: ena sec_reg;      {enable secondary register set}
```

```
{--- read input samples from codec ---------------------------------------}

   mx0 = dm(rx_buf + 1);            {left input sample}
   mx1 = dm(rx_buf + 2);            {right input sample}

{--- sample processing algorithm -----------------------------------------}

   tap(i2, m2, L, sr0);             {sr0 = sLL = L-th tap of left delay}
   tap(i3, m3, R, sr1);             {sr1 = sRR = R-th tap of right delay}

   mr = 0;
   mr1 = sr0;                       {mr1 = sLL}
   my0 = cL;
   mr = mr + mx0 * my0 (rnd);       {mr = yL = cL * xL + sLL = left output}
   if mv sat mr;

   dm(tx_buf + 1) = mr1;            {send left output to codec}

   mr = 0;
   mr1 = sr1;                       {mr1 = sRR}
   my0 = cR;
   mr = mr + mx1 * my0 (rnd);       {mr = yR = cR * xR + sRR = right output}
   if mv sat mr;

   dm(tx_buf + 2) = mr1;            {send right output to codec}

   my0 = bL;
   mr = mx0 * my0 (ss);             {mr = bL * xL}
   my0 = aL;
   mr = mr + sr0 * my0 (ss);        {mr = bL * xL + aL * sLL}
   my0 = dR;
   mr = mr + sr1 * my0 (rnd);       {mr = bL * xL + aL * sLL + dR * sRR}
   if mv sat mr;
   tapin(i2, m2, mr1);              {input to left delay}
   cdelay(i2, m2);                  {update left delay}

   my0 = bR;
   mr = mx1 * my0 (ss);             {mr = bR * xR}
   my0 = aR;
   mr = mr + sr1 * my0 (ss);        {mr = bR * xR + aR * sRR}
   my0 = dL;
   mr = mr + sr0 * my0 (rnd);       {mr = bR * xR + aR * sRR + dL * sLL}
   if mv sat mr;
   tapin(i3, m3, mr1);              {input to right delay}
   cdelay(i3, m3);                  {update right delay}

{--- return from interrupt -----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

Lab Procedure

a. Go to the directory `c:\adi_dsp\examples\stereo`. Compile and run this program. Even though the self-feedback multipliers were set to zero $a_L =$

$a_R = 0$, you will hear repeated echoes bouncing back and forth between the speakers because of the cross-coupling.

b. Next try the case $d_L \neq 0$, $d_R = 0$. And then, $d_L = 0$, $d_R \neq 0$. These choices decouple the influence of one channel but not that of the other.

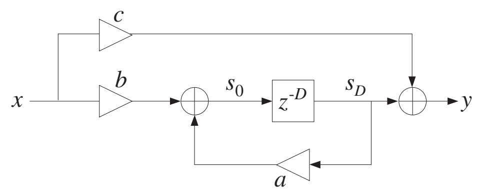
c. Next, introduce some self-feedback, such as $a_L = a_R = 0.5$. Repeat part (a).

4.10. Reverberating Delay

A prototypical delay effect found in most commercial audio effects processors was discussed in Problem 8.17 of the text [1]. Its transfer function is:

$$H(z) = c + b\frac{z^{-D}}{1 - az^{-D}}$$

Its block diagram realization and corresponding sample processing algorithm using a circular delay-line buffer are given below:



```
for each input x do:
    $s_D = \text{tap}(D, w, p, D)$
    $y = cx + s_D$
    $*p = s_0 = bx + as_D$
    $\text{cdelay}(D, w, \&p)$
```

The following program `revdel.dsp` is an implementation:

```
{revdel.dsp - reverberating delay with direct path}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Problem 8.17 of Introduction to Signal Processing.
 Sample processing algorithm:
       for each input x do:
           sD = tap(D, w, p, D);          D-th tap output of delay
           y = sD + c * x;                c = direct path gain, y = output
           s0 = b * x + a * sD;           b = gain before delay
           *p = s0;                       put s0 into delay's tap-0
           cdelay(D, w, &p);              update delay
}

{--- define sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}
```

```
.const a = 0x4000;                 {a = 0.5}     {feedback gain}
.const b = 0x7fff;                 {b = 0.9999}  {gain before delay}
.const c = 0x0000;                 {c = 0}       {gain for direct sound}

.const D = 6000;                   {TD = D/fs = 6/8 = 0.75 sec}
.var/dm/circ w[D+1];               {delay-line buffer}

i2 = ^w;  L2 = %w;                 {buffer pointer and length}

zero(i2, m2, L2);                  {clear delay line}

{--- start processing input samples -------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- read input samples from codec --------------------------------------}

   ax1 = dm(rx_buf + 1);                 {left input sample}
   mx1 = dm(rx_buf + 2);                 {right input sample}

{--- sample processing algorithm --- process right channel only ----------}

   tap(i2, m2, D, sr1);                  {sr1 = sD}
   mr1 = sr1;

   my1 = c;
   mr = mr + mx1 * my1 (rnd);            {mr = y = sD + c * x}
   if mv sat mr;
   ar = mr1;                             {save for output}

   my1 = b;
   mr = mx1 * my1 (ss);                  {mr = b * x}
   my1 = a;
   mr = mr + sr1 * my1 (rnd);            {mr = s0 = b * x + a * sD}
   if mv sat mr;

   tapin(i2, m2, mr1);                   {put s0 into delay-line input}
   cdelay(i2, m2);                       {update delay}

{--- write output samples to codec --------------------------------------}

   dm(tx_buf + 1) = ar;                  {left output sample}
   dm(tx_buf + 2) = ar;                  {right output sample}

{--- return from interrupt ----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;    {wrapup}
```

Lab Procedure

a. Go to the directory `c:\adi_dsp\examples\revdel`. Compile and run this program. Listen to its impulse response and speak into the mike. Here, the direct sound path has been removed, $c = 0$, in order to let the echoes be more

clearly heard.

b. What values of b and c would you use (expressed in terms of a) in order to implement a plain reverberator of the form:

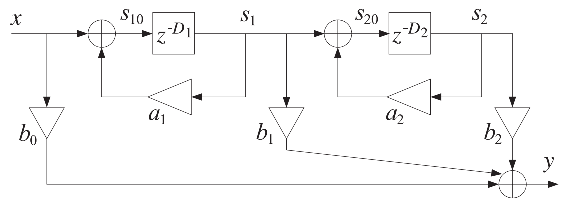
$$H(z) = \frac{1}{1 - az^{-D}}$$

For $a = 0.5$, calculate the proper values of b, c, and then compile and run the program. Compare its output with that of `plain.dsp`.

c. Compile and run the cases $a = 1, b = c = 1$ and $a = -1, b = -1, c = 1$. What are the transfer functions in these cases?

4.11. Multi-Delay Effects

Here, we consider the multi-delay effects processor shown in Fig. 8.2.27 of the text [1]. We assume that the feedback filters are plain multipliers. Using two separate circular buffers for the two delays, the block diagram realization and sample processing algorithm are in this case:



```
for each input x do:
    $s_{1D_1} = \mathrm{tap}(D_1, w_1, p_1, D_1)$
    $s_{2D_2} = \mathrm{tap}(D_2, w_2, p_2, D_2)$
    $y = b_0 x + b_1 s_{1D_1} + b_2 s_{2D_2}$
    $*p_2 = s_{20} = s_{1D_1} + a_2 s_{2D_2}$
    $\mathrm{cdelay}(D_2, w_2, \&p_2)$
    $*p_1 = s_{10} = x + a_1 s_{1D_1}$
    $\mathrm{cdelay}(D_1, w_1, \&p_1)$
```

The program `multidel.dsp` is an implementation.

```
{multidel.dsp - multi-delay effects algorithm}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Fig. 8.2.27 of Introduction to Signal Processing.
 Assuming the feedback filters are plain multipliers, G1(z)=a1,
 G2(z)=a2, and circular buffers for the two delays, the sample
 processing algorithm is:
     for each x do:
          s1D = tap(D1, w1, p1, D1)              D1-th tap output
          s2D = tap(D2, w2, p2, D2)              D2-th tap output
          y = b0 * x + b1 * s1D + b2 * s2D       filter output
          *p2 = s20 = s1D + a2 * s2D             put input s20 into delay-2
          cdelay(D2, w2, &p2)                    update delay-2
          *p1 = s10 = x + a1 * s1D               put input s10 into delay-1
          cdelay(D1, w1, &p1)                    update delay-1
}

{--- define sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const a1 = 0x4000;                {a1 = 0.50}
.const a2 = 0x3333;                {a2 = 0.40}

.const b0 = 0x7fff;                {b0 = 0.9999, that is, 1}
.const b1 = 0x6666;                {b1 = 0.80}
.const b2 = 0x4ccd;                {b2 = 0.60}

.const D1 = 3000;
.const D2 = 4000;

.var/dm/circ w1[D1+1];             {delay-1 buffer}
.var/dm/circ w2[D2+1];             {delay-2 buffer}

i2 = ^w1; L2 = %w1;                {delay-line buffer pointer and length}
i3 = ^w2; L3 = %w2;                {delay-line buffer pointer and length}

zero(i2, m2, L2);                  {clear delay-1}
zero(i3, m3, L3);                  {clear delay-2}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- read input samples from codec ---------------------------------------}

   ax1 = dm(rx_buf + 1);           {left input sample}
   mx1 = dm(rx_buf + 2);           {right input sample}

{--- sample processing algorithm --- process right channel only ----------}

   tap(i2, m2, D1, sr0);           {sr0 = s1D = D1-th tap}
   tap(i3, m3, D2, sr1);           {sr1 = s2D = D2-th tap}

   my0 = b0;
   mr = mx1 * my0 (ss);            {mr = b0 * x}

   my0 = b1;
```

```
   mr = mr + sr0 * my0 (ss);       {mr = b0 * x + b1 * s1D}

   my0 = b2;
   mr = mr + sr1 * my0 (rnd);      {mr = y = b0 * x + b1 * s1D + b2 * s2D}
   if mv sat mr;

{--- write output samples to codec ---------------------------------------}

   dm(tx_buf + 1) = mr1;           {left output sample}
   dm(tx_buf + 2) = mr1;           {right output sample}

{--- sample processing algorithm (cont'd) --------------------------------}

   mr = 0;
   mr1 = sr0;                      {mr = s1D}
   my0 = a2;
   mr = mr + sr1 * my0 (rnd);      {mr = s20 = s1D + a2 * s2D}
   if mv sat mr;
   tapin(i3, m3, mr1);             {put s20 into tap-0 of delay-2}
   cdelay(i3, m3);                 {update delay-2}

   mr = 0;
   mr1 = mx1;                      {mr = x}
   my0 = a1;
   mr = mr + sr0 * my0 (rnd);      {mr = s10 = x + a1 * s1D}
   if mv sat mr;
   tapin(i2, m2, mr1);             {put s10 into tap-0 of delay-1}
   cdelay(i2, m2);                 {update delay-1}

{--- return from interrupt -----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

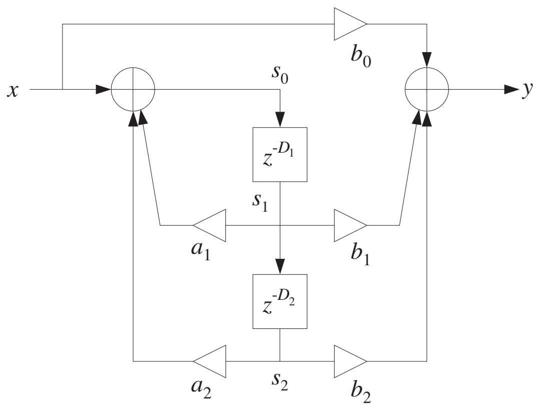
Lab Procedure

a. Go to the directory `c:\adi_dsp\examples\multidel`. Compile and run this program. Listen to its impulse response and speak into the mike.

b. Set $b_1 = 0$ and run again. Then, set $b_2 = 0$ and run. Can you explain what you hear?

4.12. Multitap Delay Effects

This experiment is based on the multi-tap delay line effects processor of Fig. 8.2.29 of the text [1]. It uses a common circular delay-line buffer of order D_1+D_2, which is tapped out at taps D_1 and D_1+D_2. The sample processing algorithm is:



```
for each input sample x do:
    $s_1 = \text{tap}(D_1 + D_2, \mathbf{w}, p, D_1)$
    $s_2 = \text{tap}(D_1 + D_2, \mathbf{w}, p, D_1 + D_2)$
    $y = b_0 x + b_1 s_1 + b_2 s_2$
    $s_0 = x + a_1 s_1 + a_2 s_2$
    $*p = s_0$
    $\text{cdelay}(D_1 + D_2, \mathbf{w}, \&p)$
```

The following program `multitap.dsp` is an assembly language implementation:

```
{multitap.dsp - multitap delay line with feedforward and feedback}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Fig. 8.2.29 of Introduction to Signal Processing.
 The sample processing algorithm is:
     for each x do:
          s1 = tap(D1+D2, w, p, D1)              D1-th tap output
          s2 = tap(D1+D2, w, p, D1+D2)           (D1+D2)-th tap output
          y = b0 * x + b1 * s1 + b2 * s2         filter output
          *p = s0 = x + a1 * s1 + a2 * s2        put input s0 into delay
          cdelay(D1+D2, w, &p)                   update delay
}

{--- define sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}
```

```
.const a1 = 0x199a;                 {a1 = 0.20}
.const a2 = 0x4000;                 {a2 = 0.50}

.const b0 = 0x7fff;                 {b0 = 0.9999, that is, 1}
.const b1 = 0x6666;                 {b1 = 0.80}
.const b2 = 0x4ccd;                 {b2 = 0.60}

.const D1 = 3000;                   {D1/fs = 3000/8000 = 0.375 sec}
.const D2 = 4500;                   {D1+D2 = 7500, 7500/8000 = 0.9375 sec}

.var/dm/circ w[D1+D2+1];            {delay-line buffer}

i2 = ^w; L2 = %w;                   {delay-line buffer pointer and length}

zero(i2, m2, L2);                   {clear delay line}

{--- start processing input samples -------------------------------------}

wait: idle; jump wait;              {wait for interrupt and loop forever}
                                    {interrupt service routine starts here}
input_samples: ena sec_reg;         {enable secondary register set}

{--- read input samples from codec --------------------------------------}

   ax1 = dm(rx_buf + 1);            {left input sample}
   mx1 = dm(rx_buf + 2);            {right input sample}

{--- sample processing algorithm --- process right channel only ----------}

   tap(i2, m2, D1, sr0);            {tap s1 into sr0}
   tap(i2, m2, D1+D2, sr1);         {tap s2 into sr1}

   my0 = b0;
   mr = mx1 * my0 (ss);             {mr = b0 * x}
   my0 = b1;
   mr = mr + sr0 * my0 (ss);        {mr = b0 * x + b1 * s1}
   my0 = b2;
   mr = mr + sr1 * my0 (rnd);       {mr = y = b0 * x + b1 * s1 + b2 * s2}
   if mv sat mr;

{--- write output samples to codec ---------------------------------------}

   dm(tx_buf + 1) = mr1;            {left output sample}
   dm(tx_buf + 2) = mr1;            {right output sample}

{--- sample processing algorithm (cont'd) --------------------------------}

   mr = 0;
   mr1 = mx1;                       {mr = x}
   my0 = a1;
   mr = mr + sr0 * my0 (ss);        {mr = x + a1 * s1}
   my0 = a2;
   mr = mr + sr1 * my0 (rnd);       {mr = s0 = x + a1 * s1 + a2 * s2}
   if mv sat mr;

   tapin(i2, m2, mr1);              {input s0 into delay}
   cdelay(i2, m2);                  {update delay}
```

```
{--- return from interrupt ---------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

Lab Procedure

a. Go to the directory `c:\adi_dsp\examples\multitap`. Compile and run this program. Listen to its impulse response and speak into the mike.

b. Repeat for the following values of the feedback parameters: $a_1 = a_2 = 0.5$, which makes the system marginally stable with a periodic steady output (any random noise would be grow unstable.)

   Repeat also for the case $a_1 = a_2 = 0.75$, which corresponds to an unstable filter. (Please reset the processor before the output grows too loud.)

4.13. Karplus-Strong String Algorithm

A model of a plucked string is obtained by running the lowpass reverb filter with zero input, but with initially filling the delay line with random numbers. These random numbers model the initial harshness of plucking the string. But, as the random numbers recirculate through the lowpass filter, their high frequencies are gradually removed, resulting in a sound that models the string vibration.

The model can be approximately "tuned" to a frequency f_1 by picking D such that $D = f_s/f_1$. (There are ways to "fine-tune", but we do not consider them in this simple experiment.) The Karplus-Strong model assumes a simple averaging FIR filter for the lowpass feedback filter as given by Eq. (8.2.40) of the text [1]. Here, we take the transfer function to be:

$$G(z) = b_0(1 + z^{-1})$$

with some $b_0 \lesssim 0.5$ to improve the stability of the closed-loop system. See text references [108-111] for more discussion on such models. The following program `guitar.dsp` implements the algorithm. The code is identical to that of `lowpass.dsp`:

```
{guitar.dsp - Karplus-Strong string algorithm}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Fig.8.2.21 and Eq.(8.2.40) of Introduction to Signal Processing.
 The filtering algorithm is similar to that of lowpass reverb, with the
 coefficient choices a = 0, b0 = b1 = 0.5. The length of the delay-line
 is designed by tuning it to a desired frequency, i.e., D = fs/f1.
 The buffer is filled with D+1 noise samples and the filter is run
 with zero input. Sample processing code is the same as in lowpass.dsp.
}

{--- define sampling rate in kHz: ---------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
```

```
.const fs = 0xc85b; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-----------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.var/dm v1;                         {state of first-order feedback filter}

.const a  = 0x0000;                 {a  = 0}
.const b0 = 0x3fdf;                 {b0 = 0.499}
.const b1 = 0x3fdf;                 {b1 = 0.499}

.const D = 100;                     {f1 = fs/D = 44100/100 = 441 Hz}
.var/dm/circ w[D+1];                {circular delay line}

i2 = ^w;  L2 = %w;                  {delay-line buffer pointer and length}

.init w: <w.hex>;                   {load w with random numbers}
                                    {w.hex copied from w440hz.hex}
ax0 = 0;
dm(v1) = ax0;                       {clear feedback delay, v1 = 0}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;              {wait for interrupt and loop forever}
                                    {interrupt service routine starts here}
input_samples: ena sec_reg;         {enable secondary register set}

   mx1 = 0;                         {zero input}

{--- sample processing algorithm --- (from lowpass.dsp) ------------------}

   tap(i2, m2, D, mr1);                    {mr1 = sD = D-th tap}

   mx0 = a;
   my1 = dm(v1);                           {state v1}
   mr = mr + mx0 * my1 (rnd);              {mr = v0 = sD + a * v1}
   if mv sat mr;
   sr0 = mr1;                              {save v0 for later updating v1}

   my0 = b0;
   mr = mr1 * my0 (ss);                    {mr = b0 * v0}
   mx0 = b1;
   mr = mr + mx0 * my1 (rnd);              {mr = u = b0 * v0 + b1 * v1}
   if mv sat mr;

   ay1 = mx1;                              {ay1 = x = input}
   ar = mr1 + ay1;                         {ar = y = x + u = output}

   dm(v1) = sr0;                           {update lowpass filter, v1 = v0}
   tapin(i2, m2, ar);                      {put y in tap-0}
   cdelay(i2, m2);                         {update delay}

{--- write output samples to codec --------------------------------------}
```

```
      dm(tx_buf + 1) = ar;                   {left output sample}
      dm(tx_buf + 2) = ar;                   {right output sample}

   {--- return from interrupt ----------------------------------------------}

      rti;

   .include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

In this experiment, the sampling rate is set to 44.1 kHz and the generated sound is the note A440, that is having frequency 440 Hz. The correct amount of delay is then

$$D = \frac{f_s}{f_1} = \frac{44100}{440} = 100$$

The delay line must be filled with $D+1$ random numbers. They were generated as follows by the routine `uran.c` given in the Appendix:

```
uran 0 1 100 2001 | dec2hex 1.15 > w440hz.hex
```

where the seed value was arbitrary and the output of `uran` was piped into `dec2hex` whose output was the hex file `w440hz.hex`. Similarly, the file `w220hz.hex` contains twice as many random numbers which are used to generate the frequency 220 Hz.

Lab Procedure

a. Go to the directory `c:\adi_dsp\examples\guitar`. Set $D = 100$ in the program, copy `w440hz.hex` into `w.hex` and run. The program disables the inputs and simply outputs the re-circulating and gradually decaying random numbers.

b. Repeat by copying `w220hz.hex` into `w.hex` and editing the value $D = 200$ into the program. The note you hear should be an octave lower.

4.14. Wavetable Generators

Here, we present some examples of wavetable generators using the macro `wavgen.dsp`. Two wavetables can be used in combination to illustrate AM and FM modulation.

Sinusoidal Wavetable

The first program, `sine.dsp`, generates a simple sinusoid.

```
   {sine.dsp - sinusoid generated from a wavetable}
   {Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

   {Based on Sec.8.1.3 of Introduction to Signal Processing}

   {--- choose sampling rate in kHz: ---------------------------------------}
```

```
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-----------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const D = 4000;
.var/dm/circ sine[D];              {min frequency f1 = fs/D = 8/4 = 2 Hz}

.init sine: <sinetbl.hex>;         {load one period of the wavetable}
                                   {generated by sinetbl.c}
.const A = 0x7fff;                 {A = 1 = signal amplitude}
.const c = 100;                    {signal frequency f = c * f1 = 200 Hz}

i6 = ^sine; L6 = %sine;            {pointer for signal generator}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- sample processing algorithm -----------------------------------------}

   wavgen(i6, m6, A, c, mx1);            {mx1 = A * sin(2*pi*f*t)}

{--- write output samples to codec ---------------------------------------}

   dm(tx_buf + 1) = mx1;           {left output sample}
   dm(tx_buf + 2) = mx1;           {right output sample}

{--- return from interrupt -----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;    {wrapup}
```

The sinusoidal wavetable has period $D = 4000$ and is read from the file `sinetbl.hex`, which was generated by the DOS commands:

```
sinetbl 0 1 4000 > sinetbl.dec
dec2hex 1.15 < sinetbl.dec > sinetbl.hex
```

Therefore, the smallest frequency that can be generated by the wavetable at an 8 kHz sampling rate is $f_1 = f_s/D = 8000/4000 = 2$ Hz. Higher frequencies can be selected by the parameter c resulting in $f = cf_1 = cf_s/D$.

At each sampling instant, the program does nothing with the codec inputs. Instead, it generates a sample of a sinusoid by a call to `wavgen` and sends it to the codec.

AM Modulation

The next program, `am.dsp`, illustrates AM modulation. The envelope frequency is chosen to be 2 Hz and the signal frequency 200 Hz. The AM-modulated signal is of the form:

$$x(t)=A(t)\sin(2\pi f t), \qquad \text{where} \quad A(t)=A_{\text{env}}\sin(2\pi f_{\text{env}}t)$$

A common sinusoidal wavetable, `sinetbl.hex`, is used to generate both the signal and its sinusoidal envelope.

```
{am.dsp - AM modulation of using common wavetable}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Sec.8.1.3 of Introduction to Signal Processing}

{--- choose sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const D = 4000;
.var/dm/circ sine[D];              {min frequency f1 = fs/D = 8/4 = 2 Hz}

.init sine: <sinetbl.hex>;         {load one period of the wavetable}
                                   {generated by sinetbl.c}
.const Aenv = 0x7fff;              {Aenv = 1 = envelope amplitude}
.const cenv = 1;                   {envelope fenv = cenv * f1 = 2 Hz}

i5 = ^sine; L5 = %sine;            {pointer for envelope generator}

.const c = 100;                    {signal frequency f = c * f1 = 200 Hz}

i6 = ^sine; L6 = %sine;            {pointer for signal generator}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- sample processing algorithm -----------------------------------------}

   wavgen(i5, m5, Aenv, cenv, ax1);        {A(t) = Aenv * sin(2*pi*fenv*t)}
   wavgen(i6, m6, ax1, c, mx1);            {A(t) * sin(2*pi*f*t)}

{--- write output samples to codec ---------------------------------------}

   dm(tx_buf + 1) = mx1;           {left output sample}
   dm(tx_buf + 2) = mx1;           {right output sample}
```

```
{--- return from interrupt ---------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

FM Modulation

The third program, `fm.dsp`, illustrates FM modulation in which the frequency of a sinusoid is itself varying sinusoidally. The generated signal is of the form

$$x(t)=\sin\big(2\pi f(t)t\big)$$

Because the frequency of the sinusoid is passed into the wavetable generator via the parameter c, we define the frequency $f(t)$ in the form $f(t)=c(t)f_s/D$, where

$$c(t)=c_0+A_m\sin(2\pi f_m t)$$

We choose the modulation depth $A_m = 0.3c_0$, so that $c(t)$ varies sinusoidally between the limits $0.7c_0 \le c(t)\le 1.3c_0$, and therefore, the frequency will vary in the limits $0.7f_0 \le f(t)\le 1.3f_0$, where $f_0 = c_0 f_s/D$. The center frequency is $f_0 = 200$ Hz, corresponding to $c_0 = 100$ and $A_m = 0.3c_0 = 30$.

The same wavetable, `sinetbl.hex`, is used to generate both the parameter $c(t)$ and the modulated signal $x(t)$. The modulation frequency is $f_m = c_m f_s/D$, where $c_m = 1$ resulting in $f_m = 2$ Hz.

```
{fm.dsp - FM modulation of using common sinusoidal wavetable}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Sec.8.1.3 of Introduction to Signal Processing}

{--- choose sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const D = 4000;
.var/dm/circ sine[D];              {min frequency f1 = fs/D = 8/4 = 2 Hz}

.init sine: <sinetbl.hex>;         {load one period of the wavetable}
                                   {generated by sinetbl.c}
.const Am = 30;                    {Am = 0.30 * c0}
.const cm = 1;                     {fm = cm * fs/D = 2 Hz}

i5 = ^sine; L5 = %sine;            {pointer for frequency generator}
```

```
.const A = 0x4000;                {A = 0.50 = signal amplitude}
.const c0 = 100;                  {signal freq f0 = c0 * f1 = 200 Hz}

i6 = ^sine; L6 = %sine;           {pointer for signal generator}

{--- start processing input samples -------------------------------------}

wait: idle; jump wait;            {wait for interrupt and loop forever}
                                  {interrupt service routine starts here}
input_samples: ena sec_reg;       {enable secondary register set}

{--- sample processing algorithm ----------------------------------------}

   wavgen(i5, m5, Am, cm, ay1);   {ay1 = Am * sin(2*pi*fm*t)}
   ax1 = c0;
   ar = ax1 + ay1;                {ar = c = c0 + Am * sin(2*pi*fm*t)}
   wavgen(i6, m6, A, ar, mx1);    {mx1 = FM-modulated sinusoid}

{--- write output samples to codec --------------------------------------}

   dm(tx_buf + 1) = mx1;          {left output sample}
   dm(tx_buf + 2) = mx1;          {right output sample}

{--- return from interrupt ----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

Lab Procedure

a. Go to directory `c:\adi_dsp\examples\wavetabl`. Run the program `sine.dsp` and listen to the 200 Hz sinusoid. Reset the frequency to 50 Hz, recompile and run. Keep decreasing the frequency by 10 Hz each time and determine the lowest frequency you can hear (but, to be fair don't increase the speaker volume; that would compensate the attenuation introduced by your ears.)

b. Set the frequency at 4000 Hz (the Nyquist frequency.) Recompile and run. Can you explain what you hear? Replace the sinusoidal wavetable, `sinetbl.hex`, with a cosinusoidal one and repeat the experiment at the Nyquist frequency.

c. Replace the sinusoidal table `sinetbl.hex` with the square wavetable `squartbl.hex`, which has period 4000 and is equal to $+1$ for the first half of the period and -1 for the second half. Run the program with frequency $f = 200$ Hz. The wavetable amplitude was chosen to be $A = 1/\sqrt{2}$ in order to make the rms value of the square wave equal to the rms value of the original sinusoid.

d. Run and listen to the program `am.dsp`, with the initial signal frequency of $f = 200$ Hz and envelope frequency of $f_{\text{env}} = 2$ Hz. Repeat for $f = 2000$ Hz. Repeat and explain what you hear for the cases: $f = 200$ Hz, $f_{\text{env}} = 100$ Hz. Then, $f = 200$ Hz, $f_{\text{env}} = 190$ Hz. Then, $f = 200$ Hz, $f_{\text{env}} = 200$ Hz.

e. Run and hear the program `fm.dsp` with the following three values of the modulation depth: $A_m = 0.3c_0$, $A_m = c_0$, $A_m = 0.1c_0$. Repeat these cases when the center frequency is changed to $f_0 = 2000$ Hz.

f. Replace the sinusoidal wavetable with the square one, `squartbl.hex`, and repeat the case $f_0 = 200$ Hz, $A_m = 0.3c_0$. You will hear a square wave whose frequency switches between a high and a low value two times a second.

g. Keep the square wavetable that generates the alternating frequency, but generate the signal by a sinusoidal wavetable. To do this , generate a second sinusoidal wavetable, say of length 800, and define a circular buffer for it and set one of the DAG registers, e.g., `i6`, to point to it. Then generate your FM-modulated sinusoid using this table. The generated signal will be of the form:

$$x(t) = \sin\big(2\pi f(t)t\big), \qquad f(t) = 2 \text{ Hz square wave}$$

h. Rewrite the `fir3.dsp` program of Section 4.3 using calls to the wavetable generator macro `wavgen`. Repeat part (c) of the lab procedure of that Section.

4.15. Notch Filters

In these experiments, we demonstrate the use of filtering for canceling periodic interference. The input signal is of the form:

$$x(n) = s(n) + v(n)$$

where $s(n)$ is the microphone input and $v(n)$ a periodic interference signal generated internally using a wavetable generator.

Multi-Notch Filter

When the noise is periodic, its energy is concentrated at the harmonics of the fundamental frequency: f_1, $2f_1$, $3f_1$, and so on. To cancel the entire noise component, we must use a filter with multiple notches at these harmonics.

As discussed in Section 8.3.2 of the text [1], a simple design arises when the period of the noise is an integral multiple of the sampling period, that is, $T_1 = DT$, which implies that the fundamental frequency f_1 and its harmonics will be:

$$f_1 = \frac{f_s}{D}, \quad f_k = kf_1 = k\frac{f_s}{D}, \quad k = 0, 1, 2, \dots$$

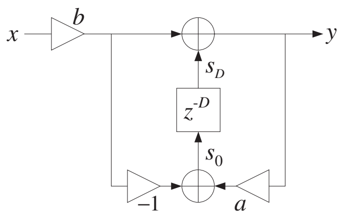
or, in units of radians per sample:

$$\omega_1 = \frac{2\pi}{D}, \quad \omega_k = k\omega_1 = \frac{2\pi k}{D}$$

These are recognized as the D-th root-of-unity frequencies. The corresponding notch filter, designed by Eqs. (8.3.26) and (8.3.27) of the text, has the form:

$$H(z) = b\frac{1 - z^{-D}}{1 - az^{-D}}$$

To avoid overflows, we use the transposed realization of this transfer function, whose block diagram and sample processing algorithm are shown below:



```
for each input x do:
    $s_D$ = tap$(D, w, p, D)$
    $y = bx + s_D$
    $*p = s_0 = ay - bx$
    cdelay$(D, w, \&p)$
```

A quick way to understand this transposed realization is to write:

$$H(z) = \frac{Y(z)}{X(z)} = b\frac{1 - z^{-D}}{1 - az^{-D}},$$

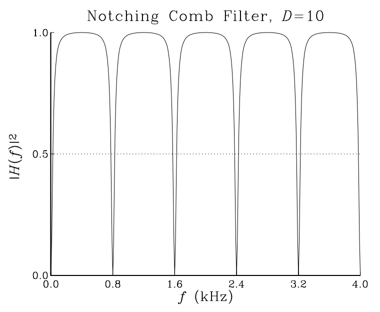
from where we obtain the I/O equation on which the block diagram is based:

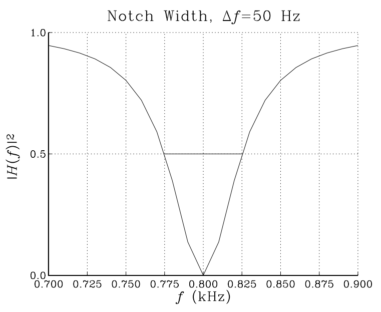
$$Y(z) = bX(z) + z^{-D}\big(aY(z) - bX(z)\big)$$

In the experiment, we take the fundamental period to be 800 Hz and the sampling rate 8 kHz. Thus, the period D is:

$$D = \frac{f_s}{f_1} = \frac{8000\text{ Hz}}{800\text{ Hz}} = 10$$

The width of the notches is taken to be $\Delta f = 50$ Hz. Then, the design equations (8.3.27) give the parameter values $b = 0.91035$, $a = 0.8207$. Their, 1.15 hex equivalents are `0x7486` and `0x690d`, respectively. The magnitude response of this filter plotted over the right-half of the Nyquist interval is shown below, together with a magnified view of the notch width at 800 Hz:





Using a square wavetable, the program `notch.dsp` generates a square wave of period $D = 10$ and adds it to the microphone input. The resulting signal is then filtered by the above multi-notch filter, removing the periodic noise.

```
{notch.dsp - notching comb filter canceling 800 Hz harmonics}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Eq.(8.3.26) and Example 8.3.9 of Introd. to Signal Processing.
 Input signal is microphone input + 800 Hz square wave:

       x = A0 * (mike) + A * (square wave)

 To better avoid overflows, we use the transpose realization with
 sample processing algorithm:
       for each input x do:
              sD = tap(D, w, p, D)
              y = b * x + sD
              *p = s0 = a * y - b * x
              cdelay(D, w, &p)
}

{--- define sampling rate in kHz: ---------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-----------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ---------------------------}

.const A0 = 0x4000;                {A0 = 0.5 = input scale factor}

.const D = 10;                     {D = fs/f1 = period of square wave}
.var/dm/circ w[D+1];               {filter's delay-line buffer}

i2 = ^w; L2 = %w;                  {delay-line buffer pointer and length}

zero(i2, m2, L2);                  {clear delay line}

.const Ds = 800;                   {wavetable's min freq: fs/Ds = 10 Hz}
.var/dm/circ s[Ds];                {square wavetable}

i6 = ^s; L6 = %s;                  {wavetable pointer and length}

.const c = 80;                     {square wave frequency f1 = c * fs/Ds}
.const A = 0x1000;                 {A = A0/4 = square wave amplitude}

.init s: <squartbl.hex>;           {load wavetable}

.const a = 0x690d;                 {coefficient a = 0.8207 = 0x690d}
.const b = 0x7486;                 {coefficient b = 0.91035= 0x7486}
                                   {design specs: f1 = 800 Hz, Df = 50 Hz}
{--- start processing input samples -------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- read input samples from codec --------------------------------------}
```

```
   ax1 = dm(rx_buf + 1);                  {left input sample}
   mx1 = dm(rx_buf + 2);                  {right input sample}

{--- sample processing algorithm ------------------------------------------}

   wavgen(i6, m6, A, c, mr1);             {mr1 = A * square wave}

   my1 = A0;
   mr = mr + mx1 * my1 (rnd);             {mr = x = A0 * mike + A * square}
   if mv sat mr;

  {ar = mr1; jump nofilter;}              {uncomment to bypass filter}

   mx1 = mr1;                             {save x temporarily}

   my0 = b;
   mr = mx1 * my0 (rnd);                  {mr1 = b * x}
   if mv sat mr;

   tap(i2, m2, D, ay1);                   {ay1 = sD = D-th tap}
   ar = mr1 + ay1;                        {ar = y = b * x + sD = output}

   my1 = a;
   mr = ar * my1 (ss);                    {mr = a * y}
   my1 = b;
   mr = mr - mx1 * my1 (rnd);             {mr = s0 = a * y - b * x}
   if mv sat mr;

   tapin(i2, m2, mr1);                    {put s0 into tap-0}
   cdelay(i2, m2);                        {update delay line}

{--- write samples to codec ---------------------------------------------}

nofilter:

   dm(tx_buf + 1) = ar;                   {left output sample}
   dm(tx_buf + 2) = ar;                   {right output sample}

{--- return from interrupt -----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

The filtering operation can be bypassed by uncommenting the `jump nofilter` instruction, in order to hear the desired signal plus the noise. The particular square wave of period 10 generated by the program is of the form:

$$v(n) = [1, 1, 1, 1, 1, -1, -1, -1, -1, -1, \dots]$$

It contains only odd harmonics. As discussed in Example 1.4.6 and Section 9.7 of the text, all harmonics that lie outside the Nyquist interval are wrapped inside the interval and get aliased with the harmonics within the interval. Thus, the above periodic signal contains only the harmonics $\omega_1 = 2\pi/10$, $\omega_3 = 3\omega_1 = 6\pi/10$, and $\omega_5 = 5\omega_1 = 10\pi/10 = \pi$. In fact, we can show using the techniques of Section

9.7 of the text that the signal $v(n)$ can be expressed in the alternative sinusoidal form, obtained from the 10-point DFT of one period of the square wave:

$$v(n) = \frac{0.4}{\sin(\frac{\omega_1}{2})}\sin(\omega_1 n + \frac{\omega_1}{2}) + \frac{0.4}{\sin(\frac{\omega_3}{2})}\sin(\omega_3 n + \frac{\omega_3}{2}) + 0.2\cos(\omega_5 n)$$

Thus, the filter only acts to remove these three odd harmonics. It may appear puzzling that the Fourier series expansion of this square wave does not contain exclusively sine terms, as it would in the continuous-time case. This discrepancy can be traced to the discontinuity of the square wave. In the continuous-time case, any finite Fourier series sinusoidal approximation to the square wave will vanish at the discontinuity points. Therefore, a more appropriate discrete-time square wave might be of the form:

$$v(n) = [0, 1, 1, 1, 1, 0, -1, -1, -1, -1, \dots]$$

The DOS square wavetable generator `squartbl.exe` has a command-line option that allows one to select this type of square wave. Using again the techniques of Section 9.7, we find for its inverse DFT expansion:

$$v(n) = \frac{0.4}{\tan(\frac{\omega_1}{2})}\sin(\omega_1 n) + \frac{0.4}{\tan(\frac{\omega_3}{2})}\sin(\omega_3 n)$$

where now only pure sines (as opposed to sines and cosines) appear. The difference between the above two square waves contains the effect of the discontinuities and is given by

$$v(n) = [1, 0, 0, 0, 0, -1, 0, 0, 0, 0, \dots]$$

Its discrete Fourier series is the difference of the above two:

$$v(n) = 0.4\cos(\omega_1 n) + 0.4\cos(\omega_3 n) + 0.2\cos(\omega_5 n)$$

Lab Procedure

a. Go to subdirectory `c:\adi_dsp\examples\notch`. Generate two square wavetables of the above two types by the DOS commands:

```
squartbl 1 -1 400 800 1 | dec2hex 1.15 > squartbl.hex
squartbl 1 -1 400 800 0 | dec2hex 1.15 > squartb2.hex
```

Compile and run the program `notch.dsp` with the filter off. Speak into the mike and listen to the interference. Repeat with the filter on. Repeat with the filter on, but using the second wavetable. Turn off the interference by setting its amplitude $A = 0$ and listen to the effect the filter has on your voice input.

b. Estimate the 60-dB time constant (in seconds) of the filter in part (a). Redesign the notch filter so that its 3-dB width is now $\Delta f = 2$ Hz. What is the new time constant? Run the new filter and listen to the filter transients as the steady-state gradually takes over and suppresses the noise. Turn off the square wave, recompile and run. Listen to the impulse response of the filter by lightly tapping the mike on the table. Can you explain what you are hearing?

c. Generate a square wave with frequency of 1000 Hz, corresponding to wavetable increment $c = 100$. Repeat part (a). Now the interference harmonics do not coincide with the filter's notches and you will still hear the interference.

d. Design the correct multi-notch filter that should be used in part (b). Edit the program `notch.dsp` to reflect the new design, and run it to verify that it does indeed remove the 1000 Hz interference.

Single- and Double-Notch Filters

Using a single-notch filter with notch frequency at $f_1 = 800$ Hz instead of the multi-notch filter would not be sufficient to cancel completely the square wave interference. The third and higher harmonics will survive it. Such a single-notch filter can be designed with Eqs. (8.2.22) and (8.2.23) of the text, which are implemented by the MATLAB function `parmeq.m`. Assuming the same width $\Delta f = 50$ Hz, the transfer function will be:

$$H_1(z) = \frac{0.980741 - 1.586872z^{-1} + 0.980741z^{-2}}{1 - 1.586872z^{-1} + 0.961481z^{-2}}$$

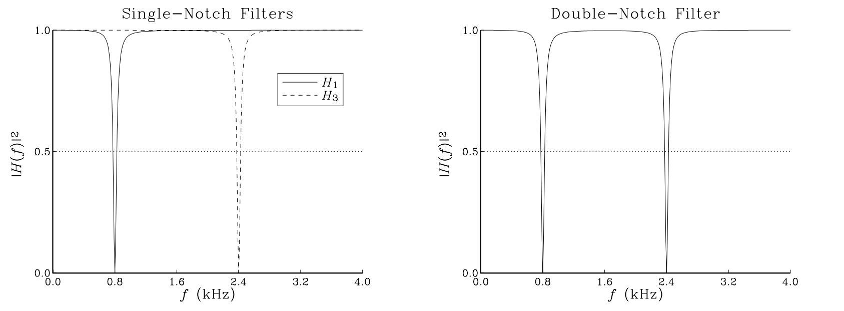
A similar design with a notch at $f_3 = 3f_1 = 2400$ Hz gives:

$$H_3(z) = \frac{0.980741 + 0.606131z^{-1} + 0.980741z^{-2}}{1 + 0.606131z^{-1} + 0.961481z^{-2}}$$

The cascade of the two is a fourth-order filter of the form $H_{13}(z) = H_1(z)H_3(z)$ with coefficients obtained by convolving the coefficients of filter-1 and filter-3:

$$H_{13}(z) = \frac{0.961852(1 - z^{-1} + z^{-2} - z^{-3} + z^{-4})}{1 - 0.980741z^{-1} + 0.961111z^{-2} - 0.942964z^{-3} + 0.924447z^{-4}}$$

The magnitude responses of the two single-notch filters $H_1(z)$, $H_3(z)$ and of the double-notch filter $H_{13}(z)$ are shown below:



The program `notch1.dsp` implements the filter $H_1(z)$ in its *direct form* using the macro `cdir`. The coefficients must be scaled down by a factor of 2 to make them fit into the 1.15 format. Thus, we must use scaling exponents $e_a = e_b = 1$.

```
{notch1.dsp - single-notch filter at 800 Hz & width 50 Hz - direct form}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Designed with parmeq.m of Appendix D, with the MATLAB commands:
       GB = 1/sqrt(2); f0 = 800; fs = 8000; Df = 50;
       [b, a, beta] = parmeq(1, 0, GB, 2*pi*f0/fs, 2*pi*Df/fs);
}


{--- define sampling rate in kHz: ---------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-----------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ---------------------------}

.const A0 = 0x4000;                {A0 = 0.5 = input scale factor}

.const M = 2;                      {filter order}
.var/dm/circ w[M+1];               {y-delay-line buffer in DM}
.var/dm/circ v[M+1];               {x-delay-line buffer in DM}
.var/pm/circ a[M+1], b[M+1];       {concatenated coeffs in PM}

.const ea = 1;                     {scaling exponent for a}
.const eb = 1;                     {scaling exponent for b}
.const ex = 0;                     {input scaling exponent}

i2 = ^w; L2 = %w;                  {y-delay-line pointer}
i3 = ^v; L3 = %v;                  {x-delay-line pointer}
i4 = ^a; L4 = 2*(M+1);             {double-length a,b coefficients}

zero(i2, m2, L2);                  {clear y-delay line}
zero(i3, m3, L3);                  {clear x-delay line}

.const Ds = 800;                   {wavetable's min freq: fs/Ds = 10 Hz}
.var/dm/circ s[Ds];                {square wavetable}

i6 = ^s; L6 = %s;                  {wavetable pointer and length}

.const c = 80;                     {square wave freq: f1 = c * fs/Ds}
.const A = 0x1000;                 {A = A0/4 = square wave amplitude}

.init s: <squartbl.hex>;           {load wavetable}
.init a: <a.hex>;                  {denominator coefficients}
.init b: <b.hex>;                  {numerator coefficients}

{--- start processing input samples -------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
```

```
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- read input samples from codec ---------------------------------------}

   ax1 = dm(rx_buf + 1);                 {left input sample}
   mx1 = dm(rx_buf + 2);                 {right input sample}

{--- sample processing algorithm -----------------------------------------}

   wavgen(i6, m6, A, c, mr1);            {mr1 = A * square wave}

   my1 = A0;
   mr = mr + mx1 * my1 (rnd);            {mr = x = A0 * mike + A * square}
   if mv sat mr;

  {sr1 = mr1; jump nofilter;}            {uncomment to bypass filter}

   sr = ashift mr1 by -ex (hi);                        {pre-scale x down}
                                                       {compute output y}
   cdir(M, i4, m4, i2, m2, i3, m3, ea, eb, sr1);       {input from sr1}
                                                       {output in ar}
   sr = ashift ar by ex (hi);                          {post-scale y up}

{--- write samples to codec ---------------------------------------------}

nofilter:

   dm(tx_buf + 1) = sr1;                  {left output sample}
   dm(tx_buf + 2) = sr1;                  {right output sample}

{--- return from interrupt ----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

The program notch2.dsp implements the filter $H_1(z)$ in its canonical form using the macro ccan. To prevent overflows, the input signal must be scaled down by a factor of $16 = 2^4$ before it is passed to the filter, and the output scaled up by the same factor. Thus, the input scaling exponent is $e_x = 4$.

```
{notch2.dsp - single-notch filter at 800 Hz & width 50 Hz - canonical form}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Designed with parmeq.m of Appendix D, with the MATLAB commands:
       GB = 1/sqrt(2); f0 = 800; fs = 8000; Df = 50;
       [b, a, beta] = parmeq(1, 0, GB, 2*pi*f0/fs, 2*pi*Df/fs);
}

{--- define sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-------------------------------------------------------------------------}
```

```
.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const A0 = 0x4000;                 {A0 = 0.5 = input scale factor}

.const M = 2;                       {filter order}
.var/dm/circ w[M+1];                {circular delay-line buffer in DM}
.var/pm/circ a[M+1], b[M+1];        {concatenated coeffs in PM}

.const ea = 1;                      {scaling exponent for a}
.const eb = 1;                      {scaling exponent for b}
.const ex = 4;                      {input scaling exponent}

i2 = ^w; L2 = %w;                   {delay-line buffer pointer and length}
i4 = ^a; L4 = 2*(M+1);              {double-length a,b coefficients}

zero(i2, m2, L2);                   {clear delay line}

.const Ds = 800;                    {wavetable's min freq: fs/Ds = 10 Hz}
.var/dm/circ s[Ds];                 {square wavetable}

i6 = ^s; L6 = %s;                   {wavetable pointer and length}

.const c = 80;                      {square wave frequency f1 = c * fs/Ds}
.const A = 0x1000;                  {A = A0/4 = square wave amplitude}

.init s: <squartbl.hex>;            {load wavetable}
.init a: <a.hex>;                   {denominator coefficients}
.init b: <b.hex>;                   {numerator coefficients}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;              {wait for interrupt and loop forever}
                                    {interrupt service routine starts here}
input_samples: ena sec_reg;         {enable secondary register set}

{--- read input samples from codec ---------------------------------------}

   ax1 = dm(rx_buf + 1);                 {left input sample}
   mx1 = dm(rx_buf + 2);                 {right input sample}

{--- sample processing algorithm -----------------------------------------}

   wavgen(i6, m6, A, c, mr1);            {mr1 = A * square wave}

   my1 = A0;
   mr = mr + mx1 * my1 (rnd);            {mr = x = A0 * mike + A * square}
   if mv sat mr;

  {sr1 = mr1; jump nofilter;}            {uncomment to bypass filter}

   sr = ashift mr1 by -ex (hi);                 {pre-scale x down}
                                                {compute output y}
   ccan(M, i4, m4, i2, m2, ea, eb, sr1);        {input from sr1}
                                                {output in sr1}
   sr = ashift sr1 by ex (hi);                  {post-scale y up}
```

```
{--- write samples to codec --------------------------------------------}

nofilter:

   dm(tx_buf + 1) = sr1;                  {left output sample}
   dm(tx_buf + 2) = sr1;                  {right output sample}

{--- return from interrupt ----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

The program `notch13.dsp` implements the filter $H_{13}(z)$ in its direct form. The coefficients already fit in 1.15 format and do not need any further scaling.

```
{notch3.dsp - double-notch filter at 800 Hz and 2400 Hz - canonical form}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Designed with parmeq.m of Appendix D, with the MATLAB commands:
       GB = 1/sqrt(2); f1 = 800; f3 = 2400; fs = 8000; Df = 50;
       [b1, a1, beta1] = parmeq(1, 0, GB, 2*pi*f1/fs, 2*pi*Df/fs);
       [b3, a3, beta3] = parmeq(1, 0, GB, 2*pi*f3/fs, 2*pi*Df/fs);
       a13 = conv(a1, a3); b13 = conv(b1, b3);
}

{--- define sampling rate in kHz: ---------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-----------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ---------------------------}

.const A0 = 0x4000;               {A0 = 0.5 = input scale factor}

.const M = 4;                     {filter order}
.var/dm/circ w[M+1];              {circular delay-line buffer in DM}
.var/pm/circ a[M+1], b[M+1];      {concatenated coeffs in PM}

.const ea = 0;                    {scaling exponent for a}
.const eb = 0;                    {scaling exponent for b}
.const ex = 3;                    {input scaling exponent}

i2 = ^w; L2 = %w;                 {delay-line buffer pointer and length}
i4 = ^a; L4 = 2*(M+1);            {double-length a,b coefficients}

zero(i2, m2, L2);                 {clear delay line}

.const Ds = 800;                  {wavetable's min freq: fs/Ds = 10 Hz}
.var/dm/circ s[Ds];               {square wavetable}

i6 = ^s; L6 = %s;                 {wavetable pointer and length}
```

```
.const c = 80;                    {square wave frequency f1 = c * fs/Ds}
.const A = 0x1000;                {A = A0/4 = square wave amplitude}

.init s: <squartbl.hex>;          {load wavetable}
.init a: <a13.hex>;               {denominator coefficients}
.init b: <b13.hex>;               {numerator coefficients}

{--- start processing input samples ------------------------------------}

wait: idle; jump wait;            {wait for interrupt and loop forever}
                                  {interrupt service routine starts here}
input_samples: ena sec_reg;       {enable secondary register set}

{--- read input samples from codec --------------------------------------}

   ax1 = dm(rx_buf + 1);                  {left input sample}
   mx1 = dm(rx_buf + 2);                  {right input sample}

{--- sample processing algorithm ----------------------------------------}

   wavgen(i6, m6, A, c, mr1);             {mr1 = A * square wave}

   my1 = A0;
   mr = mr + mx1 * my1 (rnd);             {mr = x = A0 * mike + A * square}
   if mv sat mr;

  {sr1 = mr1; jump nofilter;}             {uncomment to bypass filter}

   sr = ashift mr1 by -ex (hi);                 {pre-scale x down}
                                                {compute output y}
   ccan(M, i4, m4, i2, m2, ea, eb, sr1);        {input from sr1}
                                                {output in sr1}
   sr = ashift sr1 by ex (hi);                  {post-scale y up}

{--- write samples to codec --------------------------------------------}

nofilter:

   dm(tx_buf + 1) = sr1;                  {left output sample}
   dm(tx_buf + 2) = sr1;                  {right output sample}

{--- return from interrupt ----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

Lab Procedure

a. Run the program `notch1.dsp` with the filter off. Then, run it with the filter on. Do you hear the partial suppression of the interference? Next, replace the square wavetable with a sinusoidal wavetable generated by the DOS command:

```
sinetbl 0 1 800 | dec2hex 1.15 > sinetbl.hex
```

and replace `squartbl.hex` by `sinetbl.hex`. Now, the interference is one sinusoid at 800 Hz, and therefore it will be completely canceled by the filter.

b. Run the program `notch2.dsp`. Repeat by choosing the smaller values of the input scaling exponent: $e_x = 3, 2, 1, 0$. Listen to the overflow effects.

c. Can you explain theoretically why in the numerator of $H_{13}(z)$ you have the polynomial with alternating coefficients $1 - z^{-1} + z^{-2} - z^{-3} + z^{-4}$?

d. Run the program `notch13.dsp` using the wavetable `squartbl.hex`. Listen to the suppression of the harmonics at f_1 and f_3. However, the harmonic $f_5 = 4000$ Hz (i.e., the Nyquist frequency) can still be heard.

Next, replace `squartbl.hex` with `squartb2.hex`. You will hear no interference at all because your square wave now has harmonics only at f_1 and f_3 which are canceled by the double-notch filter.

4.16. Flangers and Phasers

Flangers and phasers are similar audio effects involving variable notch filters. The flanger is implemented as an FIR comb filter with a time-variable delay. The phaser is implemented as a single-notch second-order IIR filter with time-variable notch frequency. Variants of these can easily be constructed, such as, IIR combs with variable delays, variable multi-notch filters, or even, lowpass filters with continuously variable cutoff frequency.

Flangers

The program `flanger.dsp` implements a simple flanging processor as defined by Eqs. (8.2.17) and (8.2.18) of the text [1]. The delay varies sinusoidally from 0 to 50 msec with a frequency of 4 Hz. For simplicity, we do not use a linearly interpolated delay.

```
{flanger.dsp - flanging processor}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Eqs.(8.2.18-8.2.19) of Introduction to Signal Processing.
 I/O equation: y(n) = 0.5 * x(n) + 0.5 * x(n-d(n))
 Sample processing algorithm:
       for each x do:
           *p = s0 = x
           d = (D - D * sin(2*pi*fc*t)) / 2            variable delay
           s1 = tap(D, w, p, d)
           y = a0 * s0 + a1 * s1
           cdelay(D, w, &p)
}

{--- define sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
```

```
                    {0xc85e = 9.6    | 0xc85f = 6.615                    }
{-------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.const a0 = 0x4000;                 {a0 = 0.50}
.const a1 = 0x4000;                 {a1 = 0.50}

.const D = 400;                     {TD = D/fs = 400/8000 = 50 msec}
.var/dm/circ w[D + 1];              {delay-line buffer, max delay = D}
.const D2 = D/2;

i2 = ^w;  L2 = %w;                  {delay-line buffer pointer and length}

zero(i2, m2, L2);                   {clear delay line}

.const Ds = 4000;                   {sinusoidal wavetable}
.var/dm/circ sine[Ds];              {min frequency f1 = fs/Ds = 8/4 = 2 Hz}

.init sine: <sinetbl.hex>;          {load one period of the wavetable}
                                    {generated by sinetbl.c}
.const c = 2;                       {signal frequency fc = c * f1 = 4 Hz}

i6 = ^sine; L6 = %sine;             {pointer for signal generator}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;              {wait for interrupt and loop forever}
                                    {interrupt service routine starts here}
input_samples: ena sec_reg;         {enable secondary register set}

{--- read input samples from codec ---------------------------------------}

   ax1 = dm(rx_buf + 1);            {left input sample}
   mx1 = dm(rx_buf + 2);            {right input sample}

{--- sample processing algorithm --- process right channel only ----------}

   wavgen(i6, m6, D2, c, ay1);      {ay1 = D2 * sin(2*pi*fc*t)}
   ax1 = D2;
   ar = ax1 - ay1;                  {ar = d = D2 - D2 * sin(2*pi*fc*t)}

   my1 = a0;
   mr = mx1 * my1 (ss);             {mr = a0 * x}

   m2 = ar; modify(i2, m2);         {these three lines replace the call}
   ar = -ar;                        {to tap(i2, m2, ar, my0), because}
   m2 = ar; my0 = dm(i2, m2);       {we cannot directly set m2 = -ar}

   mx0 = a1;
   mr = mr + mx0 * my0 (rnd);       {a0 * x + a * s1}
   if mv sat mr;

   tapin(i2, m2, mx1);              {put input from mx1 into tap-0}
   cdelay(i2, m2);                  {update delay}
```

```
{--- write output samples to codec ---------------------------------------}

   dm(tx_buf + 1) = mr1;           {left output sample}
   dm(tx_buf + 2) = mr1;           {right output sample}

{--- return from interrupt -----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

The time-varying delay is generated by a sinusoidal wavetable. A special feature of this implementation is that we cannot call `tap` directly to get the d-th tap because the value of d is passed through a data register and the instruction set will not allow us to negate it. Thus, we replace `tap` by its individual instructions and take the negative with the help of the ALU register `ar`.

Lab Procedure

a. Go to directory `c:\adi_dsp\examples\flanger`. Compile and run this program. Then, repeat by lowering the frequency of the varying delay to 2 Hz.

b. Repeat with increasing the maximum delay to 300 msec.

c. Repeat parts (a,b) by replacing the sinusoidal wavetable with a square wave one.

Phasers

The following program `phaser.dsp` implements a phaser as a variable notch filter:

```
{phaser.dsp - phasing effect with a variable notch filter}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Section 8.2.2 and Eq.(8.2.22) if Introd. to Signal Processing.
 Design width Df = 80 Hz, b0 = 1 / (1 + tan(Dw/2)) = 0.9695.
 Notch frequency varies sinusoidally from 200 Hz to 800 Hz:

       f0 = 500 + 300 * sin(2*pi*fsweep*t)              (in Hz)
       w0 = 0.125*pi + 0.075*pi * sin(2*pi*fsweep*t)    (in rads/sample)

 Sweep frequency fsweep = 10 Hz. Because w0 varies between
       0.20*pi < w0 < 0.05*pi
 we can use the approximation cos(w0) = 1 - w0^2 / 2.
}

{--- define sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16     }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9   }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8   }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                      }
{-------------------------------------------------------------------------}
```

```
.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}

.var/dm/circ w[3];                  {y-delay-line buffer in DM}
.var/dm/circ v[3];                  {x-delay-line buffer in DM}
.var/pm/circ a[3], b[3];            {concatenated coeffs in PM}

.const ea = 1;                      {scaling exponent for a}
.const eb = 1;                      {scaling exponent for b}
.const ex = 1;                      {input scaling exponent}

i2 = ^w; L2 = %w;                   {y-delay-line pointer}
i3 = ^v; L3 = %v;                   {x-delay-line pointer}
i4 = ^a; L4 = 6;                    {double-length a,b coefficients}

zero(i2, m2, L2);                   {clear y-delay line}
zero(i3, m3, L3);                   {clear x-delay line}

.const Ds = 4000;                   {wavetable's min freq: fs/Ds = 2 Hz}
.var/dm/circ s[Ds];                 {cosinusoidal wavwetable}

i6 = ^s; L6 = %s;                   {points to sweep frequency}

.const csweep = 5;                  {fsweep = 10 Hz}

.const w1 = 0x3244;                 {w1 = 0.125*pi => f1 = 500 Hz}
.const w2 = 0x1e28;                 {w2 = 0.075*pi => f2 = 300 Hz}

.const b0  = 0x3e0c;                {b0  = 0.9695, scaled by 2}
.const b11 = 0x83e7;                {b11 = -2*b0 = -1.9390, scaled by 2}
.const a2  = 0x3c19;                {a2  = 2*b0-1 = 0.9390, scaled by 2}
.const a0  = 0x4000;                {a0  = 1, scaled by 2}

.init s: <sinetbl.hex>;             {load wavetable}
.init a: a0, 0, a2;                 {initial denominator coefficients}
.init b: b0, 0, b0;                 {initial numerator coefficients}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;              {wait for interrupt and loop forever}
                                    {interrupt service routine starts here}
input_samples: ena sec_reg;         {enable secondary register set}

{--- read input samples from codec ---------------------------------------}

   ax1 = dm(rx_buf + 1);                   {left input sample}
   mx1 = dm(rx_buf + 2);                   {right input sample}

{--- sample processing algorithm -----------------------------------------}

   ax0 = w1;
   wavgen(i6, m6, w2, csweep, ay0);        {ay0 = w2 * sine(sweep*n)}
   ar = ax0 + ay0;                         {ar = w0 = w1 + w2*sin(wsweep*n)}

   mr= 0;                                  {calculate cos(w0)}
   mr1 = 0x7fff;                           {mr = 1}
   sr = ashift ar by -1 (hi);              {sr1 = w0 / 2}
```

```
      my0 = ar;                               {my0 = w0}
      mr = mr - sr1 * my0 (rnd);              {mr1 = cos(w0) = 1 - w0^2 / 2}

      my0 = b11;                              {current a1 and b1 coefficients}
      mr = ar * my0 (rnd);                    {mr1 = -2 * b0 * cos(w0)}

      m5 = 1;
      modify(i4, m5);                         {start from a0, point to a1}
      pm(i4, m5) = sr1;                       {save current a1, point to a2}

      m5 = 2;
      modify(i4, m5);                         {point to b1}
      pm(i4, m5) = sr1;                       {save b1, point back to a0}

      sr1 = mx1;

     {jump nofilter;}                         {uncomment to bypass filter}

      sr = ashift sr1 by -ex (hi);                           {pre-scale x down}
                                                             {compute output y}
      cdir(2, i4, m4, i2, m2, i3, m3, ea, eb, sr1);          {input from sr1}
                                                             {output in ar}
      sr = ashift ar by ex (hi);                             {post-scale y up}
      sr = ashift sr1 by 1 (hi);                       {make it a little louder}

   {--- write samples to codec -------------------------------------------}

   nofilter:

      dm(tx_buf + 1) = sr1;                    {left output sample}
      dm(tx_buf + 2) = sr1;                    {right output sample}

   {--- return from interrupt --------------------------------------------}

      rti;

   .include <c:\adi_dsp\macros\end.dsp>;      {wrapup}
```

The notch frequency varies sinusoidally between the limits of 200 Hz and 800 Hz:

$$f_0 = 500 + 300\sin(2\pi f_{\text{sweep}} t)$$

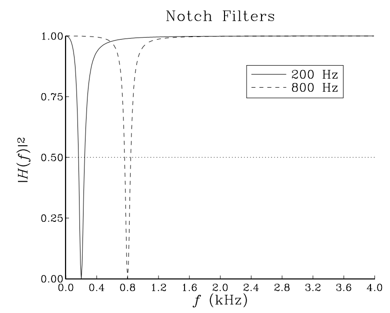
where the sweep frequency is initially chosen to be $f_{\text{sweep}} = 10$ Hz. Assuming an 8 kHz sampling rate the notch frequency in radians per sample will be:

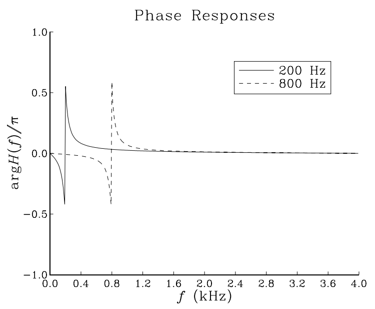
$$\omega_0 = 0.125\pi + 0.075\pi\ \sin(2\pi f_{\text{sweep}} t)$$

The filter's transfer function is given by Eq. (8.2.22) of the text [1]. The width parameter $b = 0.9695$ was calculated from Eq. (8.2.23) by assuming a 3-dB width of $\Delta f = 80$ Hz, or $\Delta\omega = 0.02\pi$. The filter's transfer function is therefore,

$$H(z) = \frac{0.9695 - 1.9390\cos\omega_0\, z^{-1} + 0.9695 z^{-2}}{1 - 1.9390\cos\omega_0\, z^{-1} + 0.9390 z^{-2}}$$

The magnitude and phase responses of the two filters corresponding to the two extreme values of the notch frequency $f_0 = 200, 800$ Hz are shown below:





As ω_0 varies, these two filters continuously "morph" into each other. In the program, the filter is implemented in its direct form to avoid overflows. Moreover, because the middle coefficients $a_1 = b_1 = -1.9390\cos\omega_0$ take on values in the 2.14 format's range, we have used that format to convert them to hex. This effectively divides the filter coefficients by 2 and must be compensated by using the scaling exponents $e_a = e_b = 1$, corresponding to the scaling factors $G_a = 2^{e_a} = 2$, $G_b = 2^{e_b} = 2$.

The varying notch frequency is generated using a sinusoidal wavetable. Because ω_0 remains small, we have calculated $\cos\omega_0$ using the approximation:

$$\cos x = 1 - \frac{1}{2}x^2$$

Lab Procedure

a. Compile and run the program `phaser.dsp`. Experiment with lower and higher values of the sweep frequency.

b. Repeat part (a) using a square wavetable.

c. Write another version of the program that calculates $\cos\omega_0$ using the improved approximation:

$$\cos x = 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4$$

Then, let the notch frequency vary over a somewhat wider range.

4.17. Simulator Examples

The first two simulator examples simulate the quantization and downsampling algorithms discussed in Section 4.1 of this manual.

Quantization Example

The program `quantex1.dsp` implements quantization to B bits. If you choose B to be 4, 8, and 12 bits, you could observe how the algorithm preserves one, two, or three of the MSB hex digits of each word, and sets the remaining digits to zero.

```
{quantex1.dsp - quantization example}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

.include <c:\adi_dsp\macros\dspmac.dsp>;         {DSP macros}

.module/ram/abs=0 decim1;

.const B = 8;                      {quantization bits per sample}
.const L = 16 - B;                 {L least-significant bits thrown away}
.const N = 8;                      {number of input samples}

.var/dm x[N];                             {input samples}
.var/dm y[N];                             {output samples}

i2 = ^x;  m2 = 1;  L2 = 0;                {input samples}
i3 = ^y;  m3 = 1;  L3 = 0;                {output samples}

.init  x: <x.hex>;                        {read input samples from file}

        jump start; nop; nop; nop;        {Interupt vector table}
        rti; nop; nop; nop;               {No interrupts used}
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;

start:

cntr=N;                                   {process N input samples}
do output until ce;
         mr1 = dm(i2, m2);                {read input sample}
         sr = ashift mr1 by -L (hi);      {shift it right by L bits}
         sr = ashift sr1 by  L (hi);      {shift it left by L bits}
output:  dm(i3, m3) = sr1;                {write output sample}

idle;

.endmod;
```

Downsampling Example

The program dnsamp1.dsp implements the downsampling operation. If you choose $M = 3$, then every third input sample will come out intact, whereas the other samples will be zero.

```
{dnsamp1.dsp - decimation/downsampling example}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

.include <c:\adi_dsp\macros\dspmac.dsp>;         {DSP macros}

.module/ram/abs=0 decim1;

.const M = 3;                             {decimation ratio}
.const N = 8;                             {number of input samples}
.const A = 0x7fff;                        {sampling pulse amplitude, A=1}
```

```
.var/dm x[N];                             {input samples}
.var/dm y[N];                             {output samples}
.var/dm/circ s[M];                        {periodic pulse train}

i2 = ^x;  m2 = 1;  L2 = 0;                {input samples}
i3 = ^y;  m3 = 1;  L3 = 0;                {output samples}
i5 = ^s;  L5 = %s; m5 = 1;                {pulse train}

.init  x: <x.hex>;                        {read input samples from file}

        jump start; nop; nop; nop;        {Interupt vector table}
        rti; nop; nop; nop;               {No interrupts used}
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;

start:

cntr=M;
do zero_s until ce;                       {initialize s to zero}
zero_s:   dm(i5, m5) = 0;

ax1 = A;                                  {set s[0] = A}
dm(s) = ax1;                              {s = [A,0,...,0] = A and M-1 zeros}

cntr=N;                                   {process N input samples}
do output until ce;
       mr1 = dm(i2, m2);                  {read input sample}

       ar = dm(i5, m5);                   {get sampling pulse}
       ar = pass ar;                      {forces updating of AZ flag}
       if ne jump output;                 {if AR != 0, write output}

       {else}  mr1 = 0;                   {if AR = 0, output is 0}

       output: dm(i3, m3) = mr1;          {write output sample}

idle;

.endmod;
```

Delay Examples

The next two simulator examples illustrate linear and circular delay-line buffers and are based on Example 4.2.1 of the text [1]. The input and 3-fold delayed output signals are:

$$\mathbf{x} = \frac{1}{4}[1, 1, 2, 1, 2, 2, 1, 1]$$

$$\mathbf{y} = \frac{1}{4}[0, 0, 0, 1, 1, 2, 1, 2, 2, 1, 1]$$

They have been scaled by 4 to fit into the 1.15 format. The corresponding 1.15 hex input samples are:

```
x = [0x2000, 0x2000, 0x4000, 0x2000, 0x4000, 0x4000, 0x2000, 0x2000]
```

Linear Buffer

The program `delex1.dsp` implements the 3-fold delay using a linear buffer:

```
{delex1.dsp - delay example using linear buffer}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Example 4.2.1 of Introduction to Signal Processing}

.include <c:\adi_dsp\macros\dspmac.dsp>;         {DSP macros}

.module/ram/abs=0  delex1;

.var/dm  x[8];                         {input samples}
.var/dm  y[11];                        {output samples}
.var/dm  w[4];                         {linear delay-line buffer in DM}

.init  x: <xfir.hex>;
.init  w: 0x0000, 0x0000, 0x0000, 0x0000;

        jump start; nop; nop; nop;               {Interupt vector table}
        rti; nop; nop; nop;                      {No interrupts used}
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;

start:  i3 = ^x;  m3 = 1;  L3 = 0;        {input samples}
        i5 = ^y;  m5 = 1;  L5 = 0;        {output samples}

       cntr = 8;
       do outputs until ce;               {process 8 input samples}
              my1 = dm(w+3);              {get tap-3}
              dm(i5, m5) = my1;           {save in y and point to next y}

              mx1 = dm(i3, m3);           {read x and point to next x}
              dm(w) = mx1;                {put x in delay line}
                                          {update linear buffer}
              my0 = dm(w+2);              {w[2] = tap-2}
              dm(w+3) = my0;              {w[3] = w[2]}

              my0 = dm(w+1);              {w[1] = tap-1}
              dm(w+2) = my0;              {w[2] = w[1]}

              my0 = dm(w);                {w[0] = tap-0}
outputs:      dm(w+1) = my0;              {w[1] = w[0]}

       cntr = 3;                          {three input-off transients}
       do transients until ce;
              my1 = dm(w+3);              {get tap-3}
              dm(i5, m5) = my1;           {save in y and point to next y}

              mx0 = 0;
              dm(w) = mx0;                {put 0 in delay line}
```

```
                                          {update linear buffer}
              my0 = dm(w+2);              {w[2] = tap-2}
              dm(w+3) = my0;              {w[3] = w[2]}

              my0 = dm(w+1);              {w[1] = tap-1}
              dm(w+2) = my0;              {w[2] = w[1]}

              my0 = dm(w);                {w[0] = tap-0}
transients:   dm(w+1) = my0;              {w[1] = w[0]}

       idle;

.endmod;
```

Circular Buffer

The program `delex2.dsp` implements the 3-fold delay using a circular buffer:

```
{delex2.dsp - delay example using circular buffer}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Example 4.2.1 of Introduction to Signal Processing}

.include <c:\adi_dsp\macros\dspmac.dsp>;         {DSP macros}

.module/ram/abs=0  delex1;

.var/dm      x[8];                     {input samples}
.var/dm      y[11];                    {output samples}
.var/dm/circ w[4];                     {circular delay-line buffer in DM}

.init  x: <xfir.hex>;
.init  w: 0x0000, 0x0000, 0x0000, 0x0000;

        jump start; nop; nop; nop;               {Interupt vector table}
        rti; nop; nop; nop;                      {No interrupts used}
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;

start:  i2 = ^w;  L2 = %w;                {circular buffer}
        i3 = ^x;  m3 = 1;  L3 = 0;        {input samples}
        i5 = ^y;  m5 = 1;  L5 = 0;        {output samples}

       cntr = 8;
       do outputs until ce;               {process 8 input samples}
              m2 = 3; modify(i2, m2);     {point to tap-3}
              m2 = 0; my1 = dm(i2, m2);   {get tap-3}
              m2 =-3; modify(i2, m2);     {point back to tap-0}

              dm(i5, m5) = my1;           {write y}

              mx1 = dm(i3, m3);           {get input x}
              m2 = 0; dm(i2, m2) = mx1;   {put it into tap-0}
```

```
                m2 = -1;
outputs:        modify(i2, m2);             {backshift buffer pointer}

        cntr = 3;                           {3 input-off transients}
        do transients until ce;
                m2 = 3; modify(i2, m2);     {point to tap-3}
                m2 = 0; my1 = dm(i2, m2);   {get tap-3}
                m2 =-3; modify(i2, m2);     {point back to tap-0}

                dm(i5, m5) = my1;           {write y}

                mx1 = 0;                    {zero input x}
                m2 = 0; dm(i2, m2) = mx1;   {put it into tap-0}

                m2 = -1;
transients:     modify(i2, m2);             {backshift buffer pointer}

        idle;

.endmod;
```

FIR Filter Examples

The next two examples illustrate FIR filtering and are based on Example 4.2.1 of the text [1]. The input signal and filter are:

$$\mathbf{h} = \frac{1}{4}[1,2,-1,1], \qquad \mathbf{x} = \frac{1}{4}[1,1,2,1,2,2,1,1]$$

The output is the convolution:

$$\mathbf{y} = \mathbf{h} * \mathbf{x} = \frac{1}{16}[1,3,3,5,3,7,4,3,3,0,1]$$

In 1.15 format, the filter, input, and output signals are:

```
h = [0x2000, 0x4000, 0xe000, 0x2000]
x = [0x2000, 0x2000, 0x4000, 0x2000, 0x4000, 0x4000, 0x2000, 0x2000]
y = [0x0800, 0x1800, 0x1800, 0x2800, 0x1800, 0x3800, 0x2000, 0x1800,
     0x1800, 0x0000, 0x0800]
```

The filter and input hex numbers can also be obtained by converting the unscaled signals in the 3.13 format. The output samples can be converted using `hex2dec` into their unscaled decimal versions using the 5.11 format to compensate for the factor 16.

The program `firex1.dsp` implements the FIR filter using the macros `tapin`, `dot`, and `cdelay`:

```
{firex1.dsp - FIR example using tapin, dot, cdelay}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Example 4.2.1 of Introduction to Signal Processing. The filter
 coefficients and are stored in the files h.dec/h.hex. The input and
 output signals are in the files xfir.dec/xfir.hex, yfir.dec/yfir.hex.
```

```
 h = 0.25 =  1/4      x = 0.25 = 1/4      y = 0.0626 = 1/16
     0.50 =  2/4          0.25 = 1/4          0.1875 = 3/16
    -0.25 = -1/4          0.50 = 2/4          0.1875 = 3/16
     0.25 =  1/4          0.25 = 1/4          0.3125 = 5/16
                          0.50 = 2/4          0.1875 = 3/16
                          0.50 = 2/4          0.4375 = 7/16
                          0.25 = 1/4          0.2500 = 4/16
                          0.25 = 1/4          0.1875 = 3/16
                                              0.1875 = 3/16
                                              0.0000 = 0/16
                                              0.0625 = 1/16

 (The h-coefficients and x have been scaled down to fit in 1.15 format,
 as compared to Example 4.2.1.)
}

.include <c:\adi_dsp\macros\dspmac.dsp>;        {DSP macros}

.module/ram/abs=0  firex1;

.const             L=8;                         {input signal length}
.const             M=3;                         {filter order}
.var/dm/ram        x[L];                        {input samples}
.var/dm/ram        y[L+M];                      {output samples}
.var/dm/ram/circ   w[M+1];                      {delay-line buffer in DM}
.var/pm/ram/circ   h[M+1];                      {filter taps in PM}

.init  x: <xfir.hex>;                           {read L input samples}
.init  h: <h.hex>;                              {read M+1 filter taps}

        jump start; nop; nop; nop;              {Interupt vector table}
        rti; nop; nop; nop;                     {No interrupts used}
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;

start:  i2 = ^w;  m2 = 0;  L2 = %w;             {delay line buffer}
        i4 = ^h;  m4 = 0;  L4 = %h;             {filter taps buffer}
        i3 = ^x;  m3 = 1;  L3 = 0;              {input samples}
        i5 = ^y;  m5 = 1;  L5 = 0;              {output samples}

       zero(i2, m2, L2);                        {clear delay line}

       cntr = L;
       do outputs until ce;                     {process L input samples}
              ax0 = dm(i3, m3);                 {read x into AX0}
              tapin(i2, m2, ax0);               {put x in delay line}
              dot(M, i4, m4, i2, m2);           {compute output in MR1}
              cdelay(i2, m2);                   {update delay}
outputs:      dm(i5, m5) = mr1;                 {output y}

       cntr = M;                                {M input-off transients}
       do transients until ce;
              ax0 = 0;                          {read x=0 into AX0}
              tapin(i2, m2, ax0);               {put it in delay line}
              dot(M, i4, m4, i2, m2);           {compute output in MR1}
              cdelay(i2, m2);                     {update delay}
transients:   dm(i5, m5) = mr1;                   {output y}

       idle;

.endmod;
```

The program `firex2.dsp` implements the FIR filter using the macro `cfir`:

```
{firex2.dsp - FIR example using cfir}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Example 4.2.1 of Introduction to Signal Processing. The filter
 coefficients and are stored in the files h.dec/h.hex. The input and
 output signals are in the files xfir.dec/xfir.hex, yfir.dec/yfir.hex.

     h = 0.25 =  1/4      x = 0.25 = 1/4      y = 0.0626 = 1/16
         0.50 =  2/4          0.25 = 1/4          0.1875 = 3/16
        -0.25 = -1/4          0.50 = 2/4          0.1875 = 3/16
         0.25 =  1/4          0.25 = 1/4          0.3125 = 5/16
                              0.50 = 2/4          0.1875 = 3/16
                              0.50 = 2/4          0.4375 = 7/16
                              0.25 = 1/4          0.2500 = 4/16
                              0.25 = 1/4          0.1875 = 3/16
                                                  0.1875 = 3/16
                                                  0.0000 = 0/16
                                                  0.0625 = 1/16

 (The h-coefficients and x have been scaled down to fit in 1.15 format,
 as compared to Example 4.2.1.)
}

.include <c:\adi_dsp\macros\dspmac.dsp>;          {dsp macros}

.module/ram/abs=0  firex2;

.const            L=8;                            {input signal length}
.const            M=3;                            {filter order}
.var/dm/ram       x[L];                           {input samples}
.var/dm/ram       y[L+M];                         {output samples}
.var/dm/ram/circ  w[M+1];                         {delay-line buffer in DM}
.var/pm/ram/circ  h[M+1];                         {filter taps in PM}

.init  x: <xfir.hex>;                             {read L input samples}
.init  h: <h.hex>;                                {read M+1 filter taps}

        jump start; nop; nop; nop;                {Interupt vector table}
        rti; nop; nop; nop;                       {No interrupts used}
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;

start:  i2 = ^w;  L2 = %w;                        {delay line buffer}
        i4 = ^h;  L4 = %h;                        {filter taps buffer}
        i3 = ^x;  m3 = 1;   L3 = 0;               {input samples}
        i5 = ^y;  m5 = 1;   L5 = 0;               {output samples}
```

```
       zero(i2, m2, L2);                          {clear delay line}

       cntr = L;
       do outputs until ce;                       {process L input samples}
              ax0 = dm(i3, m3);                   {read x into AX0}
              cfir(M, i4, m4, i2, m2, ax0);       {output in MR1}
outputs:      dm(i5, m5) = mr1;                   {output y}

       cntr = M;                                  {input-off transients}
       do transients until ce;
              ax0 = 0;                            {read x=0 into AX0}
              cfir(M, i4, m4, i2, m2, ax0);       {output in MR1}
transients:   dm(i5, m5) = mr1;                   {output y}

        idle;

.endmod;
```

IIR Filter Examples

The next two programs illustrate IIR filtering based on Example 7.5.4 of the text [1]. The transfer function is taken to be:

$$H(z) = \frac{0.25 + 0.25z^{-1} - 0.50z^{-2}}{1 - z^{-3}}$$

where the b-coefficients have been scaled down by 4 to fit into the 1.15 format. Similarly, the input has been scaled down by 8:

$$\mathbf{x} = \frac{1}{8}[1, 3, 2, 5, 4, 6, 0, 0, 0]$$

The expected output from that example was worked out in the text:

$$\mathbf{y} = \frac{1}{32}[1, 4, 7, 14, 17, 27, 28, 29, 27]$$

The program `canex1.dsp` implements an IIR filter using the canonical realization macro `ccan`:

```
{canex1.dsp - canonical realization example using ccan}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Example 7.5.4 of Introduction to Signal Processing. The filter
 coefficients are:

       a = [1, 0, 0, -1]
       b = [1, 1, 2,  0] / 4 = [0.25, 0.25, 0.50, 0]

 The coefficients define a double-length concatenated circular buffer i7,
 and are stored in the files a.dec/a.hex and b.dec/b.hex. The input and
 output signals are in the files xiir.dec/xiir.hex, yiir.dec/yiir.hex:

       x = 0.125 = 0x1000 = 1/8     y = 0.03125 = 0x0400 =  1/32
           0.375 = 0x3000 = 3/8         0.12500 = 0x1000 =  4/32
```

```
              0.250 = 0x2000 = 2/8        0.21875 = 0x1c00 =  7/32
              0.625 = 0x5000 = 5/8        0.43750 = 0x3800 = 14/32
              0.500 = 0x4000 = 4/8        0.53125 = 0x4400 = 17/32
              0.750 = 0x6000 = 6/8        0.84375 = 0x6c00 = 27/32
              0.000 = 0x0000 = 0/8        0.87500 = 0x7000 = 28/32
              0.000 = 0x0000 = 0/8        0.90625 = 0x7400 = 29/32
              0.000 = 0x0000 = 0/8        0.84375 = 0x6c00 = 27/32

 (The b-coefficients and x have been scaled down to fit in 1.15 format,
 as compared to Example 7.5.4.)

 Scaling factors:

    The a,b coefficients that are used in the program and read from
    the file ab.hex are the related to the true a,b coefficients by
    the following scale factors whoes exponents are passed to ccan():

       a_used = 2^(-ea) * a_true
       b_used = 2^(-eb) * b_true

    In this example: ea = 0, eb = 0, (no further scaling is needed)

    The input samples x must also, in general, be scaled down before
    passed into ccan() and then the output from ccan() must be scaled up:

       x1 = 2^(-ex) * x     input to ccan()
       y1 = ccan(x1)        output from ccan() due to x1
       y = 2^ex * y1        output due to x

    In this example, we must use ex = 1 (or larger).
    The unscaled input (ex = 0) causes overflows and the wrong output.
}

.include <c:\adi_dsp\macros\dspmac.dsp>;         {dsp macros}

.module/ram/abs=0  canex1;

.const        L=9;                        {input signal length}
.const        M=3;                        {filter order}
.var/dm       x[L];                       {input samples}
.var/dm       y[L];                       {output samples}
.var/dm/circ  w[M+1];                     {delay-line buffer in DM}
.var/pm/circ  a[M+1], b[M+1];             {concatenated coeffs in PM}

.init  x: <xiir.hex>;                     {input samples}
.init  a: <a.hex>;                        {denominator coefficients}
.init  b: <b.hex>;                        {numerator coefficients}

.const ea = 0;                            {scaling exponent for a}
.const eb = 0;                            {scaling exponent for b}
.const ex = 1;                            {pre/post scaling exponent}

        jump start; nop; nop; nop;               {Interupt vector table}
        rti; nop; nop; nop;                      {No interrupts used}
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
```

```
        rti; nop; nop; nop;

start:  i2 = ^w;  L2 = %w;                           {delay-line buffer}
        i4 = ^x;  m4 = 1;  L4 = 0;                   {input samples}
        i5 = ^y;  m5 = 1;  L5 = 0;                   {output samples}
        i7 = ^a;           L7 = 2*(M+1);             {double-length a,b coef-}
                                                     {-ficient buffer}
       zero(i2, m2, L2);                             {clear delay line}

       cntr = L;
       do outputs until ce;
              sr1 = dm(i4, m4);                             {read input x}
              sr = ashift sr1 by -ex (hi);                  {pre-scale x down}
                                                            {compute output y}
              ccan(M, i7, m7, i2, m2, ea, eb, sr1);         {input from sr1}
                                                            {output in sr1}
              sr = ashift sr1 by ex (hi);                   {post-scale y up}
outputs:      dm(i5, m5) = sr1;                             {write y}

        idle;

.endmod;
```

Further down-scaling of the input and up-scaling of the output are necessary to avoid overflows. The input scaling exponent was $e_x = 1$. The program `direx1.dsp` implements an IIR filter using the direct-form realization macro `cdir`. It does not need any input down-scaling and therefore $e_x = 0$.

```
{direx1.dsp - direct-form-I realization example using cdir}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on Example 7.5.4 of Introduction to Signal Processing. The filter
 coefficients are:

       a = [1, 0, 0, -1]
       b = [1, 1, 2,  0] / 4 = [0.25, 0.25, 0.50, 0]

 The coefficients define a double-length concatenated circular buffer i7,
 and are stored in the files a.dec/a.hex and b.dec/b.hex. The input and
 output signals are in the files xiir.dec/xiir.hex, yiir.dec/yiir.hex:

       x = 0.125 = 0x1000 = 1/8    y = 0.03125 = 0x0400 =  1/32
           0.375 = 0x3000 = 3/8        0.12500 = 0x1000 =  4/32
           0.250 = 0x2000 = 2/8        0.21875 = 0x1c00 =  7/32
           0.625 = 0x5000 = 5/8        0.43750 = 0x3800 = 14/32
           0.500 = 0x4000 = 4/8        0.53125 = 0x4400 = 17/32
           0.750 = 0x6000 = 6/8        0.84375 = 0x6c00 = 27/32
           0.000 = 0x0000 = 0/8        0.87500 = 0x7000 = 28/32
           0.000 = 0x0000 = 0/8        0.90625 = 0x7400 = 29/32
           0.000 = 0x0000 = 0/8        0.84375 = 0x6c00 = 27/32

 (The b-coefficients and x have been scaled down to fit in 1.15 format,
 as compared to Example 7.5.4.)

 Scaling factors:

    The a,b coefficients that are used in the program and read from
    the file ab.hex are the related to the true a,b coefficients by
    the following scale factors whoes exponents are passed to cdir():

       a_used = 2^(-ea) * a_true
       b_used = 2^(-eb) * b_true

    In this example: ea = 0, eb = 0, (no further scaling is needed)

    The input samples x must also, in general, be scaled down before
    passed into cdir() and then the output from cdir() must be scaled up:

       x1 = 2^(-ex) * x     input to cdir()
       y1 = cdir(x1)        output from cdir() due to x1
       y = 2^ex * y1        output due to x

    In this example (unlike canex1), we use ex = 0 (or larger).
}

.include <c:\adi_dsp\macros\dspmac.dsp>;         {dsp macros}

.module/ram/abs=0  direx1;

.const        L=9;                        {input signal length}
.const        M=3;                        {filter order}
.var/dm       x[L];                       {input samples}
.var/dm       y[L];                       {output samples}
.var/dm/circ  w[M+1];                     {y-delay-line buffer in DM}
.var/dm/circ  v[M+1];                     {x-delay-line buffer in DM}
.var/pm/circ  a[M+1], b[M+1];             {concatenated coeffs in PM}

.init  x: <xiir.hex>;                     {input samples}
.init  a: <a.hex>;                        {denominator coefficients}
.init  b: <b.hex>;                        {numerator coefficients}

.const ea = 0;                            {scaling exponent for a}
.const eb = 0;                            {scaling exponent for b}
.const ex = 0;                            {pre/post scaling exponent}

        jump start; nop; nop; nop;               {Interupt vector table}
        rti; nop; nop; nop;                      {No interrupts used}
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;
        rti; nop; nop; nop;

start:  i2 = ^w;  L2 = %w;                       {y-delay-line buffer}
        i3 = ^v;  L3 = %v;                       {x-delay-line buffer}
        i4 = ^x;  m4 = 1;  L4 = 0;               {input samples}
        i5 = ^y;  m5 = 1;  L5 = 0;               {output samples}
        i7 = ^a;           L7 = 2*(M+1);         {double-length a,b coef-}
                                                 {-ficient buffer}
       zero(i2, m2, L2);                         {clear y-delay line}
       zero(i3, m3, L3);                         {clear x-delay line}

       cntr = L;
       do outputs until ce;
          sr1 = dm(i4, m4);                             {read input x}
```

```
          sr = ashift sr1 by -ex (hi);                  {pre-scale x down}
                                                        {compute output y}
          cdir(M, i7, m7, i2, m2, i3, m3, ea, eb, sr1);    {input in sr1}
                                                        {output in ar}
          sr = ashift ar by ex (hi);                    {post-scale y up}
outputs:  dm(i5, m5) = sr1;                             {write y}

        idle;

.endmod;
```

Lab Procedure

Go to directory `c:\adi_dsp\examples\sim`. Compile and load each program into the simulator by running the batch file `ezs.bat`, that is,

```
ezs quantex1
ezs dnsamp1
ezs delex1
ezs delex2
ezs firex1
ezs firex2
ezs canex1
ezs direx1
```

Three predefined data windows will open. Using the `<CTRL-G>` command, set the windows to display the contents of the arrays `y`, `x`, and `w`. The windows can be toggled from hex to decimal format using `<CTRL-T>`.

Step through the programs by issuing the command `step` in the command line window. Observe and write down the contents of the delay-line buffer `w`, and the input and output arrays `x`, `y`, as each instruction is executed and as each input sample gets processed.

5. Macros

In this section we give the source code of the following macros used in the experiments:

```
zero.dsp
tap.dsp
tapin.dsp
cdelay.dsp
dot.dsp
cfir.dsp
ccan.dsp
cdir.dsp
wavgen.dsp
```

The files are in the directory `c:\adi_dsp\macros`.

5.1. zero

The macro `zero` initializes a circular delay-line buffer to zero:

```
{zero.dsp - initialize delay line buffer to zero.
 Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996.

       %0 = pointer to delay-line buffer, e.g.,  I2
       %1 = M-register to use with buffer, e.g., M2
       %2 = length of buffer, e.g., L2

       typical usage:
       --------------
       zero(i2, m2, L2);           i2 cycles back to its initial value

       internal operation:
       -------------------
       cntr = L2; m2 = 1;
              do loop until ce;
       loop:         dm(i2, m2) = 0;
}

.macro zero(%0, %1, %2);
.local loop;

cntr = %2;  %1 = 1;
       do loop until ce;
loop:         dm(%0, %1) = 0;

.endmacro;
```

5.2. tap

The macro `tap` allows the accessing of the tap outputs of a delay line and is modeled after the routines `tap.c` and `tap2.c` of the text [1]:

```
{tap.dsp - tap outputs of circular delay line.
 Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996.

 Based on tap.c and tap2.c of Introduction to Signal Processing.

       %0 = pointer to delay-line buffer,  e.g., i2
       %1 = M-register to use with buffer, e.g., m2
       %2 = d, for d-th tap content, where d=1, ... ,D
       %3 = data register for result, e.g.,
            ax0, ax1, ay0, ay1, ar, mx0, mx1, my0, my1, mr1, sr1

       typical usage:
       --------------
       tap(i2, m2, d, sr1);              put d-th tap content into SR1
                                         note: i2 is not changed
       internal operation:
       -------------------
       m2 = d;   modify(i2, m2);         point to d-th tap
       m2 =-d;   sr1 = dm(i2, m2);       put d-th tap in data register and
                                         restore i2 to its entry value
}
```

```
.macro tap(%0, %1, %2, %3);

%1 =  %2;  modify(%0, %1);                {point to d-th tap}
%1 = -%2;  %3 = dm(%0, %1);               {put d-th tap in data register}

.endmacro;
```

5.3. tapin

The macro `tapin` transfers the content of a data register into tap-0 of a delay line. The analogous statement in the text is $*p = x$, which puts x into the buffer location pointed to by the pointer p. Here, the role of the pointer is played by the I-register:

```
{tapin.dsp - put input sample into tap-0 of delay line.
 Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996.

       %0 = pointer to delay-line buffer, e.g.,  I2
       %1 = M-register to use with buffer, e.g., M2
       %2 = data register holding input, e.g.,
            ax0, ax1, ay0, ay1, ar, mx0, mx1, my0, my1, mr1, sr0, sr1

       typical usage:
       --------------
       tapin(i2, m2, mx1);               put value from MX1 into 0-th tap
                                         note: i2 is not changed
       internal operation:
       -------------------
       m2 = 0;   dm(i2, m2) = mx1;
}

.macro tapin(%0, %1, %2);

%1 = 0;  dm(%0, %1) = %2;          {put value from dreg %2 into delay line}

.endmacro;
```

5.4. cdelay

The macro `cdelay` is the assembly code equivalent of the routines `cdelay.c` and `cdelay2.c` of the text and its effect is to decrement the circular pointer pointing into the delay-line buffer:

```
{cdelay.dsp - update circular delay-line buffer.
 Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996.

 Based on cdelay.c and cdelay2.c of Introduction to Signal Processing.

       %0 = pointer to delay-line buffer,  e.g., i2
       %1 = m-register to use with buffer, e.g., m2

       typical usage:
       --------------
       cdelay(i2, m2);
```

```
        internal operation:
        -------------------
        m2 = -1;  modify(i2, m2);    (i.e., backshift pointer i2)
}

.macro cdelay(%0, %1);

%1 = -1;  modify(%0, %1);             {backshift pointer}

.endmacro;
```

5.5. dot

The macro `dot` employs multifunction instructions to compute the dot product of two circular buffers, one residing in DM and the other in PM. It is modeled after the FIR filtering routine `fir.dsp` found in [3].

The dot-product result is returned in `mr1`. The I-register circular pointers do not change—they cycle completely around to their entry values:

```
{dot.dsp - dot product of a DM with a PM circular buffer of length M+1.
 Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996.

        %0 = filter order M, i.e., length L = M+1
        %1 = pointer to filter taps buffer in PM, e.g., i4 (not modified)
        %2 = m-register to use with tap buffer, e.g., m4
        %3 = pointer to delay-line buffer in DM, e.g., i2 (not modified)
        %4 = m-register to use with delay buffer, e.g., m2

        result is returned in MR1;
        i2, i4 are not modified - they cycle around to their entry values

        typical usage:
        --------------
        dot(M, i4, m4, i2, m2);

        internal operation:
        -------------------
        m2 = 1; m4 = 1;
        mr = 0, mx0 = dm(i2, m2), my0 = pm(i4, m4);
        cntr = M;
        do loop until ce
loop:      mr = mr + mx0 * my0 (ss), mx0 = dm(i2, m2), my0 = pm(i4, m4);
        mr = mr + mx0 * my0 (rnd);
        if mv sat mr;
}

.macro dot(%0, %1, %2, %3, %4);
.local loop;

        %2 = 1; %4 = 1;
        mr = 0, mx0 = dm(%3, %4), my0 = pm(%1, %2);
        cntr = %0;
        do loop until ce;
loop:      mr = mr + mx0 * my0 (ss), mx0 = dm(%3, %4), my0 = pm(%1, %2);
        mr = mr + mx0 * my0 (rnd);
```

```
        if mv sat mr;

.endmacro;
```

When `dot` is used in the implementation of IIR filters, the DAG pointer `i4` points to a double-length circular buffer in PM consisting of the concatenation of the denominator and numerator coefficients. In that case, it takes two calls of `dot` for `i4` to wrap around completely. See the macros `ccan` and `cdir`.

5.6. cfir

The macro `cfir` is the assembly code equivalent of the routines `cfir.c` and `cfir2.c` of the text. It takes its input from a data register and puts it in tap-0 of the delay line, then it computes the dot-product output of the filter and returns it into `mr1`, and finally it updates the delay line:

```
{cfir.dsp - direct-form FIR filter of order M using circular buffers.
 Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996.

 Based on cfir.c and cfir2.c of Introduction to Signal Processing.
 In book: y = cfir(M, h, w, &p, x);

        %0 = filter order M, so that filter length is L = M+1
        %1 = pointer to filter taps buffer in PM, e.g., i4
        %2 = m-register to use with tap buffer,   e.g., m4
        %3 = pointer to delay-line buffer in DM,  e.g., i2
        %4 = m-register to use with delay buffer, e.g., m2
        %5 = data register holding input, e.g.,
             ax0, ax1, ay0, ay1, ar, mx0, mx1, my0, my1, mr1, sr0, sr1

        The filter output is returned in MR1.
        Upon exit, the delay-line pointer i2 is decremented.

        typical usage:
        --------------
        cfir(M, i4, m4, i2, m2, mx1);

        internal operation:
        -------------------
        tapin(i2, m2, mx1);         put input from MX1 into tap-0
        dot(M, i4, m4, i2, m2);     compute dot-product output
        cdelay(i2, m2);             update delay line
}

.macro cfir(%0, %1, %2, %3, %4, %5);

        tapin(%3, %4, %5);          {read input sample into delay line}
        dot(%0, %1, %2, %3, %4);    {compute filter output into MR1}
        cdelay(%3, %4);             {update delay line}

.endmacro;
```

5.7. ccan

The macro `ccan` implements the canonical realization of an IIR filter and is the assembly code equivalent of the routines `ccan.c` and `can2.c` of the text. It takes its input from a data register, returns its output into `sr1`, and updates the delay line:

```
{ccan.dsp - canonical-form IIR filter of order M using circular buffers.
 Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996.

 Based on ccan.c and can2.c of Introduction to Signal Processing.

       %0 = filter order M
       %1 = pointer to filter taps buffer in PM, e.g., i4
       %2 = m-register to use with tap buffer,   e.g., m4
       %3 = pointer to delay-line buffer in DM,  e.g., i2
       %4 = m-register to use with delay buffer, e.g., m2
       %5 = exponent for a-coefficient scale factor
       %6 = exponent for b-coefficient scale factor
       %7 = data register holding input, e.g.,
            ax0, ax1, ay0, ay1, ar, mx0, mx1, my0, my1, mr1, sr0, sr1

       The filter output is returned in SR1.
       The input may need to be scaled down further to avoid overflows.
       Upon exit, the delay-line pointer i2 is decremented.

       The filter coefficients must be stored consecutively in the order:

              [a0, a1, a2,..., aM, b0, b1,..., bM]

       and i4 is points to this double-length buffer. The a,b coefficients
       used in the program are related to the true a,b coefficients by the
       scale factors, defined by the exponents ea, eb:

              a = a_true / Ga,     Ga = 2^ea = scale factor
              b = b_true / Gb,     Gb = 2^eb = scale factor

       (because a0_true = 1, it follows that a0 = 1/Ga. This
       coefficient is redundant and not really used in the computation; it
       always gets multiplied by zero.)

       The common double-length circular buffer i4 should be declared as:

              .var/pm/circ a[M+1], b[M+1];
              i4 = ^a; L4 = 2*(M+1);

       The program assumes that numerator and denominator have order M.
       The delay-line buffer must be declared as follows:

              .var/dm/circ w[M+1];
              i2 = ^w; L2 = %w;

       typical usage:
       --------------
       ccan(M, i4, m4, i2, m2, ea, eb, ax1);

       internal operation:
```

```
       -------------------
       mx1 = ax1;
       mr1 = 0;
       tapin(i2, m2, mr1);
       dot(M, i4, m4, i2, m2);
       sr = ashift mr1 by ea (hi);
       mr = 0;
       mr1 = sr1;
       my0 = 0x8000;
       mr = mr + mx1 * my0 (rnd);
       if mv sat mr;
       ar = -mr1;
       tapin(i2, m2, ar);
       dot(M, i4, m4, i2, m2);
       sr = ashift mr1 by eb (hi);
       cdelay(i2, m2);
}

.macro ccan(%0, %1, %2, %3, %4, %5, %6, %7);

       mx1 = %7;                            {mx1 = input x}
       mr1 = 0;
       tapin(%3, %4, mr1);                  {put s0 = 0 into tap-0}
       dot(%0, %1, %2, %3, %4);             {mr1 = dot(M,a,s)}
       sr = ashift mr1 by %5 (hi);          {sr1 = Ga * dot(M,a,s)}
       mr = 0;
       mr1 = sr1;                           {mr = Ga * dot(M,a,s)}
       my0 = 0x8000;                        {facilitates MAC usage}
       mr = mr + mx1 * my0 (rnd);           {mr = Ga * dot(M,a,s) - x}
       if mv sat mr;
       ar = -mr1;                           {ar = s0 = x - Ga * dot(M,a,s)}
       tapin(%3, %4, ar);                   {input s0 into tap-0}
       dot(%0, %1, %2, %3, %4);             {mr1 = dot(M,b,s)}
       sr = ashift mr1 by %6 (hi);          {sr1 = y = Gb * dot(M,b,s)}
       cdelay(%3, %4);

.endmacro;
```

Fig. 5.1 shows this realization together with the necessary scale factors. To fit within the 1.15 format, the filter coefficients must be scaled by appropriate factors of the form:

$$G_a = 2^{e_a}, \quad G_b = 2^{e_b}$$

The scaled and unscaled coefficients are related by

$$\mathbf{a}' = [a'_0, a'_1, \dots, a'_M] = \frac{1}{G_a}[1,\ a_1, \dots, a_M] = \frac{1}{G_a}\mathbf{a}$$

$$\mathbf{b}' = [b'_0, b'_1, \dots, b'_M] = \frac{1}{G_b}[b_0, b_1, \dots, b_M] = \frac{1}{G_b}\mathbf{b}$$

where $\mathbf{a}'$ are the scaled coefficients that will be loaded in DM, and $\mathbf{a}$ are the true coefficients. (Note that a'_0 is no longer unity, but that does not matter because it is not really used in the computation.) The overall transfer function is not affected by these scale factors; only the intermediate calculations.

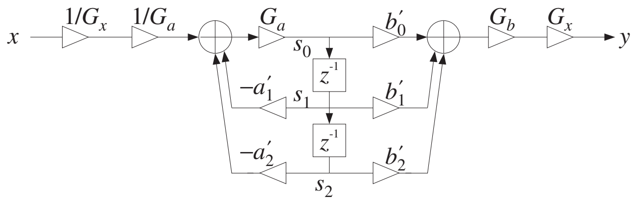


Fig. 5.1 Canonical form with scaling factors.

To be able to use the dot-product macro dot, we write the sample processing algorithm for the true coefficients in the form of Eq. (7.2.6) of the text, which we have modified here to use a circular delay-line buffer $\mathbf{w}$ with state vector $\mathbf{s}$ defined relative to the buffer pointer p:

```
for each input sample x do:
    $*p = s_0 = 0$
    $*p = s_0 = x - \mathrm{dot}(M, \mathbf{a}, \mathbf{s})$
    $y = \mathrm{dot}(M, \mathbf{b}, \mathbf{s})$
    $\mathrm{cdelay}(M, \mathbf{w}, \&p)$
```

Writing the unscaled coefficients in terms of the scaled ones, $\mathbf{a} = G_a\mathbf{a}'$ and $\mathbf{b} = G_b\mathbf{b}'$, we obtain the sample processing algorithm depicted in the above block diagram:

```
for each input sample x do:
    $*p = s_0 = 0$
    $*p = s_0 = x - G_a\mathrm{dot}(M, \mathbf{a}', \mathbf{s})$
    $y = G_b\mathrm{dot}(M, \mathbf{b}', \mathbf{s})$
    $\mathrm{cdelay}(M, \mathbf{w}, \&p)$
```

The canonical realization is more prone to overflows than the direct or transposed forms. In such cases the input x must also be scaled down by an appropriate scale factor of the form $G_x = 2^{e_x}$. After filtering, the output may be scaled back up by the same factor as shown in the block diagram. The input scaling is not implemented in the above sample processing algorithm. If necessary, it may be done outside ccan using the shifter.

In the main program, the denominator and numerator coefficients **a** and **b** must be concatenated into a double-length circular buffer. If the DAG pointer i4 is set to point to the beginning of **a** and i2 to the state vector **s**, then after the first call of dot, i2 will wrap around completely, and i4 will be left pointing to the beginning of **b**. Only after the second call of dot will the pointer i4 wrap around to the beginning of **a**, while i2 will wrap around again. Thus, upon exit from ccan, i4 will remain unchanged, whereas i2 will be backshifted due to the last call of cdelay.

5.8. cdir

The macro cdir implements the direct-form-I realization and is the assembly code equivalent of the routines dir.c and dir2.c of the text. It takes its input from a data register, returns its output into ar, and updates the delay lines:

```
{cdir.dsp - direct-form-I IIR filter of order M using circular buffers.
 Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996.

 Based on dir.c and dir2.c of Introduction to Signal Processing.

      %0 = filter order M
      %1 = pointer to filter taps buffer in PM, e.g., i4
      %2 = m-register to use with tap buffer,   e.g., m4
      %3 = pointer to y-delay-line buffer in DM,  e.g., i2
      %4 = m-register to use with y-delay buffer, e.g., m2
      %5 = pointer to x-delay-line buffer in DM,  e.g., i3
      %6 = m-register to use with x-delay buffer, e.g., m3
      %7 = exponent for a-coefficient scale factor
      %8 = exponent for b-coefficient scale factor
      %9 = data register holding input, e.g.,
           ax0, ax1, ay0, ay1, ar, mx0, mx1, my0, my1, mr1, sr0, sr1

      The filter output is returned in AR.
      The input may need to be scaled down further to avoid overflows.
      Upon exit, the delay-line pointer i2 is decremented.

      The filter coefficients must be stored consecutively in the order:

             [a0, a1, a2,..., aM, b0, b1,..., bM]

      and i4 is points to this double-length buffer. The a,b coefficients
      used in the program are related to the true a,b coefficients by the
      scale factors, defined by the exponents ea, eb:

             a = a_true / Ga,     Ga = 2^ea = scale factor
             b = b_true / Gb,     Gb = 2^eb = scale factor

      (because a0_true = 1, it follows that a0 = 1/Ga. This
      coefficient is redundant and not really used in the computation; it
      always gets multiplied by zero.)

      The common double-length circular buffer i4 should be declared as:

             .var/pm/circ a[M+1], b[M+1];
             i4 = ^a; L4 = 2*(M+1);

      Program assumes that both numerator and denominator have order M.
      The y- and x-delay-line buffers must be declared as follows:

             .var/dm/circ w[M+1];
             .var/dm/circ v[M+1];
             i2 = ^w; L2 = %w;
             i3 = ^v; L3 = %v;

      typical usage:
      --------------
      cdir(M, i4, m4, i2, m2, i3, m3, ea, eb, ax1);
```

```
        internal operation:
        -------------------
        tapin(i3, m3, ax1);
        mr = 0;
        tapin(i2, m2, mr1);
        dot(M, i4, m4, i2, m2);
        sr = ashift mr1 by ea (hi);
        ay1 = sr1;
        dot(M, i4, m4, i3, m3);
        sr = ashift mr1 by eb (hi);
        ar = sr1 - ay1;
        tapin(i2, m2, ar);
        cdelay(i2, m2);
        cdelay(i3, m3);
}

.macro cdir(%0, %1, %2, %3, %4, %5, %6, %7, %8, %9);

        tapin(%5, %6, %9);                          {u0 = x = input}
        mr = 0;
        tapin(%3, %4, mr1);                         {s0 = 0}
        dot(%0, %1, %2, %3, %4);                    {dot(M,a,s)}
        sr = ashift mr1 by %7 (hi);                 {Ga * dot(M,a,s)}
        ay1 = sr1;
        dot(%0, %1, %2, %5, %6);                    {dot(M,b,u)}
        sr = ashift mr1 by %8 (hi);                 {Gb * dot(M,b,u)}
        ar = sr1 - ay1;                             {output y in AR}
        tapin(%3, %4, ar);                          {s0 = y}
        cdelay(%3, %4);                             {update y-delay}
        cdelay(%5, %6);                             {update x-delay}

.endmacro;
```

The coefficients must be scaled in the same way as in the canonical form. Fig. 5.2 shows the scaled realization.

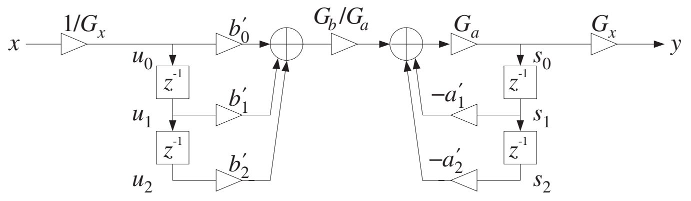


Fig. 5.2 Direct form with scaling factors.

The sample processing algorithm is the circular version of that in Eq. (7.1.10) of the text and can be stated in the following form that uses the scaled coefficients (again, the input prescaling is not implemented inside `cdir`):

```
for each input sample $x$ do:
    $*p_v = u_0 = x$
    $*p_w = s_0 = 0$
    $*p_w = y = s_0 = G_b\mathrm{dot}(M,\mathbf{b}',\mathbf{u}) - G_a\mathrm{dot}(M,\mathbf{a}',\mathbf{s})$
    $\mathrm{cdelay}(M,\mathbf{v},\&p_v)$
    $\mathrm{cdelay}(M,\mathbf{w},\&p_w)$
```

where p_v and p_w are the x-delay and y-delay buffer pointers, $\mathbf{v}$, $\mathbf{w}$ are the corresponding buffers, and $\mathbf{u}$, $\mathbf{s}$ the corresponding states relative to the current values of the pointers.

5.9. wavgen

The macro `wavgen` implements a wavetable generator as discussed in Section 8.1.3 of the text [1]. To facilitate the usage of the DAG registers, the wavetable circular buffer is loaded and circulated in forward order. If the wavetable has length D, and the address increment is c, then the wavetable is cycled over every c of its samples generating a waveform of basic frequency

$$f = c\frac{f_s}{D}$$

The stored waveform is scaled by the amplitude factor A. By assigning two different DAG registers to the wavetable and two different address increments, one can generate waveforms of two different frequencies.

```
{wavgen.dsp - wavetable generator
 Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996.

 Based on wavgen.c of Introduction to Signal Processing, Sec.8.1.3.

      %0 = pointer to wavetable buffer, e.g., i5
      %1 = m-register to use with buffer, e.g., m5
      %2 = desired amplitude A
      %3 = desired increment c (frequency f = c * fs / D)
      %4 = data register holding output, e.g.,
           ax0, ax1, ay0, ay1, ar, mx0, mx1, my0, my1, mr1, sr0, sr1

      registers mx0,my0 are altered

      typical usage:
      --------------
      wavgen(i5, m5, A, c, my1);

      internal operation:
      -------------------
      m5 = c;
      mx0 = A;
      my0 = dm(i5, m5);
      mr = mx0 * my0 (rnd);
```

```
        my1 = mr1;
}

.macro wavgen(%0, %1, %2, %3, %4);

        %1 = %3;
        mx0 = %2;
        my0 = dm(%0, %1);
        mr = mx0 * my0 (rnd);
        %4 = mr1;

.endmacro;
```

6. Appendix

This appendix contains the listings of the following files, all of which reside in the directory `c:\adi_dsp\macros`:

```
dec2hex.c, hex2dec.c
sinetbl.c, squartbl.c, trapztbl.c, uran.c
template.dsp, dspmac.dsp
```

6.1. Decimal-to-Hex Format Converters

The decimal-to-hex format converter `dec2hex.c` uses the `adc.c` routine of the text [1] to get the 2's complement binary representation of a number, and then it collects the bits in groups of fours to get the hex digits. It can be used with any value of B, such as $B = 8, 16, 24, 32$, as long as a and b are such that $a + b = B$:

```
/* dec2hex.c -- decimal to hex (a.b)-format converter
 * Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996
 *
 * Usage: dec2hex a.b < decimal.dat > hex.dat
 *
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

void adc();             2's complement ADC with rounding from I2SP-ch.2

void main(int argc, char ** argv)
{

int i, a, B, *b, *h;                      b = bit vector, h = hex vector
double x, R;

if (argc != 2) {
   puts("\nDecimal to hex (a.b)-format converter");
   puts("Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996\n");
   puts("Usage: dec2hex a.b  < decimal.dat > hex.dat");
   puts("e.g.,  dec2hex 1.15 < decimal.dat > hex.dat \n");
```

```
   puts("In the (1.15)-format, the range is -1 <= x <= 1-2^(-15)");
   puts("and in hex, -1 = 8000, 1-2^(-15) = 7fff");
   puts("\nTotal number of bits: B = a+b, (must be a multiple of 4)");
   puts("Full scale range is R = 2^a");
   puts("Outputs have B/4 hex digits\n");
   puts("The B bits [b_1, b_2, ..., b_B] represent the number:");
   puts("   x = -2^{a-1} b_1 + 2^{a-2} b_2 + ... + 2^{-b} b_B");
   puts("The limits are:");
   puts("   -2^(a-1) <= x <= 2^(a-1) - 2^(-b)");
   puts("e.g., in (4.12)-format, -8 <= x <= 8 - 2^(-12)\n");
   exit(0);
   }

a = atoi(argv[1]);
B = a + atoi(strchr(argv[1], '.') + 1);              total bits

R = pow(2, a);                                       2^a = full scale

b = (int *) calloc(B, sizeof(int));                  B-bit vector
h = (int *) calloc(B/4, sizeof(int));                B/4 hex digits

while(fscanf(stdin, "%lf", &x) != EOF) {
       adc(x, b, B, R);                              convert to binary
       for (i=0; i < B/4; i++) {
           h[i] = 8 * b[4*i]  + 4 * b[4*i+1]  + 2 * b[4*i+2]  + b[4*i+3];
           printf("%x", h[i]);
           }
       printf("\n");
       }
}
```

The program `hex2dec.c` performs the converse operation, that is, hex-to-decimal conversion. For each hex number to be converted, it reads its hex digits as chars, then works out the binary bit pattern of the number, and then calls the text routine `dac.c` to convert it to decimal. Again, one must have $a + b = B$, where the program assumes that there will be $B/4$ hex chars in each hex number to be converted:

```
/* hex2dec.c -- hex to decimal (a.b)-format converter
 * Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996
 *
 * Usage: hex2dec a.b < hex.dat > decimal.dat
 *
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

double dac();                                 dac.c - from I2SP, ch.2
int c2h();

void main(int argc, char ** argv)
{

char *s;
int i, j, a, B, *b;
double x, R;

if (argc != 2) {
   puts("\nHex to decimal (a.b)-format converter");
   puts("Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996\n");
   puts("Usage: hex2dec a.b  < hex.dat > decimal.dat");
   puts("e.g.,  hex2dec 1.15 < hex.dat > decimal.dat\n");
   puts("Total bits: B = a+b, (must be a multiple of 4)");
   puts("Full scale range is R = 2^a");
   puts("Inputs must have B/4 hex digits\n");
   puts("For the (a.b)-format, the outputs are in the range:");
   puts("   -2^(a-1) <= x <= 2^(a-1) - 2^(-b)");
   exit(0);
   }

a = atoi(argv[1]);
B = a + atoi(strchr(argv[1], '.') + 1);                total bits

b = (int *)  calloc(B, sizeof(int));                   B-bit vector
s = (char *) calloc(B/4 + 1, sizeof(char));            B/4 hex digits

R = pow(2, a);                                         2^a = full scale

while(fscanf(stdin, "%s", s) != EOF) {                    hex chars
   for (i=0; i<B/4; i++)                                  i-th hex digit
      for (j=0; j<4; j++)                                 j-th bit
         b[4*i+j] = (c2h(s[i]) & (1 << 3-j)) ? 1 : 0;     masks: 8,4,2,1
   x = dac(b, B, R);                                      scaled output
   printf("% .16lf\n", x);
   }
}                                                         end of main
```

```
/* c2h.c - convert hex char into hex integer
 *
 * input is the char: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,  a,  b,  c,  d,  e,  f
 * output is the int: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
 *
*/

int c2h(c)
char c;
{
       c = tolower(c);

       if (isdigit(c))
              return (c - '0');                  hex digits 0-9
       else
              return (c - 'a' + 10);             hex digits a-f
}
```

6.2. Wavetable Generators

The waveform generator programs below are based on those given in Section 8.1.3 of the text [1]. They all generate one period of a required waveform to be stored

in a circular wavetable buffer. The C program `sinetbl.c` generates one period of length D of a sinusoidal or cosinusoidal signal.

```
/* sinetbl.c - sine wavetable of amplitude A, period D samples */
/* Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996 */

#include <stdio.h>
#include <math.h>

void main(int argc, char ** argv)
{

int c, i, D;
double A, pi = 4 * atan(1.0);

if (argc != 4) {
   puts("\nSine/Cosine Wavetable");
   puts("Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996\n");
   puts("Usage: sinetbl c A D > file.dat\n");
   puts("     c = 0/1 for sine/cosine");
   puts("     A = amplitude");
   puts("     D = period");
   exit(0);
   }

c = atoi(argv[1]);
A = atof(argv[2]);
D = atoi(argv[3]);

for (i=0; i<D; i++)
       if (c==1)
              printf("%.15lf\n", A * cos(2 * pi * i / D));
       else
              printf("%.15lf\n", A * sin(2 * pi * i / D));
}
```

The program `squartbl.c` generates one period of length D of a square wave. One can choose between (a) a completely discontinuous jumping between two levels, or (b) a more gradual transition where the values at the discontinuities are replaced by the mid value between the levels.

```
/* squartbl.c - square wavetable of period D and subperiod D1 */
/* Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996 */

#include <stdio.h>
#include <math.h>

void main(int argc, char ** argv)
{
int i, D, D1;
double A1, A2, x;

if (argc != 6) {
   puts("\nSquare Wavetable");
   puts("Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996\n");
   puts("Usage: squartbl A1 A2 D1 D t > file.dat\n");
   puts("     A1 = amplitude of sub-period D1");
   puts("     A2 = amplitude of remaining period (D-D1)");
   puts("     D1 = sub-period");
   puts("     D  = total period");
   puts("     t = 1  alternates discontinuously from A1 to A2");
   puts("     t = 0  goes through mid-level value (A1+A2)/2");
   exit(0);
   }

A1 = atof(argv[1]);
A2 = atof(argv[2]);
D1 = atoi(argv[3]);
D  = atoi(argv[4]);
x  = 1 - atof(argv[5]);                              x = 1 - t

printf("%.15lf\n", A1 - x * (A1 - A2) / 2);          A1 or (A1+A2)/2

for (i=1; i < D1; i++)
       printf("%.15lf\n", A1);

printf("%.15lf\n", A2 + x * (A1 - A2) / 2);          A2 or (A1+A2)/2

for (i=D1+1; i < D; i++)
       printf("%.15lf\n", A2);

}
```

The program `trapztbl.c` implements a trapezoidal waveform generator, which also includes a triangular waveform as a special case.

```
/* trapztbl.c - trapezoidal wavetable of period D and subperiods D1, D2 */
/* Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996 */

#include <stdio.h>
#include <math.h>

void main(int argc, char ** argv)
{
int i, D, D1, D2;
double A;

if (argc != 5) {
   puts("\nTrapezoidal Wavetable");
   puts("Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996\n");
   puts("Usage: trapztbl A D1 D2 D > file.dat\n");
   puts("     A = amplitude");
   puts("     D1 = duration of linearly increasing portion");
   puts("     D2 = duration of constant part");
   puts("     D  = total period");
   puts("     D-(D1+D2) = duration of linearly decreasing portion");
   exit(0);
   }

A =  atof(argv[1]);
D1 = atof(argv[2]);
D2 = atoi(argv[3]);
D  = atoi(argv[4]);

for (i=0; i<D; i++)
```

```
       if (i < D1)
              printf("%.15lf\n", i * A / D1);
       else
              if (i < D1 + D2)
                     printf("%.15lf\n", A);
              else
                     printf("%.15lf\n", (D - i) * A / (D - D1 - D2));
}
```

Finally, the program `uran.c` generates a block of uniformly-distributed random numbers of any desired mean and range. It is useful in designing the random number inputs to the Karplus-Strong string algorithm. It uses the C function `ran.c` of the text [1].

```
/* uran.c - generate block of uniform random numbers
 * Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996
 *
 * range m-R <= x < m+R
 *
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

double ran();                                 ran.c from I2SP, Appendix B

void main(int argc, char **argv)
{
       int N, i;
       long iseed;
       double m, R, x;

       if (argc != 5) {
              puts("Generate a block uniform random numbers");
              puts("Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996\n");
              puts("Usage: uran m R N iseed > x.dat\n");
              puts("N = length of block");
              puts("range is the interval [m-R, m+R), mean = m\n");
              exit(0);
              }

       m = atof(argv[1]);
       R = atof(argv[2]);
       N = atoi(argv[3]);
       iseed = atol(argv[4]);

       for (i=0; i<N; i++) {
              x = 2 * R * (ran(&iseed) - 0.5) + m;
              printf("% .15lf\n", x);
              }
}
```

6.3. Template and Begin/End Files

The file `template.dsp` serves as the starting point of all program examples:

```
{template.dsp - use as template for sample processing algorithms}
{Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996}

{Based on mic2out.dsp from Analog Devices FTP site and the sample talkthru
 program of the EZ-KIT Lite Reference Manual.}

{--- choose sampling rate in kHz: ----------------------------------------}
                    {0xc850 = 8      | 0xc851 = 5.5125   | 0xc852 = 16    }
                    {0xc853 = 11.025 | 0xc854 = 27.42857 | 0xc855 = 18.9  }
.const fs = 0xc850; {0xc856 = 32     | 0xc857 = 22.05    | 0xc859 = 37.8  }
                    {0xc85b = 44.1   | 0xc85c = 48       | 0xc85d = 33.075}
                    {0xc85e = 9.6    | 0xc85f = 6.615                     }
{-------------------------------------------------------------------------}

.include <c:\adi_dsp\macros\begin.dsp>;    {initializations and DSP macros}

{--- define constants, variables, and buffers ----------------------------}
{--------------------------- example -------------------------------------}
{.const D = 7670;           approximately max delay for EZ-KIT Lite       }
{.var/dm/circ w[D+1];       circular delay-line buffer of length D+1      }
{                                                                         }
{i2 = ^w; L2 = %w;          delay-line buffer pointer and length          }
{                                                                         }
{zero(i2, m2, L2);          initialize delay-line to zero                 }
{-------------------------------------------------------------------------}

{--- start processing input samples --------------------------------------}

wait: idle; jump wait;             {wait for interrupt and loop forever}
                                   {interrupt service routine starts here}
input_samples: ena sec_reg;        {enable secondary register set}

{--- read input samples from codec ---------------------------------------}

   ax1 = dm(rx_buf + 1);           {left input sample}
   mx1 = dm(rx_buf + 2);           {right input sample}

{--- sample processing algorithm -----------------------------------------}
{---------------------------------- example ------------------------------}
{  tap(i2, m2, D, my1);            put Dth-tap output of delay into MY1   }
{  tapin(i2, m2, mx1);             put current input from MX1 into 0th tap}
{  cdelay(i2, m2);                 update delay                           }
{-------------------------------------------------------------------------}

{--- write output samples to codec ---------------------------------------}

   dm(tx_buf + 1) = ax1;           {left output sample}
   dm(tx_buf + 2) = mx1;           {right output sample}

{--- return from interrupt -----------------------------------------------}

   rti;

.include <c:\adi_dsp\macros\end.dsp>;     {wrapup}
```

The files `begin.dsp` and `end.dsp` in the directory `c:\adi_dsp\macros` are the beginning and ending parts of the talkthru program in the EZ-KIT Lite's Reference Manual [2] and the `mic2out.dsp` program found in [3].

We have made only two modifications to these files: (1) we redefined the codec's data format register so that it can be passed in as the constant `fs`, which may be selected at the beginning of every example program, and (2) we added the include-file `dspmac.dsp`, which includes all the DSP macros; its listing is:

```
{dspmac.dsp - includes all DSP macros.
 Junior DSP Lab - Rutgers ECE Dept - S. J. Orfanidis - Jan 1996.

 typical usage:
       cdelay(i2, m2);
       tap(i2, m2, d, my1);
       tapin(i2, m2, mx1);
       dot(M, i4, m4, i2, m2);
       cfir(M, i4, m4, i2, m2, mx1);
       ccan(M, i4, m4, i2, m2, ea, eb, ax1);
       cdir(M, i4, m4, i2, m2, i3, m3, ea, eb, ax1);
       wavgen(i5, m5, A, c, my1);
       zero(i2, m2, L2);
}

.include <c:\adi_dsp\macros\cdelay.dsp>;
.include <c:\adi_dsp\macros\tap.dsp>;
.include <c:\adi_dsp\macros\tapin.dsp>;
.include <c:\adi_dsp\macros\dot.dsp>;
.include <c:\adi_dsp\macros\cfir.dsp>;
.include <c:\adi_dsp\macros\ccan.dsp>;
.include <c:\adi_dsp\macros\cdir.dsp>;
.include <c:\adi_dsp\macros\wavgen.dsp>;
.include <c:\adi_dsp\macros\zero.dsp>;
```

References

[1] S. J. Orfanidis, *Introduction to Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1996.

[2] *ADSP-2100 Family EZ-KIT Lite Reference Manual*, Analog Devices, Norwood, MA, 1995. Available also from `www.analog.com`.

[3] Analog Devices BBS (617) 461-4258, or anonymous ftp site `ftp.analog.com`, or web site `www.analog.com`. The `ezld.com` utility is included in the file `ezkfo-rum.zip` in the subdirectory `pub/dsp/ezktil` of the ftp site.