Multirate filters change the sampling rate of a signal—they convert the input samples of a signal to a different set of data that represents the same signal sampled at a different rate. Some examples of applications of multirate filters and systems are:

- Sample-rate conversion between digital audio systems
- Narrow-band and band-pass filters
- Sub-band coding for speech processing in vocoders
- Quadrature modulation

Decimation and interpolation are the two basic types of multirate filtering processes.

- Decimation is the process of reducing the sample rate of the signal. A decimation filter removes redundant information from the original sampled signal thereby compacting the data.

- Interpolation is the process of increasing the sample rate of the signal. An interpolation filter inserts missing data between the existing samples.

Multirate systems, such as sample-rate conversions, are combinations of decimation and interpolation filters.

The six multirate filters described in this chapter are:

- single-stage decimation filter
- single-stage interpolation filter
- rational rate changer (timer based)
- rational rate changer (interrupt based)
- two-stage decimation filter
- two-stage interpolation filter
The descriptions of the implementation of these filters for the ADSP-21000 family assume that you already understand the theory behind them. For more details on the theory of operation of multirate filters, refer to the Multirate Filter section in [ADI90].

5.1 SINGLE-STAGE DECIMATION FILTER
A single-stage decimation filter simply resamples the digital signal with anti-aliasing. The code decimate.asm (Listing 5.1) uses an \( N \)-tap Direct Form Finite Impulse Response Filter, and decimates the signal data in real time by a factor of \( M \). This process yields a decrease in the input sample rate of \( 1/M \).

The input signal to the decimator is a signal that is oversampled by \( M \); that is, it is sampled at \( M \) times the desired output frequency. The decimation filter algorithm consists of two operations. First, a lowpass FIR filter bandlimits the input signal to one-half of the output frequency to prevent aliasing. Then an decimation removes \( M - 1 \) samples of every \( M \) samples to create an output signal that is \( 1/M \) times the input sample rate. Both of the operations are performed in a single FIR routine.

Data acquisition systems take advantage of decimation filters to avoid using expensive, high-performance analog anti-aliasing filters. A system oversamples the input signal by a factor of \( M \) and then decimates to the desired sample rate. By oversampling the input signal, the transition band of the front end filter to prevent aliasing, thus an inexpensive analog filter can be used. The decimation filter will then reduce the input single to the required sample rate.

5.1.1 Implementation
The decimate.asm code starts by initializing the data buffers (input_buf, data, and coef) and the respective DAG registers. The input buffer, input_buf, is \( M \) long where \( M \) is the decimation factor. The data buffer, data, and filter coefficient buffer, coef, are both \( N \) locations long, where \( N \) is the number of taps for the filter. The I7 and L7 registers are specifically used for the input_buf so that the circular buffer overflow interrupt can be used. The timer period is also programmed to the desired input frequency. Since the timer interrupt happens during the circular buffer overflow service routine, interrupt nesting is enabled.
Multirate Filters

The timer interrupt service routine (tmzh_svc) gets the input sample from an A/D converter port (adc) and stores the samples in the input buffer input_buf. When M samples have been acquired, the input buffer is full and a circular buffer overflow interrupt is generated.

The circular buffer overflow interrupt service routine (cb7_svc) calls the decimator routine (decimate) only every M times, not at every sample. This removes M–1 samples in between decimator calls.

The decimate routine transfers the input buffer to the data buffer as input for the decimation filter, converting from the sampled 16-bit fixed point number to an equivalent 32-bit floating point number. This double buffering allows the filter computations to proceed in parallel with the acquisition of the next M inputs, allowing a larger order filter than if all the filter calculations are made between input sample periods.

To save cycles, the data transfer is performed in a rolled loop to allow the delayed branch (db) call. The fix-to-float and scaling operations are performed in parallel, also to save cycles. If the decimation factor is less than four, the data conversion and transfer is more efficient if performed unrolled.

After the input_buf buffer is transferred to the data buffer, the decimate routine runs the FIR filter on the sampled data. The FIR filter is also a rolled loop where the first three data acquisitions are performed outside the loop. Therefore, the loop counter is set to N – 3, where N is the number of taps in the filter. The output of the FIR is then converted back to a 16-bit fixed point format and written to the output D/A converter (dac). The STKY bit for the circular buffer overflow is also cleared to prevent reentry to the interrupt service routine.

Since the acquisition of the input data and calculation of the FIR filter are in parallel, the decimator must calculate one pass of the FIR filter while M samples are being read into the input buffer, input_buf. The following equation determines the value of the timer period:

\[
\text{Minimum Timer Period} = \max(\text{ceil}[(N + 2\times M + 11 + 5\times M)/M], 20)
\]

The FIR filter is invoked by cb7_svc, which requires (N + M * 2 + 11) cycles to execute. To obtain M samples requires an additional (5 * M) cycles. Dividing by M yields the timer period, but we must take the greatest integer value larger than this number (ceiling, or ceil function). Additionally, the input buffer must be completely moved to the data buffer.
buffer at the start of the \texttt{cb7_svc} routine before the first timer interrupt occurs again. Because the entire decimate routine (19 cycles) must be executed between each sampling, there won’t be enough time for this if the timer period is not at least 20.

Here is an example calculation for a 20 MHz ADSP-210xx processor:

\[
\begin{align*}
\text{FP} &= \text{processor frequency} = 33.3 \\
N &= \text{number of taps} = 64 \\
M &= \text{decimation factor} = 8 \\
\text{Minimum Timer Period} &= \max(\lceil(64+2*8+11+5*8)/8\rceil, 20) \\
&= \max(\lceil 131/8 \rceil, 20) \\
&= \max(17, 20) \\
&= 20
\end{align*}
\]

Maximum Input Frequency = \(\text{FP}/20 = 33.3\text{MHz}/20 = 1.665 \text{MHz}\)

Maximum Output Frequency = \(\text{FP}/(M*20) = 33.3\text{mHz}/(8*20) = 208 \text{kHz}\)

If \(N=128\), and \(M=8\), 25 cycles is the minimum timer period.
5.1.2  Code Listings–decimate.asm

/************************************************************************
File Name
   DECIMATE.ASM

Version
   3/4/91

Purpose
   Real Time Decimator

Calling Parameters
   Input:  adc
         r15 = ADC input data

Return Values
   Output: dac
         r0 = DAC output data

Registers Affected
   r0 = FIR data in / ADC fltg-pt data
   r4 = FIR coefficients / temporary reg
   r8 = FIR accumulate result reg
   r12 = FIR multiply result reg / temporary reg
   r14 = 16 = exponent scaling value

Start Labels:
   init_dec    reset-time initialization
   decimate    called by CB7 interrupt

Computation Time:
   cb7_svc = 6 + decimate = N + M*2 + 11
   decimate = N + M*2 + 5 + [5 misses]
   tmzh_svc = 5

Minimum Timer Period:    max( ceil[(N+2*M+11+5*M)/M], 20 )

# PM Locations
   47 Words Code, N Words Data

# DM Locations
   N + M Words Data

************************************************************************/

(listing continues on next page)
Creating & Using the Test Case:

Running the test case requires two data files:

- coef.dat: 32-tap FIR filter coefficients (floating point)
- wave.1: waveform data file

Coef.dat contains a 32-tap FIR filter which bandlimits the input signal to 1/8 of the input frequency. Since the decimator in the test case decimates by a factor M=4, this bandlimitation is equivalent to the required limit of 1/2 the output frequency. The filter is a Parks-McClellan filter with a passband ripple of 1.5dB out to about 1/10 the input frequency, and a stopband with greater than 70dB of attenuation above 1/8 the input frequency.

As an example, if the oversampled input frequency is 64kHz, the passband extends out to about 6.4kHz. The stopband starts at 8kHz, and the output frequency is 16kHz.

The example data file is wave.1. It contains 512 signed fixed-point data which emulate a 16-bit A/D converter whose bits are read from Data Memory bits 39:24. Wave.1 contains a composite waveform generated from 3 sine waves of differing frequency and magnitude. The cb7_svc routine arithmetically shifts this data to the right by 16 bits, effectively zero-ing out the garbage data which would be found in bits 23:0 if a real 16-bit ADC port were read.

The test case writes the decimated output to a dac port. Since there are 512 samples in wave.1, and the test case performs decimation by 4, there will be 128 data values written to the dac port if all data samples are read in and processed. The values written out are left-shifted by 16 bits in parallel with the float→fixed conversion, based again on the assumption that the D/A converter port is located in the upper 16 bits of data memory.

Armed with this information, you are ready to run:

1. Assemble & Start Simulator:
   - asm21k -DTEST decimate
   - ld21k -a generic decimate
   - sim21k -a generic -e decimate

2. In the simulator,
   - a. Go to the program memory (disassembled) window,
   - d. Go to symbol cb7_done, set to break on 129th occurrence, and
   - c. Run
3. Compare wave.1 to out.1 on the graphing program of your choice.

```c
#include "def21020.h"
#define N 32                /*  number of taps */
#define M 4                 /*  decimation factor */
#define FP 20.0e6           /*  Processor Frequency = 20MHz */
#define FI 64.0e3           /*  Interrupt Frequency = 64KHz */

#ifndef TEST /*============================================================*/
#define TPER_VAL 312        /*  TPER_VAL = FP/FI - 1 */
#else /*==============================================================*/
#define TPER_VAL 19         /* interrupt every 20 cycles */
#endif /*==============================================================*/

.SEGMENT /dm dm_data;
.VAR    data[N];            /* FIR input data */
.VAR    input_buf[M];       /* raw A/D Converter data */
.ENDSEG;

.SEGMENT /pm pm_data;
.VAR    coef[N]="coef.dat"; /* FIR floating-point coefficients */
.ENDSEG;

.SEGMENT /dm ports;
.PORT   adc;                /* Analog to Digital (input) converter */
.PORT   dac;                /* Digital to Analog (output) converter */
.ENDSEG;

/*============================================================
 RESET Service Routine
============================================================*/

_SEG /pm rst_svc;
 rst_svc:        PMWAIT=0x0021;        /* no wait states, internal ack only */
                 DMWAIT=0x8421;        /* no wait states, internal ack only */
                 jump init_dec; /* initialize the test case */
 .ENDSEG;

/*============================================================
 TMZH Service Routine
============================================================*/

_SEG /pm tmzh_svc;
 /* sample input: this code occurs at the sample rate */
 tmzh_svc:       rti(db);
                  r15=dm(adc);        /* get input sample */
                  dm(i7,m0)=r15; /* load in M long buffer */
 .ENDSEG;
```

(listing continues on next page)
5 Multirate Filters

/* DAG1 Circular Buffer 7 Overflow Service Routine */
.SEGMENT /pm cb7_svc;
/* process input data when circular buffer 7 wraps around ("overflows") */
/* this code occurs at 1/M times the sample rate */
cb7_svc:
call decimate (db);
    /* call the decimator */
    r12=dm(i7,m0);
    r4=ashift r12 by -16;
    rti (db);
    /* return from interrupt */
    r0 = fix f0 by r14;
    /* convert to fixed point */
cb7_done:
    dm(dac)=r0;
    /* output data sample */
.ENDSEG;

/* efficient decimator initialization */
.SEGMENT /pm pm_code;
.init_dec:
b0=data; l0=@data; m0=1;
    /* data buffer */
b7=input_buf; l7=@input_buf;
    /* input buffer */
b8=coef; l8=@coef; m8=1;
    /* coefficient buffer */
tperiod=TPER_VAL;
    /* program timer */
tcount=TPER_VAL;
/* enableTimer high priority, CB7 interrupts */
bit set imask TMZHI|CB7I;
/* clear interrupts before setting global enable */
bit clr irpt0 TMZHI|CB7I;

r14=16;
    /* scale factor float–>fixed */
r15=0;
    /* clear data buffer */
lcnt=ln, do clrbuf until lce;
clrbuf:
    dm(i0,m0)=r15;
/* enable interrupts, nesting mode, ALU saturation */
bit set mode1 IRPTEN|NESTM|ALUSAT;
bit set mode2 TIMEN;
    /* enable the timer */
wait:
    idle;
    jump wait;
    /* infinite wait loop */
.ENDSEG;
Multirate Filters

Listing 5.1  decimate.asm
5 Multirate Filters

5.2 SINGLE-STAGE INTERPOLATION FILTER
A single-stage interpolation filter reconstructs a discrete-time signal from another discrete-time signal. This is analogous to the decimator where a sampled signal is sampled again at a different rate. By inserting more data points between the originally sampled data points, interpolation increases the output sample rate. Interpolation performs the function of a digital anti-imaging filter on the original signal. Many digital audio systems, such as compact disc and digital audio tape players, use interpolation (oversampling) to avoid using high-performance analog reconstruction filters. The anti-imaging functions in interpolators allow the use of low-order analog filters on the D/A outputs.

To increase the input sampling frequency by a factor of $L$, $L - 1$ zeroes are inserted between the original samples and then low-pass filtered. The insertion of zeros between samples and the smoothing filter fill in the data missing between the original samples. The filter must bandlimit the output signal to one-half the output frequency to prevent imaging. Since one input sample now corresponds to $L$ output samples, the sample rate is increased by $L$.

5.2.1 Implementation
The interpol.asm code (Listing 5.2) uses an $N$-tap Direct Form Finite Impulse Response Filter in an efficient algorithm to interpolate by $L=4$. This increases the input sample rate by a factor of four. The number of taps is restricted such that $N/L$ must be an integer. An undesirable by-product of the FIR filter is the attenuation of the input signal by a factor of $L$. To correct this, the input data is scaled up by $L$ before it is filtered.

Because most of the delay line inputs are zeros, calculation time would be wasted if the zero inputs were multiplied by their corresponding coefficients. Instead, the coefficients are interleaved to perform only the non-trivial calculations. This technique is known as Poly-phase filtering. Thus, the delay line must be only $N/L$ locations long. In this example $N=32$ and $L=4$, so the delay line, data, is eight location long. The coefficient buffer, coef, is 32 locations long. The example initializes the delay line address modifier to one and the coefficient buffer address modifier to $L = 4$, for the proper interleave.

The internal timer generates interrupts at $L$ times the output sample rate, which is the desired output frequency. The register R2 is used as a counter in the timer service routine (tmzh_svc) to get a new sample from the A/D converter (sample) every $Lth$ pass of the interrupt. When a new
sample is read in, it is converted from a 16-bit fixed point value to a 32-bit floating point value and scaled up by $L$ to compensate for the attenuation effect. The routine also resets the counter to $L$ and modifies the coefficient pointer by $L$ for proper interleave.

The timer service routine calls the interpolator ($\text{interpol}$). The interpolator runs $N/L$ taps of the FIR filter at the output frequency rate. To achieve the proper interleaving of coefficients with input data, the coefficient pointer is modified by $-1$ before the interpolation takes place. During the interpolation filter, the coefficient pointer is then modified by $L$ after every access. This interleaving process is equivalent to running an $N$-tap FIR filter on data generated by adding $L-1$ zero samples to each data input.

The filter output is converted back to a 16-bit fixed point value and written to a D/A converter ($\text{dac}$) after each interpolation. The technique of rolled loops and delayed branches are used to conserve cycles.
5.2.2 Code Listing–interpol.asm

/************************************************************************
File Name
   INTERPOL.ASM

Version
   3/6/91

Purpose
   Real time Interpolator

Calling Parameters
   Input:  adc
         r15 = ADC input data

Return Values
   Output: dac
         r0 = DAC output data

Registers Affected
   r0 = FIR data in / ADC data / temp register
   r1 = scale factor L
   r2 = counter
   r4 = FIR coefficients
   r8 = FIR accumulate result reg
   r12 = FIR multiply result reg / temporary reg
   r14 = 16 = exponent scale factor

Start Labels
   init_int    reset-time initialization
   interpol    called by Timer Zero High Priority interrupt

Computation Time
   tmzh_svc = 5 + interpol
      = N/L + 9   (no sample)
      = N/L + 17  (with sample)

   interpol = N/L + 4
      = N/L + 12 + [5 misses]  (with sample)

Minimum Timer Period:       N/L+20
Maximum Output Frequency:   FP/(N/L+20)
Maximum Input Frequency:    FP/(L*(N/L+20))

# PM Locations
   44 Words Code, N Words Data

# DM Locations
   N/L Words Data
************************************************************************/

124
Multirate Filters

Creating & Using the Test Case:

Running the test case requires two data files:

- coef.dat  32-tap FIR filter coefficients (floating point)
- wave.1    waveform data file

c_coef.dat contains a 32-tap FIR filter which bandlimits the output signal to 1/8 of the output frequency. Since the interpolator in the test case interpolates by a factor M=4, this bandlimitation is equivalent to the required limit of 1/2 the input frequency. The filter is a Parks-McClellan filter with a passband ripple of 1.5dB out to about 1/10 its input frequency, and a stopband with greater than 70dB of attenuation above 1/8 the input frequency.

As an example, if the interpolated output frequency is 64kHz (==input frequency of 16kHz), the passband extends out to about 6.4kHz. The stopband starts at 8kHz, and the output frequency is 16kHz.

The example data file is wave.1. It contains 512 signed fixed-point data which emulate a 16-bit A/D converter whose bits are read from Data Memory bits 39:24. wave.1 contains a composite waveform generated from 3 sine waves of differing frequency and magnitude. The cb7_svc routine arithmetically shifts this data to the right by 16 bits, effectively zero-ing out the garbage data which would be found in bits 23:0 if a real 16-bit ADC port were read.

The test case writes the interpolated output to a dac port. Since there are 512 samples in wave.1, and the test case performs interpolation by 4, there will be 2048 data values written to the dac port if all data samples are read in and processed. The values written out are left-shifted by 16 bits in parallel with the float→fixed conversion, based again on the assumption that the D/A converter port is located in the upper 16 bits of data memory.

Armed with this information, you are ready to run:

1. Assemble & Start Simulator:

  asm21k -DTTEST interpol
  ld21k -a generic interpol
  sim21k -a generic -e interpol -k port1

   Note: the keystroke file port1.ksf opens two ports:
   1st port - DM, F0000000, Fixed-Pt., auto-wrap, wave.1
   2nd port - DM, F0000001, Fixed-Pt., no wrap, out.1

2. In the simulator,
   a. Go to the program memory (disassembled) window,
   d. Go to symbol sample, set to break on 513th occurrence, and
   c. Run

3. Compare wave.1 to out.1 on the graphing program of your choice.
#include "def21020.h"
#define N 32          /* number of taps */
#define L 4           /* interpolate by factor of L */
#define FP 20.0e6     /* Processor Frequency = 20MHz */
#define FI 64.0e3     /* Interrupt Frequency = 64KHz */

#ifdef TEST /*——————————————————————————*/
#define TPER_VAL 312 /* TPER_VAL = FP/FI - 1 */
#else /*————————————————————————————————*/
#define TPER_VAL 27  /* interrupt every 28 cycles */
#endif /*————————————————————————————————*/

.SECTION /dm dm_data;
.VAR data[N/L];              /* ADC fixed-point data buffer */
.ENDSECT;

.SECTION /pm pm_data;
.VAR coef[N]="coef.dat";     /* fltg-pt. FIR coefficients */
.ENDSECT;

.SECTION /dl ports;
.PORT adc;                /* Analog to Digital (input) converter */
.PORT dac;                /* Digital to Analog (output) converter */
.ENDSECT;

/*______________________________________________________________________________
RESET Service Routine
______________________________________________________________________________*/

.SECTION /pm rst_svc;
 rst_svc:        PMWAIT=0x0021;      /* no wait states,internal ack only */
                   DMWAIT=0x8421;      /* no wait states,internal ack only */
                   jump init_int;      /* initialize the test case */
.ENDSECT;

/*______________________________________________________________________________
TMZH Service Routine
______________________________________________________________________________*/

.SECTION /pm tmzh_svc;
/* sample input: this code occurs at the L*input rate */
 tmzh_svc:       r2=r2-1,modify(i8,m9);          /* decrement counter, and */
                   if eq call sample;              /* test and input if L times */
                   jump interpol (db);             /* perform the interpolation */
                   f0=dm(i0,m0), f4=pm(i8,m8);
                   f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
.ENDSECT;

/*———————————————————————————————————————
Initialize Interpolation Routine
———————————————————————————————————————*/

.SEGMENT /pm    pm_code;
.init_int:       /*  initialize buffer index registers  */
b0=data; t0=@data; m0=1;
b8=coef; t8=@coef; m8=L;        /* modifier for coef is L */
m9=-1;

/* Register Initialization */
  r1=L;                   /* value for upscaling sample */
f1=float r1;            /* fix —> float conversion */
r2=1;                   /* set counter to 1 for 1st sample */
r14=16;                 /* exponent scale factor */
tperiod=TPER_VAL;       /* program timer */
tcount=TPER_VAL;
  bit set imask TMZHI;    /* enable TMZH interrupt */
  bit clr irptl TMZHI;    /* clear any pending interrupts */
  bit set mode1 IRPTEN|ALUSAT; /* enable interrupts, ALU sat */
  bit set mode2 TIMEN;    /* turn timer on */
  r0=0;                   /* clear data buffer */
  lcnd=N, do clrbuf until lce;
clrbuf:             dm(i0,m0)=r0;
wait:           idle;
  jump wait;              /* infinite wait loop */
.ENDSEG;

/*———————————————————————————————————————
sample & interpolate
code executes at the (L*input_rate)
———————————————————————————————————————*/

.SEGMENT /pm    pm_code;
.interpol:       /* FIR Filter, occurs at L times the input input rate. */
  /* First two instructions of this filter have been run */
  /* in the delay field of the jump to this routine */
f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
  lcnd=N/L-3, do taps until lce;
taps:               f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
f12=f0*f4, f8=f8+f12;
f0=f8+f12;
rti (db);
  r0 = fix f0 by r14;       /* float -> fixed */
    dm(dac)=r0;              /* output data sample */
.ENDSEG;

(listing continues on next page)
5 Multirate Filters

Listing 5.2 interpol.asm

/*
get sample
code executes at the input_rate
*/

.SEGMENT /pm    pm_code;
/* input data sample once every L times (i.e., at the input rate) */
sample:         r0=dm(adc);                     /* input data sample */
/* input sample occupies bits 39:24; shift to 15:0 */
r0 = ashift r0 by -16;          /* shift w/ sign extend */
/* do fix->float, and shift coef pointer up by L */
f0=float r0, modify(i8,m8);    /* upscale sample by L */
 rts(db), f0=f0*f1;               /* upscale sample by L */
 dm(i0,m0)=f0;                    /* update delay line */
 r2=m8;                          /* reset counter to L */

.ENDSEG;
5.3 RATIONAL RATE CHANGER (TIMER-BASED)

This section describes a rational rate changer implemented by cascading an interpolating filter and a decimating filter. Interpolating filters and decimating filters can only change the rate of a signal by a factor that is an integer. This restriction makes these filters useless for many real world applications when used by themselves. By cascading these filters, the frequency may be changed by a factor that is a rational number.

For example, to transfer data between a compact disc with a sampling rate of 44.1 kHz to a digital audio tape player with a sampling rate of 48.0 kHz, the sampling rate of the compact disc must be increased by a factor of 48/44.1, which is obviously not an integer. By using a combination of an interpolation filter and decimation filter, a rational number can be determined that approximates the desired frequency change.

The interpolation must take place first so as not to lose some of the desired output frequency bandwidth that otherwise is filtered out by the decimator. The anti-imaging filter (which operates after interpolation) and anti-aliasing filter (which operates before decimation) can be combined into a single low-pass filter.

5.3.1 Implementation

This example, ratiobuf.asm (Listing 5.3), uses an $N$-tap (in this case, $N = 32$) real time direct FIR filter that performs interpolation by $L = 4$ and decimation by $M = 3$ for a $L/M = 4/3$ change in the input sample rate. The same techniques used in both interpolation and decimation filters are utilized to dedicate an entire output period for calculating one output sample. The ratio $N/L$ is restricted to be an integer.

Like the interpolation algorithm, the ratio changer samples the incoming signal at $L$ times the sampling rate using the timer interrupt, and only updates the input buffer every $L$th pass. Also, every zero-valued multiplication is skipped by interleaving the coefficients. Like the decimator algorithm, the input is double-buffered so that the filter computations can occur in parallel with the input and output of data. This allows the use of a larger order filter for a given input sample rate.

This ratio changer implementation is counter based. The counter for the inputs, IN_CNTR, is set to $L$, and the counter for the outputs, OUT_CNTR, is set to $M$. If the IN_CNTR expires, a new input sample is read into the input buffer. If the OUT_CNTR expires, a flag is set to indicate that the main routine must calculate a new output sample.
The initialization routine sets the input buffer (input_buf) length to an integer value larger than \( M/L \). For this example, the interpolation factor \( \( L \) \) is four and the decimation factor \( (M) \) is three. Therefore, the input buffer length is equal to one (the integer \( \geq M/L \); \( 1 \geq 3/4 \)). This means that an output occurs at least once before the next input is received. The data buffer length is set to \( N/L = 32/4 = 8 \) just as in the single-stage interpolation. To properly interleave the coefficients for the filter performing both interpolation and decimation, the buffer pointer \( I9 \) is used to keep track of the coefficient updates. Both counters (IN_CNTR and OUT_CNTR) are initialized to one so that the first sample is read in and an output is generated.

The program then waits in an idle loop for the timer interrupt. Every timer interrupt, the counters are decremented and checked to see if either or both have expired:

- If \( \text{IN\_CNTR} \) has not expired, the service routine jumps to skipin where it checks to see if the \( \text{OUT\_CNTR} \) has expired.
- If \( \text{IN\_CNTR} \) has expired, the program jumps to the routine input where the next value is read in from the A/D, scaled up by \( L \), and stored in \( \text{input\_buf} \). The counter is reset and the \( \text{OUT\_CNTR} \) is then checked to see if an output sample needs calculation.
- If the \( \text{OUT\_CNTR} \) has not expired, the program returns to the idle statement to wait for the next timer interrupt.
- If the \( \text{OUT\_CNTR} \) has expired, the User Software Interrupt 0 (bit 24 in the IRPTL register) is set to force a software interrupt. This interrupt service routine jumps to the routine calc_out.

The routine calc_out first determines the number of elements in input_buf:

- If the value is zero, a new A/D value was not input since the last time an output was calculated. The data transfer from \( \text{input\_buf} \) to \( \text{data\_buf} \) is skipped.
- If the \( \text{input\_buf} \) is not empty, the stored samples are transferred to the \( \text{data\_buffer} \).
The `calc_out` routine then adjusts the pointer to the coefficients for the proper interleave, and executes the low-pass filter code to generate an output. The low-pass filter is an \( \frac{N}{L} \) tap filter implemented as a rolled loop, similar to both the single-stage interpolation and decimation filters.

At the end of the `calc_out` routine, the `input_buf` pointer and `coef` are reset to their original values. The final output is converted back to a 16-bit fixed-point value and sent to the DAC. The routine then returns to the idle loop to wait for the next timer interrupt.

There are several instances of unusual instruction ordering in the `calc_out` subroutine. This ordering is used in order to minimize total execution time, but requires explanation.

If an instruction modifies an I or M register and then the next instruction uses the same DAG, a single NOP cycle is inserted. As explained in the *ADSP-21020 User’s Manual*, this dead cycle is inserted because the DAG needs to calculate indirect addresses for a given cycle during the previous cycle. To avoid this dead cycle, modifications of an I or M register are separated from use of a DAG where possible, even though the code ordering becomes more difficult to follow.

For example, the index register I9 is used to track coefficient updates, and must be modified in the `output` subroutine. These instructions are executed:

```c
modify(i9,-M);
m10=i9;
modify(i8,m10);
i9=0;
```

A dead cycle occurs after the second instruction and possibly after the fourth instruction (if the fifth instruction uses DAG2). To avoid dead cycles, the first two instructions are moved away from the third, and the fourth instruction is moved to a place where a DAG2 operation does not follow it.

Additionally, there are two places in the `output` subroutine where `i7=b7` must be executed. Performing this ureg-to-ureg transfer by itself wastes cycles, so the transfer is combined with unrelated computations to form a multifunction instruction. The first `i7=b7` transfer must occur before the input buffer is transferred to the delay line. This transfer is moved earlier in the code in order to combine it with an unrelated
5 Multirate Filters

computation, saving one cycle. The second \( i_7 = b_7 \) can occur immediately after the input buffer is transferred, but it is moved later in the code so it can be combined with an unrelated computation.

There are two reasons why the user software interrupt 0 (USI0) is used to call the `calc_out` routine. The first reason is that the `calc_out` routine must be allowed to be interrupted by the timer interrupt service routine. The timer interrupt occurs more frequently than the `calc_out` routine. If the timer interrupt routine called the `calc_out` routine, the timer interrupt would occur again before the `calc_out` routine had completed. An interrupt routine cannot interrupt itself. Therefore, the `calc_out` routine is called from a lower priority interrupt (USI0).

This code structure also allows an easier transition from the counter-based rational rate changer algorithm to the external interrupt based algorithm described in the next section.
5.3.2 Code Listings–ratiobuf.asm

/**************************************************************************************/
File Name
RATIOBUF.ASM

Version
3/4/91

Purpose
Timer-Driven Real Time Rational Rate Changer

Calling Parameters
Input: adc
r0 = ADC input data

Return Values
Output: dac
r0 = DAC output data

Registers Affected
r0 = FIR data in / ADC data / temp register
r1 = temp register
r2 = input counter
r3 = output counter
r4 = FIR coefficients
r5 = scale value L
r8 = FIR accumulate result reg
r12 = FIR multiply result reg / temporary reg
r14 = 16 = exponent scale factor

Start Labels
init_rat reset-time initialization
input called by Timer Zero High Priority interrupt
calc_out called by Software Interrupt 0

Computation Time
tmzh_svc = 15
sft0_svc = NoverL + 2+intMoverL + 16

# PM Locations
67 Words Code, N Words Data

# DM Locations
N/L + M/L Words Data
**************************************************************************************/

(listing continues on next page)
# 5 Multirate Filters

/*———————————————————————————————————————
Creating & Using the Test Case:

Running the test case requires two data files:
    coef.dat        32-tap FIR filter coefficients (floating point)
    wave.1          waveform data file

Descriptions of coef.dat and wave.1 can be found in decimate.asm &
interpol.asm.

The test case writes the rate-changed output to a dac port. Since there are
512 samples in wave.1, and the test case performs interpolation by 4 followed
by decimation by 3, there will be 682 data values written to the dac port if
all data samples are read in and processed. The values written out are
left-shifted by 16 bits in parallel with the float->fixed conversion, based
again on the assumption that the D/A converter port is located in the upper 16
bits of data memory.

Armed with this information, you are ready to run:

1. Assemble & Start Simulator:

    asm21k -DTEST ratiobuf
    ld21k -a generic ratiobuf
    sim21k -a generic -e ratiobuf -k port1

    Note: the keystroke file port1.ksf opens two ports:
    1st port - DM, F0000000, Fixed-Pt., auto-wrap, wave.1
    2nd port - DM, F0000001, Fixed-Pt., no wrap,   out.1

2. In the simulator,
   a. Go to the program memory (disassembled) window,
   d. Go to symbol input, set to break on 513th occurrence, and
   c. Run

3. Compare wave.1 to out.1 on the graphing program of your choice.

*———————————————————————————————————————*/

#include "def21020.h"
#define N               32      /* N taps, N coefficients               */
#define L               4       /* interpolate by factor of L           */
#define NoverL          8       /* N must be an integer multiple of L   */
#define M               3       /* decimate by factor of M              */
#define intMoverL       2       /* smallest integer GE M/L              */
#define FP              20.0e6  /* Processor Frequency = 20MHz          */
#define FI              64.0e3  /* Interrupt Frequency = 64KHz          */

#define IN_CNTR         r2      /* input counter register               */
#define OUT_CNTR        r3      /* output counter register              */

#ifdef TEST
/*—————————————————————————————*/
#define TPER_VAL 312        /*  TPER_VAL = FP/FI - 1 */
#else
/*—————————————————————————————*/
#define TPER_VAL 39         /* interrupt every 40 cycles */
#endif
/*—————————————————————————————*/

.SELEMENT /pm    pm_data;
.VAR    coef[N]=”coef.dat”;     /* coefficient circular buffer */
.ENDSEG;

.SELEMENT /dm    dm_data;
.VAR    data[NoverL];           /* data delay line */
.VAR    input_buf[intMoverL];   /* input buffer is not circular */
.ENDSEG;

.SELEMENT /dm ports;
.PORT   adc;                /* Analog to Digital (input) converter */
.PORT   dac;                /* Digital to Analog (output) converter */
.ENDSEG;

/*—————————————————————————————
RESET Service Routine
—————————————————————————————*/

.SELEMENT /pm    rst_svc;
rst_svc:        PMWAIT=0x0021;      /* no wait states,internal ack only */
DMWAIT=0x8421;      /* no wait states,internal ack only */
jump init_rat;      /* initialize the test case */
.ENDSEG;

/*—————————————————————————————
Timer=zero high priority Service Routine
—————————————————————————————*/

.SELEMENT /pm    tmzh_svc;
tmzh_svc:       IN_CNTR=IN_CNTR-1;          /* test if time for input */
    if ne jump skipin(db);
    OUT_CNTR=OUT_CNTR-1; /* test if time for output */
    nop;
    jump input (db); /* calculate the input sample */
    r0=dm(adc);     /* get input sample */
    r0=ashift r0 by -16; /* right-justify the data */
.ENDSEG;

/*—————————————————————————————
Software Interrupt 0 Service Routine
—————————————————————————————*/

.SELEMENT /pm    sft0_svc;
sft0_svc:       jump calc_out (db);       /* calculate the output sample */
    r0=17;             /* get current ptr to input buffer */
    r1=input_buf;     /* get start addr of input buffer */
.ENDSEG;

(listing continues on next page)
5 Multirate Filters

/* Initialize Interpolation Routine */

.SEGMENT /pm pm_code;
init_rat:
    /* initialize buffer index registers */
b0=data; l0=data; m0=1; /* data buffer */
b7=input_buf; l7=0; /* input buffer */
b8=coef; l8=coef; m8=L; /* modifier for coef is L! */
b9=0; l9=0; m9=-M; /* track coefficient updates */

IN_CNTR = 1; /* initialize flags, counters */
OUT_CNTR = 1;
r5=L; /* scale register for interp. */
f5=float r5;
r14=16; /* exponent scale factor */

r0=0; /* clear data buffer */
lcntr=NoverL, do clrbuf until lce;
clrbuf:

   tperiod=TPER_VAL; /* program timer */
tcount=TPER_VAL;
bit set mode2 TIMEN;

   /* enable timer zero high priority, software 0 interrupts */
bit set imask TMZHI|SFT0I;
bit clr irptl TMZHI|SFT0I; /* clear any pending irpts */
bit set model IRPTEN|ALUSAT; /* enable irpts, ALU sat */

wait:
   idle;
   jump wait; /* infinite wait loop */

.ENDSEG;

/* Calculate output */

_calculate output initiated by software interrupt 0, code below occurs at the output sample rate */

.SEGMENT /pm pm_code;
calc_out:
    r1=r0-r1, i7=b7; /* calculate amount in inbuffer */
    if eq jump modify_coef; /* skip do loop if buffer empty */
    modify(i9,-M); /* modify coef update by -M */
    m10=i9;

    /* dump the input buffer into delay line if L inputs have */
    /* been acquired */
lcntr=r1, do load_data until lce;
    f1=dm(17, m0);
load_data:
   dm(10, m0)=f1;
modify_coef: modify(i8, m10); /* modify coef ptr by coef update */
Multirate Filters

Listing 5.3  ratiobuf.asm

/ * filter pass, occurs at L times the input sample rate */
   
   fir:
       f0=dm(i0,m0), f4=pm(i8,m8);
       f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
       f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
   
   lcntr=NoverL-3, do taps until lce;
   
   taps:
       f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
       f12=f0*f4, f8=f8+f12;
       f0=f8+f12, i7=b7; /* reset input buffer in || w/comp */
   
       i9=0; /* reset coef update */
       rti (db); 
       r0 = fix f0 by r14; /* float -> fixed */
       dm(dac)=r0; /* output data sample */
   .ENDSEG;

/ *———————————————————————————————————*/

Acquire input

initiated by Timer Interrupt, code occurs at L times the input rate

/ *———————————————————————————————————*/

SEGMENT /pm  pm_code;

input:

    /* convert fixed->float, modify the coef update by L */
    f0=float r0, modify(i9,m8);
    f0=f0*f5; /* scale by L */
    dm(i7,m0)=f0; /* load in M long buffer */
    /* set A2 flag w/ OUT_CNTR, reset the input count to L */
    OUT_CNTR=pass OUT_CNTR, IN_CNTR=m8;

skipin:

    if ne rti; /* return if OUT_CNTR!=0 */

output:

    rti(db); /* cause output routine if OUT_CNTR==0 */
    bit set irptl SFT0I; 
    OUT_CNTR=m8; /* reset output counter to M */

.ENDSEG;
5.4 RATIONAL RATE CHANGER (EXTERNAL INTERRUPT-BASED)

The rational rate changer shown in `rat2int.asm` (Listing 5.4) performs the same operation as the rational rate changer (counter based) algorithm. The interrupt based algorithm, however, uses two external interrupts instead of the counters to generate the proper interpolation and decimation interrupts.

5.4.1 Implementation

The IRQ3 interrupt service routine performs the input function of the rational rate changer. An external timer is necessary to cause this interrupt to occur at the input sample rate. This interrupt replaces the timer interrupt and `IN_CNTR` in the counter based changer. The IRQ2 interrupt service routine performs the output function of the rational rate changer. Again, another external timer is necessary to cause the interrupt to occur at the output sample rate. This interrupt routine replaces the User Software Interrupt 0 routine and `OUT_CNTR` in the counter based changer.

Other aspects of the code, initialization of buffers, modifying coefficient pointers, and filter calculations, are executed in the same manner as in the previous example.
5.4.2 Code Listing–rat_2_int.asm

/************************************************************************
File Name
   RAT_2INT.ASM

Version
   3/4/91

Purpose
   Interrupt-Driven Real Time Rational Rate Changer

Calling Parameters
   Input:  adc
      r0 = ADC input data

Return Values
   Output: dac
      r0 = DAC output data

Registers Affected
   r0 = FIR data in / ADC data / temp register
   r1 = temp register
   r4 = FIR coefficients
   r5 = scale value L
   r8 = FIR accumulate result reg
   r12 = FIR multiply result reg / temporary reg
   r14 = 16 = exponent scale factor

Start Labels
   init_rat    reset-time initialization
   irq3_svc    IRQ3 interrupt service routine for input
   output      called by IRQ2 interrupt service routine

Computation Time
   irq2_svc = 6
   irq3_svc = 3 + output
   output    >= NoverL + 14                     ( Notes: 1,3 )
            <= NoverL + 2*intMoverL + 13         ( Notes: 2,3 )

Notes:
   1. w/ no input samples to transfer
   2. w/max input samples to transfer
   3. maximum of 5 cache misses on the 1st pass

# PM Locations
   53 Words Code, N Words Data

# DM Locations
   N/L + M/L Words Data
************************************************************************/
(listing continues on next page)
Creating & Using the Test Case:

Running the test case requires two data files:
  coef.dat  32-tap FIR filter coefficients (floating point)
  wave.1    waveform data file

Descriptions of coef.dat and wave.1 can be found in decimate.asm & interpol.asm.

The test case writes the rate-changed output to a dac port. Since there are 512 samples in wave.1, and the test case performs interpolation by 4 followed by decimation by 3, there will be 682 data values written to the dac port if all data samples are read in and processed. The values written out are left-shifted by 16 bits in parallel with the float->fixed conversion, based again on the assumption that the D/A converter port is located in the upper 16 bits of data memory.

Armed with this information, you are ready to run:

1. Assemble & Start Simulator:

   asm21k -DTEST rat_2int
   ld21k -a generic rat_2int
   sim21k -a generic -e rat_2int

2. In the simulator,
   a. Run keystroke file setint.ksf, which programs periodic interrupts,
   a. Go to the program memory (disassembled) window,
   d. Go to symbol irq3_svc, set to break on 513th occurrence, and
   c. Run

3. Compare wave.1 to out.1 on the graphing program of your choice.

#include "def21020.h"
#define N               32      /* N taps, N coefficients */
#define L               4       /* interpolate by factor of L */
#define NoverL          8       /* N must be an integer multiple of L */
#define M               3       /* decimate by factor of M */
#define intMoverL       2       /* smallest integer GE M/L */

.SEGMENT /pm pm_data;
.VAR    coef[N]="coef.dat";  /* coefficient circular buffer */
.ENDSEG;

.SEGMENT /dm dm_data;
.VAR    data[NoverL];        /* data delay line */
.VAR    input_buf[intMoverL]; /* input buffer is not circular */
.ENDSEG;

.SEGMENT /dm ports;
.PORT   adc;                  /* Analog to Digital (input) converter */
.PORT dac; /* Digital to Analog (output) converter */
.ENDSEG;

/*__________________________________________________________________________*/
RESET Service Routine
__________________________________________________________________________*/
.SEGMENT /pm rst_svc;
rst_svc:        PMWAIT=0x0021;      /* no wait states, internal ack only */
DMWAIT=0x8421;      /* no wait states, internal ack only */
jump init_rat; /* initialize the test case */
.ENDSEG;

/*__________________________________________________________________________*/
IRQ3 Interrupt Service Routine
occurs at input rate
__________________________________________________________________________*/
.SEGMENT /pm irq3_svc;
irq3_svc:       r0=dm(adc);             /* get input sample */
r0=ashift r0 by -16;    /* right-justify the data */
/* convert fixed->float, modify the coef update by L */
f0=float r0,modify(i9,m8);
rti(db);
     f0=f0*f5;            /* scale by L */
     dm(i7,m0)=f0;       /* load in input buffer */
.ENDSEG;

/*__________________________________________________________________________*/
IRQ2 Interrupt Service Routine
occurs at output rate
__________________________________________________________________________*/
.SEGMENT /pm irq2_svc;
irq2_svc:       jump output (db); /* go to the output routine */
r0=i7;              /* get input buffer pointer */
r1=input_buf;       /* get start addr of input buffer */
.ENDSEG;

/*__________________________________________________________________________*/
Initialize Interpolation Routine
__________________________________________________________________________*/
.SEGMENT /pm pm_code;
init_rat:       b0=data; 10=@data; m0=1;    /* data buffer */
b7=input_buf; 17=0;        /* input buffer */
b8=coef; 18=@coef; m8=L;    /* modifier for coef is L! */
b9=0;19=0;m9=-M;         /* track coefficient updates */
r5=L;            /* scale reg for interpolator */
f5=float r5; /* fix -> float conversion */
r14=16;        /* exponent scale factor */
     r0=0;            /* clear data buffer */
lcntr=NoverL, do clrbuf until lce;
clrbuf: dm(i0,m0)=r0;

(listing continues on next page)
5 Multirate Filters

/* enable timer zero high priority, software 0 interrupts */
bit set imask IRQ3I|IRQ2I;

bit clr irpt1 IRQ3I|IRQ2I;       /* clear any pending irpts */
bit set model IRPTEN|ALUSAT;     /* enable interrupts, ALU sat*/

wait:           idle;                           /* infinite wait loop */
                 jump wait;

.ENDSEG;

Calculate output

initiated by IRQ2, occurs at output sample rate

.SEgment /pm pm_code;
output:         r1=r0-r1,i7=b7;  /* r1 = # samples in input buffer */
                 /* also reset input buffer */
                 
                 if eq jump mod_coef(db);/* skip do loop if buffer empty */
                 modify(i9,-M);    /* modify coef update by -M */
                 m10=i9;
                 
                 /* dump the buffer into delay line */
                 lcntr=r1, do load_data until lce;
                 f1=dm(i7,m0);
load_data:          dm(i0,m0)=f1;

mod_coef:         modify(i8,m10);   /* modify coef ptr by coef update */

/* filter pass, occurs at L times the input sample rate */

fir:                       f0=dm(i0,m0), f4=pm(i8,m8);
                     f8=f0*f4,  f0=dm(i0,m0), f4=pm(i8,m8);
                     f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                 lcntr=NoverL-3, do taps until lce;

                 /* reset input buffer in || w/comp */
                 i9=0;
                 /* reset coef update */
                 rti (db);
                 
                 r0 = fix f0 by r14;  /* float -> fixed */
                 dm(dac)=r0;       /* output data sample */

.ENDSEG;

Listing 5.4 rat2int.asm
5.5 TWO-STAGE DECIMATION FILTER

In the previous example of a single-stage decimation filter, a single low-pass filter is used to prevent the aliasing that otherwise occurs with the rate compression. The filter presented here uses two stages to reduce the number of calculations required with no loss of precision.

If the output frequency from the decimation filter is much smaller than the input oversampled frequency, the transition band of the filter needs to be very small. A small transition band requires a large number of taps, which increases the computation time.

By cascading filters, the number of taps per stage can be reduced, thereby increasing the computational efficiency. For example, the first stage filter may not reduce the input sampling rate by very much, but the filter has a very large transition band requiring fewer taps. The second stage filter may then require a smaller transition band, but the output sample rate is smaller as well, still only needing a few taps. This two-stage implementation can be easily extended to more stages. See [CROCH83] for design curves to help determine optimal rate changed factors for intermediate stages. A two- or three-stage design provides a substantial reduction in filter computations; further reductions in computations come from adding more stages. Because the filters are cascaded, each stage must have a passband ripple equal to the final passband ripple divided by the number of stages used. The stopband ripple does not have to be less than the final stopband ripple because each successive stage attenuates the stopband ripple of the previous stage.

5.5.1 Implementation

This example (dec2stg.asm, Listing 5.5) uses two real time direct-form FIR filters of $N_1$ and $N_2$ taps to decimate by $M_1 \times M_2$ for a decrease of $1 / (M_1 \times M_2)$ times the input sample rate. The use of an input buffer allows the filter computations to proceed in parallel with the acquisition of the next $M_1 \times M_2$ inputs, allowing a larger order filter than if all calculations are made between samples.

The timer interrupt is set at a rate equal to the input sample rate to store all the incoming values in the input buffer. A counter (CNTR) is set to $M_1 \times M_2$. This allows the input buffer to have enough samples for both stages of filters. When the counter expires, User Service Interrupt 0 (USI0) is set. The USI0 interrupt service routine calls the dec2stg routine that performs the data transfer and two-stage filter.
The length of the input buffer is twice as long as the decimation rate (that is, $2 \times M_1 \times M_2$). This double-length buffer acts as two separate input buffers. While one buffer is being transferred to the data buffer for filtering, the second buffer stores values read in from the A/D. This double buffering is accomplished by utilizing two pointers (I7 and I6) that are offset from each other by $M_1 \times M_2$. The buffer is circular, so that the pointers “ping-pong” between being the input pointer and the transfer pointer.

Since the input buffer is twice as long as needed, the input samples are now stored in the lower half of the input buffer with pointer I7, while the transfer from the input buffer to the data buffer is done with pointer I6. Since the double-buffer is circular, after the first time through the stages, the pointers “ping-pong,” reversing roles. For more details, see [ADI90].

Because $M_1$ is less than three for this test case, the movement of data from the input buffer to the buffer data1 is put at the start of the subroutine dec2stg as straight-line (i.e., not looped) code. If $M_1$ is greater than three, the following code is used:

```c
r12=dm(i6,m0);
r4=ashift r12 by -16;
f0=float r4, r12=dm(i6,m0);
lcnt=M_1-2, do load_data until lce;
r4=ashift r12 by -16, dm(i0,m0)=f0;
load_data:    f0=float r4, r12=dm(i6,m0);
r4=ashift r12 by -16, dm(i0,m0)=f0;
f0=float r4;
dm(i0,m0)=f0;
```

The filter operations and coefficient interleaving are performed in the same manner as with the single-stage filters.
5.5.2 Code Listing—dec2stg.asm

/************************************************************************
File Name
    DEC2STG.ASM

Version
    3/18/91

Purpose
    Two Stage Decimator

Calling Parameters
    Input: adc
    r15 = ADC input data

Return Values
    Output: dac
    r0 = DAC output data

Registers Affected
    r0 = FIR data in / ADC fltg-pt data
    r3 = counter
    r4 = FIR coefficients
    r8 = FIR accumulate result reg
    r12 = FIR multiply result reg / temporary reg
    r14 = 16 = exponent scale factor

Start Labels:
    init_dec    reset-time initialization
    dec2stg     called by software 0 interrupt

Computation Time:
    sft0_svc = N_2 + 10 + M_2*(N_1 + 2*M_1 + 7)
    tmzh_svc = 7

# PM Locations
    73 Words Code, N_1 + N_2 Words Data

# DM Locations
    N_1 + N_2 + M_1*M_2*2 Words
/************************************************************************/

(listing continues on next page)
5 Multirate Filters

Creating & Using the Test Case:

Running the test case requires two data files:
- `coef1.dat`: 32-tap FIR filter #1 coefficients (floating point)
- `coef2.dat`: 128-tap FIR filter #2 coefficients (floating point)
- `wave.1`: waveform data file

`coef1` & `coef2` are the two Parks-McClellan FIR filters used in this two-stage decimator. Assume that the input sample rate is 192kHz, and the decimated output is 192/4 = 48kHz. The first FIR filter has 32 taps, and is designed to have maximum of 1dB of attenuation from 0 - 48kHz, then >90dB attenuation at >96kHz. The second FIR filter has 128 taps, and is designed to have maximum of 1dB of attenuation from 0 - 20kHz, then >90dB attenuation at >24kHz.

Descriptions of `wave.1` can be found in `decimate.asm` & `interpol.asm`.

The test case writes the decimated output to a dac port. Since there are 512 samples in `wave.1`, and the test case performs decimation by 4, there will be 128 data values written to the dac port if all data samples are read in and processed. The values written out are left-shifted by 16 bits in parallel with the float—>fixed conversion, based again on the assumption that the D/A converter port is located in the upper 16 bits of data memory.

Armed with this information, you are ready to run:

1. Assemble & Start Simulator:
   ```
   asm21k -DTEST dec2stg
   ld21k -a generic dec2stg
   sim21k -a generic -e dec2stg
   ```

2. In the simulator,
   a. Go to the program memory (disassembled) window,
   d. Go to symbol output, set to break on 129th occurrence, and
   c. Run

3. Compare `wave.1` to `out.1` on the graphing program of your choice.

```c
#include "def21020.h"
#define N_1     32          /* number of taps in FIR #1                    */
#define N_2     128         /* number of taps in FIR #2                    */
#define M_1     2           /* 1st decimation factor                       */
#define M_2     2           /* 2nd decimation factor                       */
#define CNTR    r3          /* counter                                     */
#define FP      20.0e6      /* Processor Frequency = 20MHz                 */
#define FI      192.0e3     /* Input Frequency = 192KHz                   */

#ifndef TEST    /*—————————————————————————————*/
#define TPER_VAL 312        /* TPER_VAL = FP/FI - 1                        */
#else           /*—————————————————————————————*/
#define TPER_VAL 104        /* interrupt every 160 cycles                  */
#endif          /*—————————————————————————————*/
```
Multirate Filters

```
.SEGMENT /dm dm_data;
.VAR data1[N_1];       /* FIR input data #1 */
.VAR data2[N_2];       /* FIR input data #2 */
.VAR input_buf[M_1*M_2*2]; /* raw ADC data */
.ENDSEG;

.SEGMENT /pm pm_data;
.VAR coef1[N_1]="coef1.dat";   /* FIR floating-point coefficients */
.VAR coef2[N_2]="coef2.dat";   /* FIR floating-point coefficients */
.ENDSEG;

.SEGMENT /dm ports;
.PORT adc;                /* Analog to Digital (input) converter */
.PORT dac;                /* Digital to Analog (output) converter */
.ENDSEG;

/*———————————RESET Service Routine———————————*/
.SEGMENT /pm rst_svc;
rst_svc:        PMWAIT=0x0021;       /* no wait states,internal ack only */
.DMWAIT=0x8421;       /* no wait states,internal ack only */
call init_dec;      /* initialize the test case */
wait:           idle;               /* infinite wait loop */
jump wait;
.ENDSEG;

/*———————————TMZH Service Routine———————————*/
.SEGMENT /pm tmzh_svc;
/* sample input: this code occurs at the sample rate */
tmzh_svc:       CNTR=CNTR-1,r15=dm(i2,m2);
    if eq jump zero_yes;
zero_no:        rti(db);
    nop;
    dm(i7,m0)=r15;       /* load in input buffer */
zero_yes:       rti(db);
    irptl=SFT0I;        /* if counter==0, set interrupt */
    dm(i7,m0)=r15;      /* load in input buffer */
```
5  Multirate Filters

```c
/*
.ENDSEG;

/*]
Interrupt Request 3 Service Routine
Dummy procedure so that the TMZH Service Routine does not fetch
instructions from empty memory
*/
.SEGMENT /pm irq3_svc;
/* dummy procedure */
irq3_svc:       rti;
.ENDSEG;

/*]
Software Interrupt 0 Service Routine
*/
.SEGMENT /pm sft0_svc;
/* process input data: this code occurs at 1/M times the sample rate */
sft0_svc:       jump dec2stg (db);       /* call the 2-stage decimator */
                CNTR=M_1*M_2;       /* reset input counter */
                nop;
.ENDSEG;

/*]
efficient decimator initialization
*/
.SEGMENT /pm pm_code;
init_dec:       b0=data1; 10=@data1; m0=1;  /* data buffer #1 */
                b1=data2; 11=@data2;  /* data buffer #2 */
                b8=coef1; 18=@coef1; m8=1;  /* coefficient buffer #1 */
                b9=coef2; 19=@coef2;  /* coefficient buffer #2 */
                b7=input_buf; 17=@input_buf;  /* input buffer sample ptr */
                b6=input_buf; 16=@input_buf;  /* inp. buffer working ptr */
                b2=adc; 12=1; m2=0;  /* set A/D conv. pointer */
                r14=16;  /* exponent scale factor */
```
Multirate Filters

CNTR=M_1*M_2; /* set input counter */

r0=0; /* clear data buffers */
lcntr=N_1, do clrbuf1 until lce;
clrbuf1:
dm(i0,m0)=r0;
lcntr=N_2, do clrbuf2 until lce;
clrbuf2:
dm(i1,m0)=r0;

/* program timer */
tperiod=TPER_VAL;
tcount=TPER_VAL;
bit set mode2 TIMEN;
bit clr irpt1 TMZHI|SFT0I; /* clear any pending irpts */

/* enable Timer high priority, software 0 interrupts */
bit set imask TMZHI|SFT0I;
/* enable interrupts, nesting, ALU saturation */
bit set model IRPTEN|NESTM|ALUSAT;

/* ————————————————————————————————————————
efficient decimator routine executes at 1/(M_1*M_2) times the sample rate ————————————————————————————————————————*/

.pdf SEGMENT /pm pm_code;
dec2stg: lcntr=M_2, do stage_1 until lce;
/* transfer input samples to the FIR data memory space */
/* using the input buffer working pointer. Perform */
/* fix->float conversion in parallel */
r12=dm(i6,m0);
r4=ashift r12 by -16;
f0=float r4, r12=dm(i6,m0);
r4=ashift r12 by -16, dm(i0,m0)=f0;
f0=float r4;
dm(i0,m0)=f0;
/* 1st FIR filter */
5  Multirate Filters

fir_1:                     f0=dm(i0,m0), f4=pm(i8,m8);
  f8=f0*f4,  f0=dm(i0,m0), f4=pm(i8,m8);
  f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
  lcntr=N_1-3, do taps_1 until lce;
taps_1:             f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
  f12=f0*f4, f8=f8+f12;
  f0=f8+f12;
stage_1:        dm(i1,m0)=f0;   /* output to next filter data buffer */

  /* 2nd FIR filter */
  fir_2:                 f0=dm(i1,m0), f4=pm(i9,m8);
  f8=f0*f4,  f0=dm(i1,m0), f4=pm(i9,m8);
  f12=f0*f4, f0=dm(i1,m0), f4=pm(i9,m8);
  lcntr=N_2-3, do taps_2 until lce;
taps_2:         f12=f0*f4, f8=f8+f12, f0=dm(i1,m0), f4=pm(i9,m8);
  f12=f0*f4, f8=f8+f12;
  f0=f8+f12;
  rti (db);
    r0 = fix f0 by r14;     /* float —> fix */
  output:         dm(dac)=r0;             /* output data sample */
  .ENDSEG;

Listing 5.5  dec2stg.asm

5.6  TWO-STAGE INTERPOLATION FILTER
The two-stage interpolation filter (Listing 5.6) uses two real time N-tap direct form FIR filters to interpolate by \( L_1 \times L_2 \) for an increase of \( L_1 \times L_2 \) times the input sample rate. The number of taps is restricted such that \( \frac{N}{L_1 \times L_2} \) must be an integer. The internal timer is used to generate interrupts at \( L_1 \times L_2 \) times the sample rate. Two counters are set up so that the delay line is only updated when a sample is ready at the ADC, that is, both counters have expired.

5.6.1  Implementation
Two delay lines are used, one for each filter stage. The delay line for the first stage filter, \( int2stg \), is loaded from the ADC. The delay line for the
second stage filter, `do_fir2`, is loaded from the output of the first stage filter.

After the initialization of the buffers, the main program sits in an idle loop waiting for timer interrupts. The timer interrupts occur at an interval of $L_1 \times L_2$ times the sample rate. In the timer service routine, the counters for each stage are decremented. The counter with the smaller interpolation factor is decremented first (CNT2=2). If the counter has not expired, the service routine jumps to the `do_fir2` routine. Therefore, the second stage of the filter is executed at an interval of $L_1 \times L_2$ times the input sample rate. If the counter has expired, the routine decrements the second counter.

The second counter is set to the higher interpolation factor (CNT=4). If the counter has not expired, the service routine jumps to the `int2stg` routine. This routine is called at an interval of $L_1$ times the input sample rate. When the `int2stg` routine is called, both the first stage and second stage filtering is performed.

If both counters have expired, the next input value is read from the ADC and then both filter stages are performed. The filter computations and coefficient interleaving are performed in the same manner as the single-stage interpolation filter.

### 5.6.2 Code Listing–int2stg.asm

```asm
/**************************************************************************
File Name
INT2STG.ASM

Version
3/18/91

Purpose
Two Stage Interpolator

Calling Parameters
Input: adc
r15 = ADC input data

Return Values

```
Multirate Filters

Output: dac
r0 = DAC output data

Registers Affected
r1 = scale factor L
r2 = counter #1
r3 = counter #2
r4 = FIR coefficients
r8 = FIR accumulate result reg
r12 = FIR multiply result reg / temporary reg
r13 = 1 = counter compare register
r14 = 16 = exponent scale factor

Start Labels
init_int      reset-time initialization
int2stg       called by Timer Zero High Priority interrupt

Computation Time
tmzh_svc      = N_1/L_1 + N_2/L_2 + 25

# PM Locations
67 Words Code, N_1 + N_2 Words Data

# DM Locations
N_1/L_1 + N_2/L_2 Words Data

Creating & Using the Test Case:

Running the test case requires two data files:

coef.dat        32-tap FIR filter coefficients (floating point)
sinX.dat        sine-wave data files (X=1,2,3,4,5)

coef.dat contains a 32-tap FIR filter which bandlimits the input signal to 1/8 of the input frequency. Since the decimator in the test case decimates by a factor M=4, this bandlimitation is equivalent to the required limit of 1/2 the output frequency. The filter is a Parks-McLellan filter with a passband ripple of 1.5dB out to about 1/20 the input frequency, and a stopband with greater than 70dB of attenuation above 1/8 the input frequency.

As an example, if the oversampled input frequency is 64kHz, the passband extends out to about 3.2kHz. The stopband starts at 4kHz, and the output frequency is 16kHz.

The data files are all of the form sin.X, where X ranges from 1 to 5. Each data file contains 512 signed fixed-point data values in the range +/- 32767. These data points are meant to resemble data read from a 16-bit A/D converter which
generates signed data. sin.1 through sin.4 contain simple sine waves at different frequencies. sin.5 contains a composite waveform generated from 3 sine wave of differing frequency and magnitude.

The test case writes the interpolated output to a dac port. Since there are 512 samples in sin.X, and the test case performs decimation by 4, there will be 128 data values written to the dac port if all data samples are read in and processed.

Armed with this information, you are ready to run:

1. Assemble & Start Simulator:
   asm21k -DTEST int2stg
   ld21k -a generic.ach int2stg
   sim21k -a generic.ach -e int2stg

2. In the simulator,
   a. Go to the program memory (disassembled) window,
   b. Go to symbol output, set to break on 4097th occurrence, and
   c. Run

3. Compare wave.1 to out.1 on the graphing program of your choice.

#include "def21020.h"
define N_1 32  /* number of taps, FIR #1 */
define N_2 32  /* number of taps, FIR #2 */
define L_1 4   /* interpolate by factor of L_1 */
define L_2 2   /* interpolate by factor of L_2 */
define CNT1 r2
#define CNT2 r3
#define FP 20.0e6  /* Processor Frequency = 20MHz */
define FI 64.0e3  /* Input Frequency = 64KHz */
#ifdef TEST  /*------------------------------------------*/
define TPER_VAL 312  /* TPER_VAL = FP/FI - 1 */
#else  /*------------------------------------------*/
define TPER_VAL 53   /* interrupt every 54 cycles */
#endif  /*------------------------------------------*/

.SEGMENT /dm       dm_data;
5 Multirate Filters

```plaintext
.VAR data1[N_1/L_1]; /* ADC fixed-point data buffer #1 */
.VAR data2[N_2/L_2]; /* ADC fixed-point data buffer #2 */
.ENDSEG;

.SECTION /pm pm_data;
.VAR coef1[N_1]="coef1.dat"; /* fltg-pt. FIR #1 coefficients */
.VAR coef2[N_2]="coef2.dat"; /* fltg-pt. FIR #2 coefficients */
.ENDSEG;

.SECTION /dm ports;
.PORT adc; /* Analog to Digital (input) converter */
.PORT dac; /* Digital to Analog (output) converter */
.ENDSEG;

/* ————————————————————————————
RESET Service Routine
——————————————————————————— */

.SECTION /pm rst_svc;
rst_svc: PMWAIT=0x0021; /* no wait states, internal ack only */
        DMWAIT=0x8421; /* no wait states, internal ack only */
        jump init_int; /* initialize the test case */
.ENDSEG;

/* ————————————————————————————
TMZH Service Routine
——————————————————————————— */

.SECTION /pm tmzh_svc;
/* sample input: this interrupt occurs at the L_1*L_2*input rate */
tmzh_svc: CNT2=CNT2-1,modify(i9,m15); /* decrement counter, and */
         /* shift coef #2 pointer back */
        if ne jump do_fir2; /* do second FIR if CNT2!=0 */
         CNT1=CNT1-1,modify(i8,m15); /* decrement counter, and */
         /* shift coef #1 pointer back */
        if ne jump int2stg; /* go to first FIR if CNT1!=0 */
         jump sample (db); /* get sample if CNT1==CNT1==0 */
         r15=dm(adc); /* input data sample */
         r15=ashift r15 by -16; /* right-justify & zero garbage bits */
.ENDSEG;

/* ————————————————————————————
Initialize Interpolation Routine
——————————————————————————— */

.SECTION /pm pm_code;
init_int: /* initialize buffer index registers */
         b0=data1; lo=datal; m0=1; /* data buffer #1 */
         b1=data2; li=data2; /* data buffer #2 */
         b8=coef1; 18=coef1; m8=L_1; /* modifier for coef = L_1 */
         b9=coef2; 19=coef2; m9=L_2; /* modifier for coef = L_2 */
         m15=-1;

         /* Register Initialization */
         r1=L_1*L_2; /* value for upscaling sample */
         f1=float r1; /* fix -> float conversion */
```
Multirate Filters

CNT1=1;                  /* set interpolate cntrs to 1        */  
CNT2=1;                  /*    for first data sample          */  
r13=1;                   /* counter compare reg               */  
r14=16;                  /* exponent scale factor             */  

r0=0;                    /* clear data buffers                  */  

lcntr=N_1/L_1, do clrbuf1 until lce;
clrbuf1:            dm(i0,m0)=r0;

lcntr=N_2/L_2, do clrbuf2 until lce;
clrbuf2:            dm(i1,m0)=r0;

tperiod=TPER_VAL;       /* program timer                             */  
tcount=TPER_VAL;

bit set imask TMZHI;    /* enable TMZH interrupt                    */  
bit clr irptl TMZHI;    /* clear any pending irpts                */  

/* enable interrupts, ALU sat */
bit set mode1 IRPTEN|ALUSAT;
bit set mode2 TIMEN;    /* turn timer on                             */  

wait_interrupt: idle;
jump wait_interrupt;    /* infinite wait loop                       */  

.ENDSEG;

/*———————————————————————————————————————
Two-stage Interpolate
   code executes at L_1*L_2*(sample_rate)
———————————————————————————————————————*/

.SEGMENT /pm    pm_code;

int2stg:    /* filter pass, occurs at L_1 times the input sample rate */
            f0=dm(i0,m0), f4=pm(i8,m8);
            f8=f0*f4,  f0=dm(i0,m0), f4=pm(i8,m8);
            f12=f0*f4,  f0=dm(i0,m0), f4=pm(i8,m8);
            lcntr=N_1/L_1-3, do taps1 until lce;
            
taps1:          f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
            f12=f0*f4, f8=f8+f12;
            f0=f8+f12;
            dm(i1,m0)=f0;
            modify(i9,m9);
            /* reset counter #2 to L_2               */  

CNT2=m9;

/* filter pass, occurs at L_1*L_2 times the input sample rate */
do_fir2:               f0=dm(i1,m0), f4=pm(i9,m9);
            f8=f0*f4,  f0=dm(i1,m0), f4=pm(i9,m9);
            f12=f0*f4,  f0=dm(i1,m0), f4=pm(i9,m9);
            lcntr=N_2/L_2-3, do taps2 until lce;
            
taps2:          f12=f0*f4, f8=f8+f12, f0=dm(i1,m0), f4=pm(i9,m9);
5 Multirate Filters

\[ f_{12} = f_0 \times f_4, \quad f_8 = f_8 + f_{12}; \]
\[ f_0 = f_8 + f_{12}; \]
\[ dm(i_1, m_0) = f_0; \]
\[ rti (db); \]
\[ r_0 = \text{fix} \ f_0 \text{ by } r_{14}; \quad /* \text{float} \rightarrow \text{fixed} */ \]
\[ \text{output: } \quad dm(dac) = r_0; \quad /* \text{output data sample} */ \]
\[ \text{.ENDSEG;} \]

/* Acquire Sample */

.SEGMENT /pm pm_code;
/* sample input: this code occurs at the input rate */
sample: /* do fix->float, and scale data up by \( L_1 \times L_2 \) */
\[ f_0 = \text{float} \ r_{15}, \text{modify}(i_8, m_8); \]
\[ f_0 = f_0 \times f_1; \quad /* \text{upscale sample} */ \]

/*
jump int2stg (db);
\[ dm(i_0, m_0) = f_0; \quad /* \text{update delay line with latest} */ \]
*/
CNT1 = m_8; /* reset counter #1 to \( L_1 \) */

.ENDSEG;

Listing 5.6 int2stg.asm

5.7 REFERENCES
