Lattice wave digital filter test
and performance verification

By Andrea Vitali

Main components

<table>
<thead>
<tr>
<th>STM32L476xx</th>
<th>STM32L486xx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra-low-power Arm® Cortex®-M4 32-bit MCU+FPU, 100 DMIPS, up to 1 Mbyte Flash, 128 Kbytes SRAM, USB OTG FS, LCD, analog, audio</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STM32F411xx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arm® Cortex®-M4 32-bit MCU+FPU, 125 DMIPS, 512 Kbytes Flash, 128 Kbytes RAM, USB OTG FS, 11 TIMs, 1 ADC, 13 comm. interfaces</td>
</tr>
</tbody>
</table>

Purpose and benefits

This design tip explains how to test and verify the performance of lattice wave digital filters (LWDF) when the C source code implementation is available. In particular, the following procedures will be described:

- How to compute the frequency response from the gamma coefficients
- How to compute the frequency response from the impulse response
- How to verify the frequency response by filtering white noise
- How to verify the performance and stability of the filter implementation
- How to verify the performance of the interpolation/decimation implementation

Description

Wave digital filters have some notable advantages: excellent stability under non-linear operating conditions due to overflows and round-off errors, low coefficient word-length and good dynamic range.

An important subclass of lattice wave digital filters is made of filters with a bi-reciprocal transfer function: \( H(f) = 1 - H(F/2 - f) \), where \( F \) is the sampling frequency and \( f \) goes from 0 to \( F/2 \). Because of the symmetry, \( H(F/4) = 0.5 \) (-3dB), the pass-band attenuation is dependent on the stop-band attenuation and cannot be specified.

Bi-reciprocal filters have half as many adaptors as the usual lattice wave digital filter of the same order. Also, interpolation or decimation with a factor of two is very economical: the upper and lower branches work at the lowest frequency. For decimation, the input is distributed in a round-robin fashion to halve the frequency. For interpolation, the output is concatenated to double the frequency.
The companion design tip, DT0091, provides the design tool needed to generate the list of gamma coefficient and the C source code implementation.

How to compute the frequency response from the gamma coefficients

The following MATLAB® script imports the file with the gamma coefficients ("WDgamma.txt") and computes the frequency response of the filter, magnitude and phase. The group delay is also plotted.

How to compute and verify the frequency response using an impulse and noise

The following MATLAB® script probes the filter with an impulse. The output of the filter is then used to compute the frequency response. The script also probes the filter with white noise and the power spectrum of the output is plotted over the frequency response magnitude to verify that they match.
The script uses the following MATLAB® function which in turn calls the C utility shown below.

```matlab
function [outLP, outHP] = WDtest(in, iflag, fflag)
  if (flag==1), in1=in(1:2:end); in2=in(2:2:end); % decimator
  else
      in1=in; in2=in;
  end;
  h=fopen('WDin1.txt', 'w'); fprintf(h,fmt1,in1); fclose(h);
  h=fopen('WDin2.txt', 'w'); fprintf(h,fmt1,in2); fclose(h);
  out1=importdata('WDout1.txt'); L1=length(out1);
  out2=importdata('WDout2.txt'); L2=length(out2);
  if (L1==L2), fprintf('error!
'); return; end;
  if (iflag==1), % interpolator
      outLP(1:2:L1+L2)=out1; outLP(2:2:L1+L2)=out2;
  else % normal or decimator
      outLP(1:2)=out1/2; outHP=(out1-out2)/2;
  end;

  % unused... %

The C utility includes the source code of the filter implementation generated by the design tool ("WDfilter.c"). The utility reads two input files, which are then fed to the inputs of the lattice wave digital filter, and writes two output files taken from the outputs of the filter.

If the filter has a floating point implementation, this is the utility to be compiled:

```c
#include <stdio.h>
#include <stdlib.h>
#include "WDfilter.c"
#include <math.h>

int main(int argc, char *argv[]) { FILE *fin1, *fin2, *fout1, *fout2;
  float gain, ofs, in1, in2, out1, out2, cnt;
  for (fin1=fin2=fout1=fout2=NULL;;) {
    if (argc==4) { printf("usage: %s input1 input2 output1 output2
      offset gain \n",argv[0]); break; }
    else if (argc==5) { printf("error!
"); return; }
    if (argc>6) gain=atof(argv[6]);
  }
  if (argc>5)  ofs=atof(argv[5]); else ofs=0.0;
  if (argc<=4) { printf("usage: %s input1 input2 output1 output2
      offset gain \n",argv[0]); break; }
  else if (argc>4) gain=atof(argv[4]);

  system(sprintf("%s WDin1.txt WDin2.txt WDout1.txt WDout2.txt",
    argv[0])); L1=length(out1);
  out1=out1+out2; out2=out1;
  outLP=out1+out2; outHP=out1-out2;
  % unused... %

  % unused... %

  if (fout1!=NULL) fclose(fout1);
  if (fout2!=NULL) fclose(fout2);
  return 0;
}
```

If the filter has a fixed-point implementation, this is the utility to be compiled:

```c
#include <stdio.h>
#include <stdlib.h>
#include "WDfilter.c"
#include <inttypes.h>

int main(int argc, char *argv[]) { FILE *fin1, *fin2, *fout1, *fout2;
  int gain, ofs, in1, in2, out1, out2, cnt;
  for (fin1=fin2=fout1=fout2=NULL;;) {
    if (argc==4) { printf("usage: %s input1 input2 output1 output2
      offset gain \n",argv[0]); break; }
    else if (argc==5) { printf("error!
"); return; }
    if (argc>6) gain=atof(argv[6]);
  }
  if (argc>5)  ofs=atof(argv[5]); else ofs=0.0;
  if (argc<=4) { printf("usage: %s input1 input2 output1 output2
      offset gain \n",argv[0]); break; }
  else if (argc>4) gain=atof(argv[4]);

  outLP=out1+out2; outHP=out1-out2;
  % unused... %

  % unused... %

  if (fout1!=NULL) fclose(fout1);
  if (fout2!=NULL) fclose(fout2);
  return 0;
}
```
How to verify the performance and stability of the filter implementation

The following MATLAB® script uses sinusoids in the pass-band and in the stop-band to verify the pass-band ripples, the stop-band attenuation, the recovery from hard discontinuities, the magnitude of over/under-shoots following a step discontinuity, and the return to zero-output when the input goes to zero.

How to verify the performance of interpolation/decimation

The following MATLAB® script uses a sinusoid to verify the interpolation by a factor of two. Interpolation is usually done by inserting one zero after every other sample and then filtering. In the frequency domain, both the low and high frequency interpolated signals are plotted. In the time domain, the low frequency interpolated signal is plotted over the original signal to verify that they match.
Examples

This is the floating-point implementation for the Elliptic/Cauer filter, order 7:

```matlab
%-- test
in=A*sin(2*pi*t/[0:NFFT-1]/Fs);
[outLP,outHP]=WDecest(in,iFlag,eFlag);
NOVL=round(NFFT*0.5); win=hann(NFFT);
[pLP,pf]=pwelch(outLP,win,NOVL,NFFT,Fs/2);
[pHP,pf]=pwelch(outHP,win,NOVL,NFFT,Fs/2);

%-- plot
figure; subplot(2,1,1); hold on;
plot(pf,10*log10(abs(pLP))-20*log10(10),',b');
plot(pf,10*log10(abs(pHP))-20*log10(10),',k');
xlabel('frequency Hz'); ylabel('dB'); title('frequency domain');
legend('lowpass interp','highpass interp'); axis tight; grid on;

subplot(2,1,2); hold on; L=fix(Fs/f)*4; L=min(L,NFFT);
plot([0:L]*2/(2*Fs),abs([pLP,pf]),',b');
plot([0:L]*2/(2*Fs),abs([pHP,pf]),',k');
xlabel('sample'); ylabel('NADC'); title('time domain');
legend('input','lowpass decim'); axis tight; grid on;
```

The following MATLAB® script uses a sinusoid to verify the decimation by a factor of two. Decimation is usually done by filtering and then dropping every other sample. In the frequency domain, both the low and high frequency folded decimated signals are plotted. In the time domain, the low frequency decimated signal is plotted over the original signal to verify that they match.

```matlab
%print('LADF test for 2:1 decimation (must be designed for this\n');
NFFT=2^13; % FFT resolution
iflag=1; % decimator
if ((iFlag==0) \n    A=input('test signal max amplitude? '); \n) \nelse \n    A=input('test signal bits (max: half of integer bits'); \nend
Fs=input('sampling frequency Hz? ');
f1=input('test frequency (0 to %.0f Hz, will be folded)? ',Fs/2);
f2=input('test frequency (%.0f to %.0f Hz, will not be folded)? ',Fs/2/2);

%-- test
in=A*sin(2*pi*t*[0:NFFT-1]/Fs)+sin(2*pi*f2*[0:NFFT-1]/Fs))/2;
[outLP,outHP]=WDecest(in,iFlag,eFlag);
[pLP,pf]=pwelch(outLP,win,NOVL,NFFT,Fs/2);
[pHP,pf]=pwelch(outHP,win,NOVL,NFFT,Fs/2);

%-- plot
figure; subplot(2,1,1); hold on;
plot([0:L/2]*2/outLP(1:L)/2,'b.');</t0 ; x0 ; x1 ; T0 ; T1 ; T2 ; T3 ; T4 ; T5 ; T6;
// filter input: i1=i2=sample(n)

//**** Upper arm ****
t0 = i2 - T0 ; T0 = 0.487102*t0 ; x0 = x0 - T0 ; // adaptor 0: g=0.512898
t0 = T4 - T3 ; x0 = 0.332436*t0 ; T4 = x0 - T0 ; // adaptor 4: g=0.334236
t0 = w0 - e1 ; T3 = 0.331276*t0 ; x0 = *u2*T3 - t0 ; // adaptor 3: g=0.668724

//**** Lower arm ****
t0 = T1 + T2 ; T2 = 0.392923*t0 ; T3 = 0.392923*t0 ; x0 = x0 - T0 ; // adaptor 2: g=0.607707
t0 = w0 - e1 ; x1 = 0.404406*t0 ; T1 = x0 - T1 ; // adaptor 1: g=0.404406
t0 = T6 - T5 ; x1 = 0.206694*t0 ; T6 = x0 - T6 ; // adaptor 6: g=0.206694
t0 = w0 - e1 ; T5 = 0.103866*t0 ; x0 = T5 - x0 ; // adaptor 5: g=0.896134

// filter output: sample(n)=[n+1]/2 for lowpass/highpass
```
The output of the tests are in Figures 1-3: the recovery after a hard discontinuity is verified, over/undershoots after each step discontinuity are clearly visible, the output goes to zero after the input goes to zero.

**Figure 1. Frequency response**

**Figure 2. Group delay**
Figure 3. Stability check for the Elliptic/Cauer filter, order 7.

This is the floating-point implementation for the bi-reciprocal Elliptic/Cauer filter, order 19, designed for interpolation/decimation:

```c
// Elliptic/Cauer order 19, bireciprocal, for interpolation/decimation
// stopband min attenuation as>77 dB at fp=0.51% (100%=F/2)
// passband attenuation spread apr=0.00 dB at fp=0.49% (100%=F/2)
void filter(float i1, float i2, float *o1, float *o2) {
    float t0, x0;
    static float T1, T3, T5, T7, T9, T11, T13, T15, T17;
    // interpolator input: i1=i2=sample(n)
    // decimator input: i1=sample(n), i2=sample(n+1)
    // Upper arm
    t0 = i1 - T3; x0 = 0.226119*t0 - T3; T3 = x0 - t0;
    // adaptor 3: g=-0.226119
    t0 = i1 - T7; x0 = 0.397578*t0 - T7; T7 = x0 - t0;
    // adaptor 7: g=-0.602422
    t0 = T11 - x0; T11 = 0.160677*t0 - T11; x0 = T11 - t0;
    // adaptor 11: g=-0.839323
    t0 = T15 - x0; T15 = 0.049153*t0 - T15; *o2 = T15 - t0;
    // adaptor 15: g=-0.950847
    // Lower arm
    t0 = i2 - T1; x0 = 0.063978*t0 - T1; T1 = x0 - t0;
    // adaptor 1: g=0.063978
    t0 = T5 - x0; T5 = 0.423068*t0 - T5; x0 = T5 - t0;
    // adaptor 5: g=-0.423068
    t0 = T9 - x0; T9 = 0.258673*t0 - T9; x0 = T9 - t0;
    // adaptor 9: g=-0.741327
    t0 = T13 - x0; T13 = 0.094433*t0 - T13; x0 = T13 - t0;
    // adaptor 13: g=-0.905567
    t0 = T17 - x0; T17 = 0.017579*t0 - T17; *o1 = T17 - t0;
    // adaptor 17: g=-0.984721
    // interpolator output: sample(n)=(o1+/-o2)/2 for lowpass/highpass
    // decimator output: sample(n)=o1, sample(n+1)=o2 for lowpass/highpass
}
```

The output of the tests are in Figures 4-5.

The interpolator has been tested with a 1 kHz sinusoid, the input sampling frequency is $F_i=64$ kHz, the output is $F_o=128$ kHz: the low-pass interpolated signal is at 1 kHz, the high-pass interpolated signal is at 63 kHz. In general, an input frequency $0<f<Fi/2$ will be seen at $f$ on the low-pass output and $Fi-f$ on the high-pass output.

The decimator has been tested with a 1 kHz and an 18 kHz sinus signal, the input sampling frequency is $F_i=64$kHz, the output is $F_o=32$ kHz: the low-pass decimated signal is at 1 kHz, the high-pass folded decimated signal is at 14 kHz. In general, an input frequency $0<f<Fi/4$ will be kept by the low-pass and seen at $f$ in the output, an input frequency $Fi/4<f<Fi/2$ will be kept by the high-pass and folded at $Fi/2-f$. 
Figure 4. Interpolator test for the bi-reciprocal Elliptic/Cauer filter, order 19, designed for interpolation/decimation

Figure 5. Decimator test for the bi-reciprocal Elliptic/Cauer filter, order 19, designed for interpolation/decimation
Support material

<table>
<thead>
<tr>
<th>Related design support material</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wearable sensor unit reference design, STEVAL-WESU1</td>
</tr>
<tr>
<td>SensorTile development kit, STEVAL-STLKT01V1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design tip, DT0091, Lattice wave digital filter design and automatic C code generation</td>
</tr>
</tbody>
</table>

Revision history

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-Nov-2017</td>
<td>1</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>
IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers' products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2017 STMicroelectronics – All rights reserved