FIR and IIR Filtering using Streaming SIMD Extensions

Version 1.1

01/99

Order Number: 243547-002

Table of Contents

Revision History

Revision	Revision History	Date
1.1	FCS revision.	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1 Introduction

Streaming SIMD Extensions for the Intel® Architecture (IA) instruction set provide floating point single-instruction, multiple-data (SIMD) instructions. This application note discusses the use of Streaming SIMD Extensions in the implementation of FIR (Finite Impulse Response) and IIR (Infinite Impulse Response) filters. The Streaming SIMD Extensions provide performance improvement for these filter operations through SIMD instructions that operate on packed single-precision floating point data.

2 FIR and IIR Filtering

FIR and IIR filtering are two of the fundamental operations in digital signal processing. In FIR/IIR filters (as is discussed below, these can both be thought of as a single algorithm, since an FIR filter is a subset of an IIR filter), a sequence of input digital samples is convolved with either one or two kernels of *filter coefficients* to produce a filtered output sequence.

A general linear digital signal processing system, which can perform any IIR or FIR filter, is shown schematically in Figure 1. In this system, a sequence of input samples, x_k, is filtered by multiplying terms of the sequence by the filter coefficients a_i and b_i and adding up the resulting products to produce the output sequence y_k. In this diagram, z^{-1} represents a single-sample delay.

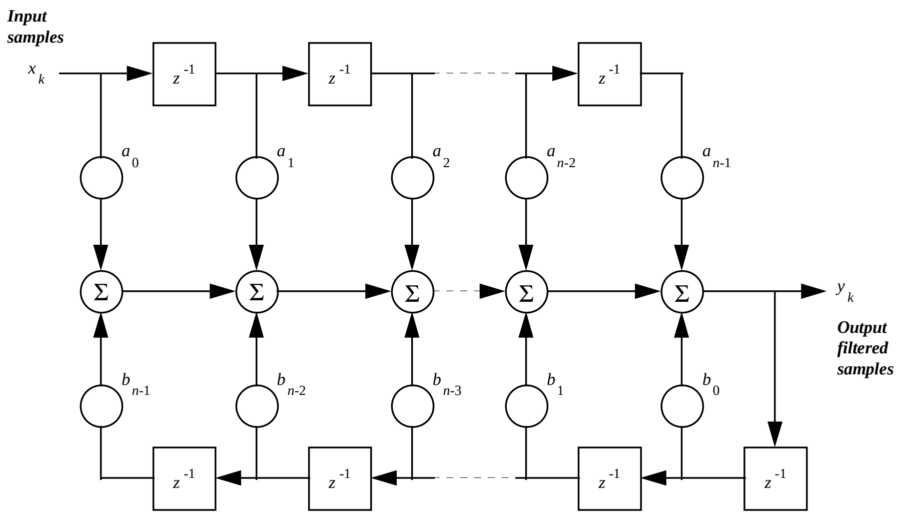

Figure 1: Flow graph for general IIR filter

As can be seen from the diagram, an equation describing the outputs of this filter is:

$$y_k = \sum_{i=0}^{n-1} a_i x_{k-i} + \sum_{i=0}^{n-1} b_i y_{k-1-i}$$

If there is no "feedback" or recursive component to the filter (that is, if all the $b_i = 0$) then this is called a *finite impulse response* (FIR) filter since the response of such a filter to an impulse input (that is, the sequence [. . . 0 0 1 0 0 . . .]) is finite in length. On the other hand, recursive filters (with non-zero *b*'s) have an *infinite impulse response*, and hence are called IIR filters. Both FIR and IIR filters are used extensively in many kinds of signal-processing applications, as described in Section 2.1 below.

In Section 2.2, the FIR/IIR filter algorithm is discussed specifically in the context of a speech compression algorithm using 10th-order LPC analysis (a popular choice), since the use of *n*=10 leads to some special considerations when coding the filter with Streaming SIMD Extensions.

For more details on the theory and practice of FIR and IIR filters, see any introductory text on digital signal processing, such as Oppenheim and Shafer's *Discrete-Time Signal Processing.*

2.1 Applications for FIR/IIR Filters

The applications of FIR and IIR filters in signal processing are numerous. They can be used for many kinds of frequency-domain alterations of a signal: low-pass filtering (in which high frequencies are removed), band-pass filtering (in which just a range of frequencies are retained), high-frequency emphasis (e.g., treble boost), and so on. For example, an audio equalizer is just a bank of filters that boost or attenuate different frequency ranges in the input signal.

FIR/IIR filters are also used to produce 3D (positional) audio effects, using the *head-related transfer function* (HRTF). To make a sound appear to be positioned at a certain location in space, a separate filter is applied to the Left and Right stereo signals before presenting to the left and right ears.

Adaptive FIR/IIR filters are also used extensively in speech compression. In Linear Predictive Coding (LPC) – a popular component of speech compression algorithms – each sample in a signal is predicted as a linear combination of the *n* previous samples in time, which is equivalent to calculating each sample using an FIR filter. Both FIR and IIR filters (of order *n*, the order of LPC analysis) are used extensively inside a speech compressor for this purpose as well as others (such as filtering the signal to emphasize certain peaks in the frequency response, thus improving the perceptual quality).

2.2 Implementing an FIR/IIR Filter

Consider now the implementation of an FIR/IIR filter routine in C code. Suppose, as is often the case, that the order of the filter (*n*) is a pre-known constant for the given application. In other words, the task is to write an FIR/IIR filter for a specific value of *n*.

For illustrative purposes, and also because this is a realistic value, we choose to use *n*=10. This is the value of *n* most commonly used in an LPC speech compression algorithm such as the ITU standards G.723.1 and G.729. The value *n*=10 is also a useful one because, not being divisible by 4, it requires some special care when implementing the filter in Streaming SIMD Extensions. Thus it is a good value to use for the purpose of illustrating some Streaming SIMD Extensions coding techniques.

Here is an implementation of an order-10 FIR/IIR filter in straightforward C code:

```
#define BLOCKLEN 240    // Number of points to be filtered
#define ORDER 10        // Order of the filter

void Filter(
float *x,               // Input (and output) array
float *taps,            // Array of filter coefficients (ORDER a's followed
by ORDER b's)
float *Fir, float *Iir  // Delay lines - hold last n samples of input and
output
)
{
  int  i,k;
  float acc;
```

```
   for (i=0; i <BLOCKLEN; i++)
   {
     acc = 0;
     for (k=0; k<ORDER; k++)
       acc += Fir[k] * taps[k];

     for (k=0; k<ORDER; k++)
       acc += Iir[k] * taps[ORDER+k];

     for (k=ORDER-1; k>0; k--)
     {
       Fir[k] = Fir[k-1];
       Iir[k] = Iir[k-1];
     }

     Fir[0] = *x;
     *x++ = Iir[0] = acc;
   }
 }
```

Example 1: C code for an order 10 FIR/IIR filter

In this routine, *x* is the input array as well as the output array (that is, the filtering is done in-place). The input variable *taps* points to the array of filter coefficients (or *filter taps*), which have been combined into a single array containing first all the *a*'s and then all the *b*'s. The reason for this combining is to reduce by one the number of registers needed inside the routine to point to different areas of memory. This helps when we get to the implemetation of the filter in Streaming SIMD Extensions assembly language.

Note that this routine requires and uses two *n*-element *delay lines* – the *Fir* and *Iir* arrays. These arrays hold the last *n* samples of the input and output, respectively, and thus emulate the *n* one-sample delays shown in Example 1. Note that these arrays must be *static* since they must be preserved across calls to the *Filter* routine.

Before implementing this filter in Streaming SIMD Extensions assembly code, we first perform two optimizations on this C code: (1) unrolling the inner loops, and (2) making the circular buffers more efficient.

Loop unrolling is a common programming technique for reducing the overhead associated with a *for* loop. It is particularly important in cases, like this one, in which the amount of work being done in each iteration is small. In this code, each iteration does a single multiply-accumulate. Since each of the two multiply/accumulate loops is only 10 iterations long, we can unroll both of them completely, as follows:

```
 void Filter(float *x, float *taps,
 float *Fir, float *Iir)
 {
   int  i,k;
   float acc;
```

```
  for (i=0; i <BLOCKLEN; i++)
  {
    acc =   Fir[0] * taps[0] +
         Fir[1] * taps[1] +
         Fir[2] * taps[2] +
         Fir[3] * taps[3] +
         Fir[4] * taps[4] +
         Fir[5] * taps[5] +
         Fir[6] * taps[6] +
         Fir[7] * taps[7] +
         Fir[8] * taps[8] +
         Fir[9] * taps[9] +

         Iir[0] * taps[10] +
         Iir[1] * taps[11] +
         Iir[2] * taps[12] +
         Iir[3] * taps[13] +
         Iir[4] * taps[14] +
         Iir[5] * taps[15] +
         Iir[6] * taps[16] +
         Iir[7] * taps[17] +
         Iir[8] * taps[18] +
         Iir[9] * taps[19];

    for (k=ORDER-1; k>0; k--)
    {
      Fir[k] = Fir[k-1];
      Iir[k] = Iir[k-1];
    }

    Fir[0] = *x;
    *x++ = Iir[0] = acc;
  }
}
```

Example 2: Technique of unrolling a for loop

Although this technique makes the code larger and makes it take longer to load into the instruction cache the first time through the outer loop, this is more than made up for by the increased efficiency of the remaining BLOCKLEN-1 iterations. In this example, taken from the G.723 speech compressor, BLOCKLEN has the large value of 240. So one loop gets slower while the other 239 get nearly twice as fast – a clear improvement.

This version of the loop has 20 multiply/accumulate operations and 18 data shuffling operations (to shift the delay lines over by one position) per outer loop. The next step is to get rid of most of the work relating to the delay lines, which are in reality being used as circular buffers.

One technique for efficiently implementing a circular buffer on the Intel Architecture is to use a double-size linear buffer whose upper half is identical to the lower half. If we have

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
```

in an array, then any circular subset of the array (for instance, 4567890123) can be obtained by simply accessing a linear subset of this array. Applying this idea gives the code shown in Example 2.

Note the following two important features of the code in Example 2:

- the *Fir* and *Iir* arrays are now double the size they were before, and
- the value of *p* must be static (just like the contents of the *Fir* and *Iir* arrays) since it must be preserved across calls to the *Filter* routine.

The technique in Example 2 requires the computation of

```
    p = (p-1) % 10;
```

each time through the loop. This is done most efficiently with a lookup table, as shown in the above code. This optimization reduces the work required to do a FIR/IIR filter to the absolute minimum, of 20 multiply/accumulates plus some assignments and outer loop overhead.

3 Conclusion

This application note describes the implementation of a 10th-order FIR/IIR filter in assembly code using Streaming SIMD Extensions. A significant performance increase is obtained by using Streaming SIMD Extensions.

In addition, two common Streaming SIMD Extensions programming idioms have been described:

- Duplicating data arrays three times in order to guarantee four-element memory alignment, and
- Padding operation lengths to the next multiple of four in order to make them fit into four-wide floating-point SIMD.

These techniques can be applied to many Streaming SIMD Extensions programming tasks.

4 Optimzed C-code Implementation

The C code given in Example 2, when compiled with a modern C compiler for the Intel Architecture, yields very good (essentially optimal) performance, thus obviating the need for a pure assembly-language implementation. The reason for this is that a good C compiler automatically “schedules” the floating-point instructions needed to do the 20 multiply-accumulates (using the FXCH instruction to manage the values on the floating-point stack) so that internal processor stalls are minimized.

The optimized C version of the FIR/IIR filter can also be found in Listing 1 at the end of this application note.

5 Streaming SIMD Extensions Assembly Code Example

There are two issues to be resolved in order to turn the C implementation into an efficient Streaming SIMD Extensions FIR/IIR filter:

1) The order of the filter, 10, is not a multiple of 4. To use the SIMD floating-point instructions effectively, each multiply and add operation that's done must operate on four floating-point numbers that are contiguous in memory.
2) For efficient operation, every SIMD memory read and write (in which four contiguous floating-point numbers are read or written) needs to be 16-byte (4-number) aligned. Examining the efficient C code presented above, we can see that all accesses to the `taps` array are naturally aligned (since we always start with `taps[0]`), however this is *not* the case with reads from the Fir[ ] and Iir[ ] arrays. Indeed, accesses to those arrays are 4-element aligned only once every four times through the outer loop – when *p* is a multiple of 4.

We now describe how to resolve each of these issues.

Ensuring a multiple of 4

This problem is solved in a simple way. Even though the filter is *really* a 10th-order filter, we just round the order up to the nearest multiple of four, and pretend it's a 12th-order filter. The delay lines become of length twelve, and the filter taps, instead of (0,1,2,3,4,5,6,7,8,9), become (0,1,2,3,4,5,6,7,8,9,x,x), where *x* represents the value zero. A tap value of zero results in zero when multiplied by its corresponding delay line element; therefore, it contributes nothing to the accumulated sum, and the calculated result remains correct.

Of course, a 12th-order filter requires more computation than a 10th-order filter (all things being equal), but given the 4-wide SIMD nature of the Streaming SIMD Extensions SIMD-floating-point instructions, this is the most efficient way to deal with 10-element arrays using Streaming SIMD Extensions.

Memory alignment

The memory alignment problem can be solved using a technique that's fairly common in Streaming SIMD Extensions programming: *data duplication*.

Recall that we have a taps array of the form

```
0123456789..0123456789..
```

that we want to access, in 4-element chunks, starting at any position *p*. One fourth of these positions (ones where *p* is a multiple of four) are already aligned, so we can use this array directly when we want one of those subsets.

In order to handle the other cases, we duplicate the taps array three times: one in which element 1 is aligned in memory, one in which element 2 is aligned in memory, and one in which element 3 is aligned in memory. These arrays look like:

Filter 1:

```
.012 3456 789.      (repeat)
```

Filter 2:

```
..01 2345 6789      (repeat)
```

Filter 3:

```
...0 1234 5678 9... (repeat)
```

Note that the first two of these fit into 12 elements (actually, 24 since the array is repeated in order to emulate the circular buffer, as explained earlier), whereas the fourth one requires 16, because of the way the 10 elements straddle the "quads" (groups of four numbers). Thus, on average, the 10 filters taps have turned into (12+12+12+16)/4 = 13 taps.

Now, recalling the C code of Example 2, observe how any convolution of the taps array by a delay line can be accomplished using all 4-element-aligned accesses.

The first time through the outer loop, we need to convolve Fir, starting at position 0, with the filter taps. This is accomplished using Fir starting at position 0 and the unmodified taps array. The second time through the outer loop, we need to convolve Fir, starting at position 11, with the filter taps. This is accomplished by using Fir starting at position 8 and Filter 3 shown above. This yields:

```
Fir:        89ab0123456789ab
Filter 3:   ...0123456789...
```

which, since the dots in the representation of Filter 3 represent the value zero, is precisely the convolution of the 10 real filter taps with the Fir delay line starting at position 11 (hexadecimal *b* above). Note that this is the one that requires 16 multiply-adds, rather than just 12. Since Filter 3 is aligned in memory (as well as the Fir array), then all the memory accesses in this computation are aligned.

The third time through the outer loop, we need to convolve Fir, starting at position 10, with the filter taps. This is accomplished by using Fir starting at position 8 and Filter 2 shown above. This yields:

```
Fir:        89ab01234567
Filter 2:   ..0123456789
```

Finally, the fourth time, we need to convolve Fir, starting at position 9, with the filter taps. This is accomplished by using Fir starting at position 8 and Filter 1 shown above. This yields:

```
Fir:        89ab01234567
Filter 1:   .0123456789.
```

After this sequence of four convolutions we switch back to using the unmodified Filter 0 again. Thus, the rest of the filter outputs can be computed by wrapping a loop around these first four. Note that *p*, the pointer into the delay lines, gets decremented by 4 (mod 12) after the first of the four convolutions (as in the example above, where the first one was done using *p*=0, and the next three with *p*=8). This decrementing mod 12 is done using a lookup table, just like we did in the C code.

Listing 1

10th-order FIR/IIR filter in optimized C code

```
int minus1mod10[]= {9,0,1,2,3,4,5,6,7,8};

void Filter(float *x, float *taps,
float *Fir, float *Iir, int p)
{
  int  i,k;
  float acc;

  for (i=0; i <BLOCKLEN; i++)
  {
    acc = Fir[p+0] * taps[0] +
          Fir[p+1] * taps[1] +
          Fir[p+2] * taps[2] +
          Fir[p+3] * taps[3] +
          Fir[p+4] * taps[4] +
          Fir[p+5] * taps[5] +
          Fir[p+6] * taps[6] +
          Fir[p+7] * taps[7] +
          Fir[p+8] * taps[8] +
          Fir[p+9] * taps[9] +

          Iir[p+0] * taps[10] +
          Iir[p+1] * taps[11] +
          Iir[p+2] * taps[12] +
          Iir[p+3] * taps[13] +
          Iir[p+4] * taps[14] +
          Iir[p+5] * taps[15] +
          Iir[p+6] * taps[16] +
          Iir[p+7] * taps[17] +
          Iir[p+8] * taps[18] +
          Iir[p+9] * taps[19];

    p = minus1mod10[p];
    Fir[p] = Fir[p+ORDER] = *x;
    *x++ = Iir[p] = Iir[p+ORDER] = acc;
  }
}
```

Listing 2

Optimized 10th-order FIR/IIR filter using Streaming SIMD Extensions

```
/*=========================================================================
This routine performs a 10th-order IIR filter on an input
array (data) and puts the results back in 'data'.  A filter like this
is needed in the G.723 and G.729 speech compression algorithms.

It uses a pair of arrays for the two delay lines, using the common
trick of making them double size (0 1 2 ... N-1 0 1 2 ... N-1 instead
of just 0 1 2 ... N-1) so that we can emulate a circular buffer.

In addition, we want all accesses to the delay lines to be 8-byte
aligned.  To do this, we make three copies of the filter taps.
If the original is 0123456789, the copies are:
  .012 3456 789.
  ..01 2345 6789
  ...0 1234 5678 9...
We do four consecutive outputs in the inner loop, one using each
filter.  (Note that 'p', which points to the delay lines, gets
decremented after the first filter.)

The taps are FIR taps (a's) first then IIR (b's).  After duplication,
the taps array contains

a0 b0  a1 b1  a2 b2  a3 b3

where each of these is a set of 12 taps, except for the last
one which is 16, because the 0123456789 overlaps 4 quads not just 3.
=======================================================================*/


#define Order 12   /* pretend it's 12 even though really 10 */
#define FrameLen 240

int zero[4] = {0,0,0,0};
int minus4mod[] = {8,0,0,0,0,0,0,0,4};



/*-----------------------------------------------------------*/
```

```
void Filter(float *x, float *taps, float *Fir, float *Iir)
{
#define fir eax
#define iir ecx
#define tap edx
#define p   ebx
#define cnt esi
#define dat edi

#define TAP0 tap
#define TAP1 tap+4*24
#define TAP2 tap+4*24*2
#define TAP3 tap+4*24*3
#define SZ (Order*2)

  int i,j;

/*---------------------------------
   Create the other 3 sets of taps.  The original taps[] array is
   20 elements long (10 for the a’s, 10 for the b’s) but the final
   one has the structure

         a’s          b’s             a’s            b’s
     0123456789.. 0123456789..  .0123456789.  .0123456789.
         a’s          b’s               a’s                b’s
     ..0123456789 ..0123456789  ...0123456789...  ...0123456789...

   which is groups of 12,12,12,12,12,12,16,16

   Note that this whole duplication operation should probably be
   done outside this routine, under the assumption that the taps
   are either constants or change infrequently.  It is done here
   solely for simplicity’s sake.
---------------------------------*/

  for (j=0; j<=12; j+=12)  // once for the a’s, once for the b’s
  {
    taps[j+SZ] = taps[j+2*SZ-1] = 0.0f;
    taps[j+2*SZ] = taps[j+2*SZ+1] = 0.0f;
    taps[j+3*SZ] = taps[j+3*SZ+1] = taps[j+3*SZ+2] =
      taps[j+4*SZ+1] = taps[j+4*SZ+2] = taps[j+4*SZ+3] = 0.0f;
    for (i=0; i<12; i++)
      taps[j+SZ+1+i] = taps[j+2*SZ+2+i] = taps[j+3*SZ+3+i] = taps[j+i];
  }

// Now start the actual filtering

  __asm
  {
    xor p,p;   // p=0
    mov cnt,FrameLen;
    mov fir,Fir;
    mov iir,Iir;
    mov tap,taps;
    mov dat,x;
    push ebp;  // free up ebp so we can use it

loop1:
//--------------------- filter 0 -----------------------

    movaps xmm7,zero;

    movaps xmm0,[fir+4*p];
    mulps  xmm0,[TAP0];
    addps  xmm7,xmm0;
    movaps xmm1,[fir+4*p+16];
    mulps  xmm1,[TAP0+16];
    addps  xmm7,xmm1;
    movaps xmm2,[fir+4*p+32];
    mulps  xmm2,[TAP0+32];
    addps  xmm7,xmm2;

    movaps xmm3,[iir+4*p];
    mulps  xmm3,[TAP0+48+0];
    addps  xmm7,xmm3;
    movaps xmm4,[iir+4*p+16];
    mulps  xmm4,[TAP0+48+16];
    addps  xmm7,xmm4;
    movaps xmm5,[iir+4*p+32];
    mulps  xmm5,[TAP0+48+32];
    addps  xmm7,xmm5;

    mov p,minus4mod[4*p];

    movaps xmm6,xmm7;
    shufps xmm6,xmm6,14;
    addps  xmm7,xmm6;
    movaps xmm6,xmm7;
    shufps xmm6,xmm6,1;
    addps  xmm7,xmm6;

; Store *dat in fir[p+3], fir[p+3+N].
; Store filter result in same places in iir as well as *dat

    mov  ebp,[dat];
    mov  [fir+4*p+4*3],ebp;
    mov  [fir+4*p+4*3+4*Order],ebp;
    movss DWORD PTR[iir+4*p+4*3],xmm7;
    movss DWORD PTR[iir+4*p+4*3+4*Order],xmm7;
    movss DWORD PTR[dat],xmm7;

//--------------------- filter 3 ----------------------
// Note that this one must be done with length 16, since the
// way the 10 taps fall are
//    ...0 1234 5678 9...

    movaps xmm6,zero;

    movaps xmm0,[fir+4*p];
    mulps  xmm0,[TAP3];
    addps  xmm6,xmm0;
    movaps xmm1,[fir+4*p+16];
    mulps  xmm1,[TAP3+16];
    addps  xmm6,xmm1;
    movaps xmm2,[fir+4*p+32];
    mulps  xmm2,[TAP3+32];
    addps  xmm6,xmm2;
    movaps xmm0,[fir+4*p+48];
    mulps  xmm0,[TAP3+48];
    addps  xmm6,xmm0;

    movaps xmm3,[iir+4*p];
    mulps  xmm3,[TAP3+48+0];
    addps  xmm6,xmm3;
    movaps xmm4,[iir+4*p+16];
    mulps  xmm4,[TAP3+48+16];
    addps  xmm6,xmm4;
    movaps xmm5,[iir+4*p+32];
    mulps  xmm5,[TAP3+48+32];
    addps  xmm6,xmm5;
    movaps xmm3,[iir+4*p+48];
    mulps  xmm3,[TAP3+48+48];
    addps  xmm6,xmm3;

    movaps xmm7,xmm6;
    shufps xmm7,xmm7,14;
    addps  xmm6,xmm7;
    movaps xmm7,xmm6;
    shufps xmm7,xmm7,1;
    addps  xmm6,xmm7;

; Store *dat in fir[p+2], fir[p+2+N].
; Store filter result in same places in iir as well as *dat

    mov  ebp,[dat+4];
    mov  [fir+4*p+4*2],ebp;
    mov  [fir+4*p+4*2+4*Order],ebp;
    movss DWORD PTR[iir+4*p+4*2],xmm6;
    movss DWORD PTR[iir+4*p+4*2+4*Order],xmm6;
    movss DWORD PTR[dat+4],xmm6;

//-------------------- filter 2 ----------------------

    movaps xmm7,zero;

    movaps xmm0,[fir+4*p];
    mulps  xmm0,[TAP2];
    addps  xmm7,xmm0;
    movaps xmm1,[fir+4*p+16];
    mulps  xmm1,[TAP2+16];
    addps  xmm7,xmm1;
    movaps xmm2,[fir+4*p+32];
    mulps  xmm2,[TAP2+32];
    addps  xmm7,xmm2;

    movaps xmm3,[iir+4*p];
    mulps  xmm3,[TAP2+48+0];
    addps  xmm7,xmm3;
    movaps xmm4,[iir+4*p+16];
    mulps  xmm4,[TAP2+48+16];
    addps  xmm7,xmm4;
    movaps xmm5,[iir+4*p+32];
    mulps  xmm5,[TAP2+48+32];
    addps  xmm7,xmm5;

    movaps xmm6,xmm7;
    shufps xmm6,xmm6,14;
    addps  xmm7,xmm6;
    movaps xmm6,xmm7;
    shufps xmm6,xmm6,1;
    addps  xmm7,xmm6;

; Store *dat in fir[p+1], fir[p+1+N].
; Store filter result in same places in iir as well as *dat

    mov  ebp,[dat+8];
    mov  [fir+4*p+4],ebp;
    mov  [fir+4*p+4+4*Order],ebp;
    movss DWORD PTR[iir+4*p+4],xmm7;
    movss DWORD PTR[iir+4*p+4+4*Order],xmm7;
    movss DWORD PTR[dat+8],xmm7;

//-------------------- filter 1 ----------------------

    movaps xmm6,zero;

    movaps xmm0,[fir+4*p];
    mulps  xmm0,[TAP1];
    addps  xmm6,xmm0;
    movaps xmm1,[fir+4*p+16];
    mulps  xmm1,[TAP1+16];
    addps  xmm6,xmm1;
    movaps xmm2,[fir+4*p+32];
    mulps  xmm2,[TAP1+32];
    addps  xmm6,xmm2;

    movaps xmm3,[iir+4*p];
    mulps  xmm3,[TAP1+48+0];
    addps  xmm6,xmm3;
    movaps xmm4,[iir+4*p+16];
    mulps  xmm4,[TAP1+48+16];
    addps  xmm6,xmm4;
    movaps xmm5,[iir+4*p+32];
    mulps  xmm5,[TAP1+48+32];
    addps  xmm6,xmm5;
    movaps xmm7,xmm6;
    shufps xmm7,xmm7,14;
    addps  xmm6,xmm7;
    movaps xmm7,xmm6;
    shufps xmm7,xmm7,1;
    addps  xmm6,xmm7;

; Store *dat in fir[p], fir[p+N].
; Store filter result in same places in iir as well as *dat

    mov  ebp,[dat+12];
    mov  [fir+4*p],ebp;
    mov  [fir+4*p+4*Order],ebp;
    movss DWORD PTR[iir+4*p],xmm6;
    movss DWORD PTR[iir+4*p+4*Order],xmm6;
    movss DWORD PTR[dat+12],xmm6;


    add dat,16;
    sub cnt,4;
    jne loop1;

    pop ebp;
  }
}
```