Analog Devices公司DSP、处理器和开发工具的使用技术指南
联系我们的技术支持中心：dsp.support@analog.com 和 dsptools.support@analog.com
查看我们的在线资源：http://www.analog.com/ee-notes 和 http://www.analog.com/processors

ADSP-TS20x TigerSHARC(R)处理器上的16-bit FIR滤波器

Klas Brink和 Rickard Fahlquist提供

第一版—2004年1月13日

引言

这篇文章介绍了两套直接型FIR的汇编代码实现方案的存储器需求，及以16-bit整数精度处理复数输入和输出，复数或实数系数的能力。这些方案提供的方法在节省存储器的同时还可以获得高性能。

总述

下面给出了直接型FIR滤波器的数学表达式。

$$y[i] = \sum_{k=0}^{M-1} x[i-k] \cdot h[k]$$

I 0，1，…，*N*−1
N 样本的数量
M 滤波器系数数量
y[*i*] 输出样本数量 *i*
x[*i*−*k*] 输入样本数量 *i*−*k*
h[*k*] 滤波器系数数量*k*

等式1. 直接型FIR

该等式是滤波器系数向量h和（时间）反顺序输入数据向量x的内积。这也被称为h和x之间的卷积。

图1以图解方式表示了相同的等式。

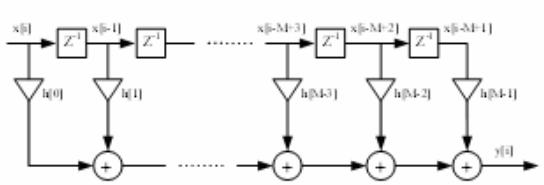

图1 直接型FIR

下面给出了同一FIR的C伪代码描述，其中**表示复数乘法。

```
for(i=0; i< N; i++){
  y[i] = 0;
  Ncoeffs = i < (M-1) ? i : (M-1);
  for(k=0; k<=Ncoeffs; k++){
    y[i] = y[i] + x[i-k] ** h[k];
  }
}
```

列表1 直接型FIR的C伪代码运算法则

TigerSHARC 处理器的并行处理

ADSP-TS20x TigerSHARC(R)处理器是高度并行的计算器件，有3种独特的并行处理类型：

* 延迟—2执行流水线
* 多个计算单元
* 宽存储器结构

这三种并行方式相互补充，为获得高效率，应尽量同时考虑这三种方式。本器件中计算速率和存储器带宽以这样的方式平衡：就是如果忽略三种平衡方式中的一种，将导致次佳性能。

复数系数、输入输出数据的16–bit整数FIR

介绍

每个内核时钟周期，ADSP–TS20x TigerSHARC处理器都支持两种复数乘法（每个计算块各一），同时伴随着同步数据传输。当然，类似等式1的滤波器计算可通过直接连序的方式实现，就是通过使用一个（内部）循环求和，每个迭代在旧的输入样本和系数之间执行乘法，并将其加入到最后迭代的输出中，另一个（外部）循环通过相同的程序再次生成输出样本。然而，使用ADSP–TS20x TigerSHARC处理器的并行功能和内部高带宽，可以获得更高的性能。

流水线操作和并行资源利用

FIR表达式表明大多数用于计算输出y[i]的数据和用于计算y[i+1]的数据是一样的。该情况同样适用于y[i+1]和y[i+2]，以次类推。C伪代码的“外部”循环执行所有计算全部输出样本所需的步骤。展开该外部循环，可得到下面三种好处：

1. 不同输出样本的计算之间可以再次使用数据。
2. MAC 操作可并行实现。
3. 循环溢出的影响减少。

不要忽视减少循环开销，因为这可以减少循环中执行必要条件分支所需的时间，从而提高进行实际计算可用时间的百分比。

展开外部循环4次，可产生由下面的C伪代码描述的一个运算法则。

```
for(i=0; i< N; i+=4){
  y[i]   = 0;
  y[i+1] = 0;
  y[i+2] = 0;
  y[i+3] = 0;
  Ncoeffs = i < (M-1) ? i : (M-1);
  for(k=0; k<=Ncoeffs; k++){
    y[i]   = y[i]   + x[i-k]   **h[k];
    y[i+1] = y[i+1]+ x[i+1-k]**h[k];
    y[i+2] = y[i+2]+ x[i+2-k]**h[k];
    y[i+3] = y[i+3]+ x[i+3-k]**h[k];
  }
}
```

列表2　外部循环展开4次

到目前，我们所做的是重复利用各系数。通过展开“内部”循环，我们也实现了输入数据的重复利用。展开4次内部循环可给出表3中的C伪代码。

```
for(i=0; i< N; i+=4){
  y[i]   = 0;
  y[i+1] = 0;
  y[i+2] = 0;
  y[i+3] = 0;
  Ncoeffs = i < (M-1) ? i : (M-1);
  for(k=0; k<=Ncoeffs; k+=4){
    y[i]   = y[i]   + x[i-k]   **h[k];
    y[i]   = y[i]   + x[i-1-k]**h[k+1];
    y[i]   = y[i]   + x[i-2-k]**h[k+2];
    y[i]   = y[i]   + x[i-3-k]**h[k+3];
    y[i+1] = y[i+1]+ x[i+1-k]**h[k];
    y[i+1] = y[i+1]+ x[i-k]   **h[k+1];
    y[i+1] = y[i+1]+ x[i-1-k]**h[k+2];
    y[i+1] = y[i+1]+ x[i-2-k]**h[k+3];
    y[i+2] = y[i+2]+ x[i+2-k]**h[k];
    y[i+2] = y[i+2]+ x[i+1-k]**h[k+1];
    y[i+2] = y[i+2]+ x[i-k]   **h[k+2];
    y[i+2] = y[i+2]+ x[i-1-k]**h[k+3];
    y[i+3] = y[i+3]+ x[i+3-k]**h[k];
    y[i+3] = y[i+3]+ x[i+2-k]**h[k+1];
    y[i+3] = y[i+3]+ x[i+1-k]**h[k+2];
    y[i+3] = y[i+3]+ x[i-k]   **h[k+3];
  }
}
```

列表3　外部和内部循环展开4次

我们已达到了数据重复利用的高级水平，且能够进行并行计算。C伪代码中不太明朗的事情就是我们也可以进行流水线操作（例如在进行计算操作的同时存取数据）。

数据划分

如图2中所示，复数16-bit数据在ADSP-TS20x TigerSHARC处理器中是以32-bit 字的方式呈现的。

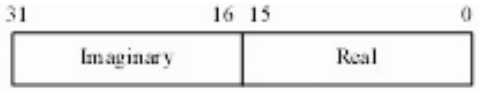

图2 复数16-bit数据表示法

输入和系数缓冲器结构

输入的样本和系数以图3所示方式存储于存储器中。

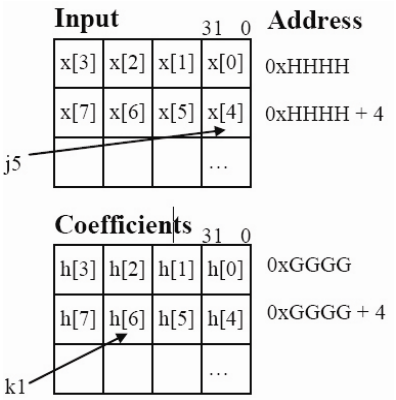

图3 输入和系数存储于存储器中

这些地址是以4字对齐的。这种数据存储方式使得用于输入样本的4字读取成为可能。4字读取也用于各系数。J5指示了当前正在加载输入样本的位置，k1指出了当前系数加载的位置。

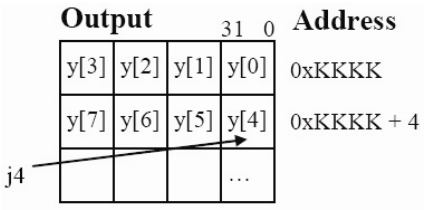

图4 存储器的输出存储

输出缓冲器结构

在每个外部循环迭代中，两个计算块（CBX 和CBY）每个计算两个输出样本。CBX生成样本y[i+3] 和 y[i+1]，CBY生成样本y[i+2] 和y[i]。将输出样本写入存储器时，使用4字排列存储，j4记下了写入的当前位置。

延迟线结构

滤波器配有存储器，用来保存滤波器所使用的最后M输入样本的历史记录，该历史记录被称为延迟线。延迟线中的样本是以4字方式读取的，图5显示了这样的存储方式。

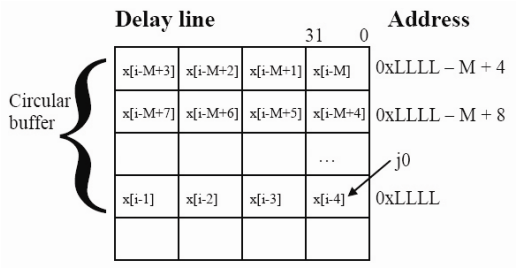

图5 存储器的延迟线

该延迟线是以循环缓存区实现的，j0为指针，指示了缓存区的当前位置/索引。

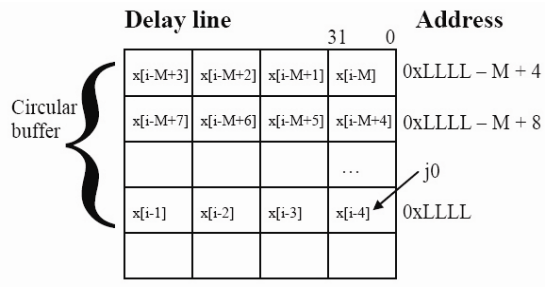

图6 更新后的延迟线

旧样本从延迟线读取，以j0所指示位置开始并向后寻址，然后在循环缓存区分界线处返回缓存区起始地址。假定图5显示了样本y[i]、y[i+1]、y[i+2]和y[i+3] 产生之前的延迟线，图6则显示了输出产生和延迟线更新之后延迟线的内容。

接口

连接滤波器的接口包括以下部分：

* 输出缓存区的指针

* 输入缓存区的指针

* 滤波器的输入样本数量

* 滤波器状态的指针（包括延迟线和系数缓冲器）

滤波器状态包括系数缓冲器的指针、系数数量、延迟线缓存区的指针和指示当前延迟线缓存区位置的索引/指针。

```
typedef struct
{
  int2x16 *h; // Filter coefficients
  int2x16 *d; // Delay line
  int2x16 *p; // Delay line Index
  int k;      // Number of coeff.
} fir_state_t;
```

列表4 滤波器状态构成

滤波器状态由列表4中的C代码定义。

该结构必须在滤波器首次使用之前被初始化（可查看附录中C代码初始化“功能”的例子）。

```
typedef struct
{
  int2x16 *h; // Filter coefficients
  int2x16 *d; // Delay line
  int2x16 *p; // Delay line Index
  int k;      // Number of coeff.
} fir_state_t;
```

列表5 滤波器功能原型

表5显示了接口的一个C代码原型。

实数系数和复数输入/输出数据的16-bit整数FIR

有时需要分别过滤复数样本中的实数和虚数部分。比如，天线数据的I部分和Q部分被当作分开的数据流，独立的受同一滤波器内核的影响。

格式

实数系数和输入数据的实数和虚数部分都是16-bit的。图7和图8显示了滤波器运算法则的输入是如何被格式化的。

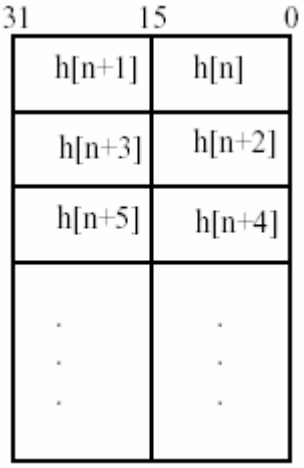

图7 滤波器系数格式

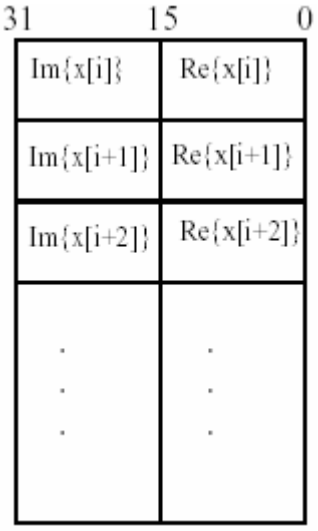

图8 输入样本格式

由于被滤波的数据是32-bit宽，每个16-bit短字被单独处理，一种可以利用ADSP-TS20x TigerSHARC处理器并行结构的方法是复制每个滤波器的系数，生成一对对的系数，每对系数为32-bit。

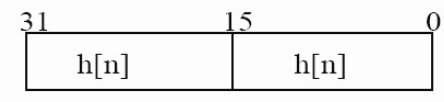

图9 复制的系数

在每个计算块中，4个16-bitMACs每个周期都执行，实部结果累积在两个MR寄存器中，虚部结果存储在另外两个寄存器中。

实现

经滤波的输出样本(y[i])在计算块X中进行计算，同时下一个(y[i+1])在计算块Y中进行计算。通过使用这种方法，特定数量的输入样本只需要一半数量的迭代。为实现此方法，从存储器加载时，系数一个计算块歪斜一个位置。在附录A (fir16_real.asm)列出的滤波器执行方式中，滤波器内核是很短的，足以完全存储于X和Y的寄存器文件中。

延迟线

延迟线与滤波器内核是一样长的。由于滤波器系数实16bit的而输入样本实32（16＋16）bit的，如果使用与复数滤波器相同的延迟线方式将出现问题。简单的解决办法是将最后处理的样本（延迟线）直接放置在下一个缓冲器之前，然后在存储器中进行滤波。这样做的缺点是在对滤波器的每次访问之间，延迟线需要被传输到下一个缓冲器的开端。然而，这对于中等长度的内核而言，开销并不很明显。

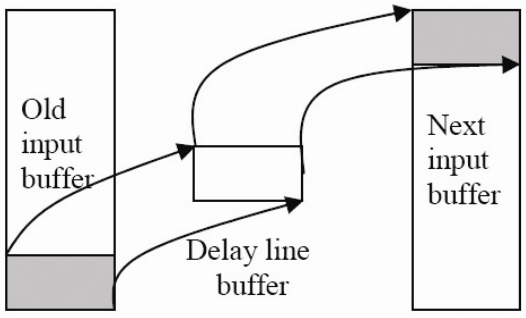

图10 延迟线操作处理

附录

下面给出了滤波器功能的汇编代码，以及滤波器状态结构的头文件，滤波器状态的初始化“函数”和滤波器功能原型，以便能在C代码中使用。

fir_16_comp.h

```
/* ************************************************************************************
 *
 * Copyright (c) 2003 Analog Devices Inc.  All rights reserved.
 *
 * ************************************************************************************/

#include <i16.h>

typedef struct
{
  int2x16 *h;// Filter coefficients
  int2x16 *d;// Start of (circular) delay line
  int2x16 *p;// Current index into delay line
  int k;     // Number of coefficients
} fir_state_t;


void fir_16_comp(int2x16 *outdata, int2x16 *indata, int N, fir_state_t *fir_state);

#define fir_init(state, coeffs, delay, ncoeffs)      \
                 (state).h = (int2x16 *) (coeffs);   \
                 (state).d = (int2x16 *) (delay);    \
                 (state).p = (int2x16 *) (delay);    \
                 (state).k = (int) (ncoeffs)
```

Listing 6. fir_16_comp.h

fir_16_comp.asm

```
/* ************************************************************************************
 *
 * Copyright (c) 2003 Analog Devices Inc.  All rights reserved.
 *
 * ************************************************************************************/

.global _fir_16_comp;
.section program;
.align_code 4;
_fir_16_comp:

   #define Yout        j4    // Pointer to output sample buffer
   #define Xin         j5    // Pointer to input sample buffer
   #define N           j6    // Number of samples to be filtered
   #define FState      j7    // Pointer to filter state structure
   #define h_offs      0     // Filter state structure offset to Coefficients
                             // buffer pointer
   #define d_offs      1     // Filter state structure offset to Delay line pointer
```

```
   #define p_offs      2     // Filter state structure offset to Delay line index
   #define k_offs      3     // Filter state structure offset to Number of
                             // coefficients


   // Save stack so we can use the internal registers
   // Stack PROLOGUE
   J26 = J27 - 64;           K26 = K27 - 64;;
   [J27 += -28] = CJMP;      K27 = K27 - 20;;
   Q[J27 + 24] = XR27:24;    Q[K27 + 16] = YR27:24;;
   Q[J27 + 20] = XR31:28;    Q[K27 + 12] = YR31:28;;
   Q[J27 + 16] = J19:16;     Q[K27 + 8 ] = K19:16;;
   Q[J27 + 12] = J23:20;     Q[K27 + 4 ] = K23:20;;
   // Stack PROLOGUE ENDS


   // Set number of samples to be generated/filtered
   // (outerloop, 4 samples each iteration).
   // Set number of times to go through filter kernel to use whole filter
   // (innerloop, 4 coeffs/taps each iteration).
   yR24 = N                  ;     // Number of samples
   xR24 = [FState + k_offs]  ;;    // Number of coeffs
   j0   = [FState + p_offs]  ;;    // Set j0 to point to latest sample in delay
                                   // line, i.e x[i+k-1]
   jl0  = xR24               ;     // Circular buffer length = Number of coeffs
   R24  = ASHIFT R24 BY -2   ;;    // Divide number of samples and coeffs by 4
   LC1  = yR24               ;     // Number of iterations for outerloop (LC1) =
                                   // Number of samples/4
   jb0  = [FState + d_offs]  ;;    // Circular buffer base address = delay line
                                   // buffer base address


   .align_code 4;
outerloop:
   k1   = [FState + h_offs]  ;     // Set k1 to point to coefficient buffer
   LC0  = xR24               ;;    // Number of iterations for innerloop (LC0) =
                                   // Number of coeffs/4

   // Load input samples and coeffs.
   R7:4   = q[Xin+=4]        ;;    // Get x[i+k+3]:x[i+k] from input sample buffer
   R11:8  = q[k1+=4]         ;;    // Get c[k+3]:c[k] from coefficient buffer
   R19:16 = R7:4             ;     // Save x[i+k+3]:x[i+k] for later store in
                                   // delay line

   // Perform initial complex mult between data and coeffs and store in cleared
   // MACs.
   xMR3:2 += R7 ** R8  (CI)  ;     // y[i+3] = 0 + x[i+k+3] ** c[k]
   yMR3:2 += R5 ** R8  (CI)  ;;    // y[i+1] = 0 + x[i+k+1] ** c[k]
   xMR1:0 += R6 ** R8  (CI)  ;     // y[i+2] = 0 + x[i+k+2] ** c[k]
   yMR1:0 += R4 ** R8  (CI)  ;     // y[i+0] = 0 + x[i+k+0] ** c[k]
   R3:0    = CB Q[j0+=-4]    ;;    // Get x[i+k-1]:x[i+k-4]


   .align_code 4;
innerloop:
   // Iterate through filter length
   xMR3:2 += R6 ** R9  (I)   ;     // y[i+3] = y[i+3] + x[i+k+2] ** c[k+1]
   yMR3:2 += R4 ** R9  (I)   ;;    // y[i+1] = y[i+1] + x[i+k+0] ** c[k+1]
```

```
xMR1:0 += R5 ** R9  (I)   ;     // y[i+2] = y[i+2] + x[i+k+1] ** c[k+1]
yMR1:0 += R3 ** R9  (I)   ;     // y[i+0] = y[i+0] + x[i+k-1] ** c[k+1]
R23:20  = q[k1+=4]        ;;    // Get c[k+7]:c[k+4]
xMR3:2 += R5 ** R10 (I)   ;     // y[i+3] = y[i+3] + x[i+k+1] ** c[k+2]
yMR3:2 += R3 ** R10 (I)   ;;    // y[i+1] = y[i+1] + x[i+k-1] ** c[k+2]
xMR1:0 += R4 ** R10 (I)   ;     // y[i+2] = y[i+2] + x[i+k+0] ** c[k+2]
yMR1:0 += R2 ** R10 (I)   ;     // y[i+0] = y[i+0] + x[i+k-2] ** c[k+2]
R9:8    = R21:20          ;;    // Use c[k+5]:c[k+4]
xMR3:2 += R4 ** R11 (I)   ;     // y[i+3] = y[i+3] + x[i+k+0] ** c[k+3]
yMR3:2 += R2 ** R11 (I)   ;     // y[i+1] = y[i+1] + x[i+k-2] ** c[k+3]
R7:4    = R3:0            ;;    // Shift x[i+k-1]:x[i+k-4] into x[i+k+3]:x[i+k]
xMR1:0 += R3 ** R11 (I)   ;     // y[i+2] = y[i+2] + x[i+k-1] ** c[k+3]
yMR1:0 += R1 ** R11 (I)   ;     // y[i+0] = y[i+0] + x[i+k-3] ** c[k+3]
R11:10  = R23:22          ;;    // Use c[k+7]:c[k+6]
xR15:14 = MR3:2, MR3:2 += R7 ** R8 (I);  // y[i+3] = y[i+3] + x[i+k-1] ** c[k+4]
yR15:14 = MR3:2, MR3:2 += R5 ** R8 (I);; // y[i+1] = y[i+2] + x[i+k-3] ** c[k+4]
if NLC0E, JUMP innerloop                 ;  // All filter taps computed?
xR13:12 = MR1:0, MR1:0 += R6 ** R8 (I);  // y[i+2] = y[i+2] + x[i+k-2] ** c[k+4]
yR13:12 = MR1:0, MR1:0 += R4 ** R8 (I);  // y[i+0] = y[i+0] + x[i+k-4] ** c[k+4]
R3:0    = CB Q[j0+=-4]                   ;; // Get x[i+k-5]:x[i+k-8]
j0=j0+8 (CB)              ;
sR12 = COMPACT R13:12 (IS);;    // Transfer result from MACs and compact
                                // from 32-bit to 16-bit (with saturation).
CB q[j0+=j31] = xR19:16   ;     // Store x[i+k+3]:x[i+k] in delay line buffer
sR13 = COMPACT R15:14 (IS);;    // Transfer result from MACs and compact from
                                // 32-bit to 16-bit (with saturation).

.align_code 4;
if NLC1E, JUMP outerloop  ;     // All samples computed?
q[Yout+=4] = R13:12       ;;    // Store 4 output samples in output buffer.
[FState + p_offs] = j0    ;;    // Save j0 to point to latest sample in delay
                                // line, i.e x[i+k-1]
```

fir16_real.asm

```
.section program;
.global  fir16 real;

_fir16_real:
```

```
// Local defines
#define Yout j4
#define Xin j5
#define INPUT_LEN j6
#define FIR_STATE j7
// Offsets to state struct elements
#define coeff_offs 0
#define delay_offs 1
#define idx_offs 2
#define nof_coeff_offs 3

//PROLOGUE
   J26 = J27 - 64;            K26 = K27 - 64;;
   [J27 += -28] = CJMP;K27 = K27 - 20;;
   Q[J27 + 24] = XR27:24;     Q[K27 + 16] = YR27:24;;
   Q[J27 + 20] = XR31:28;     Q[K27 + 12] = YR31:28;;
   Q[J27 + 16] = J19:16;      Q[K27 + 8 ] = K19:16;;
   Q[J27 + 12] = J23:20;      Q[K27 + 4 ] = K23:20;;
//PROLOGUE ENDS

k0 = [FIR_STATE + coeff_offs];;
k0 = k0 + k0;; // Times 2 for short data access
j0 = k0;;
j0 = j0 + 0x1;;

xR3:0   = sDAB q[k0 += 0x8]; // Preload filter coeffs to CBX
yR3:0   = sDAB q[j0 += 0x8];; // Preload skewed coeffs copy into CBY
xR3:0   = sDAB q[k0 += 0x8];
yR3:0   = sDAB q[j0 += 0x8];;

xR20 = INPUT_LEN;;
// Expand the coeffs into 2 identical short words(16 bits) each
SR11:8 = MERGE R1:0, R1:0;;
SR15:12 = MERGE R3:2, R3:2;;
// Divide by two since two outputs are calculated simultaneously
xR20 = ASHIFT R20 by -1;;

j11 = -10;; // increment for i/p pointer

j0 = Xin + 0x2; // The first data will be picked from delay line
LC0 = xR20 ;;
R27:24 =  DAB q[j0 += 4];; // Prefetch
R27:24 =  DAB q[j0 += 4];;
j8 = [FIR_STATE + delay_offs];; // Get pointer to delay line

///////// Loop over number of input samples ////////
.align_code 4;
loop_:
R3:0 = R27:24;;
MR3:0 += R9:8   * R25:24 (CI);
R31:28 = DAB q[j0 += 4];;
MR3:0 += R11:10 * R27:26 (I);
R27:24 =  DAB q[j0 += j11];;
MR3:0 += R13:12 * R29:28 (I);
R27:24 =  DAB q[j0 += 4];;
MR3:0 += R15:14 * R31:30 (I);
R27:24 =  DAB q[j0 += 4];;
sR23:22 = COMPACT MR3:0 (IS);;
```

```
R7:4 = R31:28;;
sR21 = R23 + R22;;
if NLC0E, jump loop ; l[Yout += 0x2] = xyR21;;
/////////// End of loop  ////////////////
q[j8 += 0x4] = xR3:0;; // Store delay line
q[j8 += j31] = xR7:4;;

// EPILOGUE STARTS
   CJMP = [J26 + 64];;
   YR27:24 = q[K27 + 16];     XR27:24 = q[J27 + 24];;
   YR31:28 = q[K27 + 12];     XR31:28 = q[J27 + 20];;
   K19:16  = q[K27 + 8 ];     J19:16  = q[J27 + 16];;
   K23:20  = q[K27 + 4 ];     J23:20  = q[J27 + 12];;
   CJMP (ABS); J27:24=q[J26+68]; K27:24=q[K26+68]; nop;;
// EPILOGUE ENDS

_fir16_real.end:
```

Listing 8. fir16_real.asm

文件历史

版本	描述
第一版—2004年1月13日，Klas Brink提供	第一版