Application Note, V1.1, October 2007

AP16121

XC2000 & XE166 Families

Implementation of FIR and IIR Filter Based on the XC2000 & XE166 Families

Microcontrollers

Never stop thinking

Edition 2007-10

Published by
Infineon Technologies AG
81726 Munich, Germany

AP16121

Revision History:	2007-10	V1.1
Previous Version:	none	

Page	Subjects (major changes since last revision)

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:

mcdocu.comments@infineon.com

Table of Contents **Page**

1 Introduction

The FIR (Finite Impulse Response) filter and IIR (Infinite Impulse Response) filter are two kinds of digital filters used in digital signal processing. The definitions of FIR and IIR filters are not complicated, but their implementations are quite different. Firstly, there are many implementation structures for FIR and IIR filters.. Secondly, the implementation of FIR and IIR filters is hardware-dependent. The performance of the implemented FIR and IIR filter depends on different factors, e.g. hardware used (microcontroller and DSP processor), implementation structure used, compiler used, and so on. In this application note we introduce how to implement the FIR and IIR filters based on Infineon the XC2000/XE166 microcontroller families with DSP MAC unit. This application note consists of two parts, the first part for FIR filter and the second part for IIR filter.

2 FIR Filters

The FIR (Finite Impulse Response) filter, as its name suggests, will always have a finite duration of non-zero output values for a given finite duration of non-zero input values. FIR filters use only current and past input samples and none of the filters previous output samples, to obtain a current output sample value.

For causal FIR systems, the system function has only zeros (except for poles at z=0). The FIR filter can be realized in many forms. But the transversal and lattice forms are in practice most useful.

2.1 Transversal FIR Filter

A transversal FIR filter is realized by a tapped delay line. The delay line stores the past input values. The input x(n) for the current calculation will become x(n-1) for the next calculation. The output from each tap is summed to generate the filter output. For a general N tap FIR filter, the difference equation is:

$$y(n) = \sum_{i=0}^{N-1} h(i)x(n-i) \tag{1}$$

Where:
x(n) : the filter input for nth sample,
y(n) : the output of the filter for nth sample,
h(i): filter coefficients,
N: filter order.

The filter coefficients, which decide the scaling of current and past input samples stored in the delay line, define the filter response.

The transfer function of the filter in Z-transform is:

$$H(z) = \frac{Y(z)}{X(z)} = \sum_{i=0}^{N-1} h(i)z^{-i} \tag{2}$$

Figure 1 shows the implementation of transversal Fir filter.

2.2 Lattice FIR Filter

The structure of a lattice FIR filter is showed in Figure 2. Each stage of the filter has an input and output that are related by the equations:

$$\begin{aligned} y_i(n) &= y_{i-1}(n) + k_i u_i(n-1) \\ u_i(n) &= k_i y_{i-1}(n) + u_{i-1}(n-1) \end{aligned} \qquad 1 \le i \le M \tag{3}$$

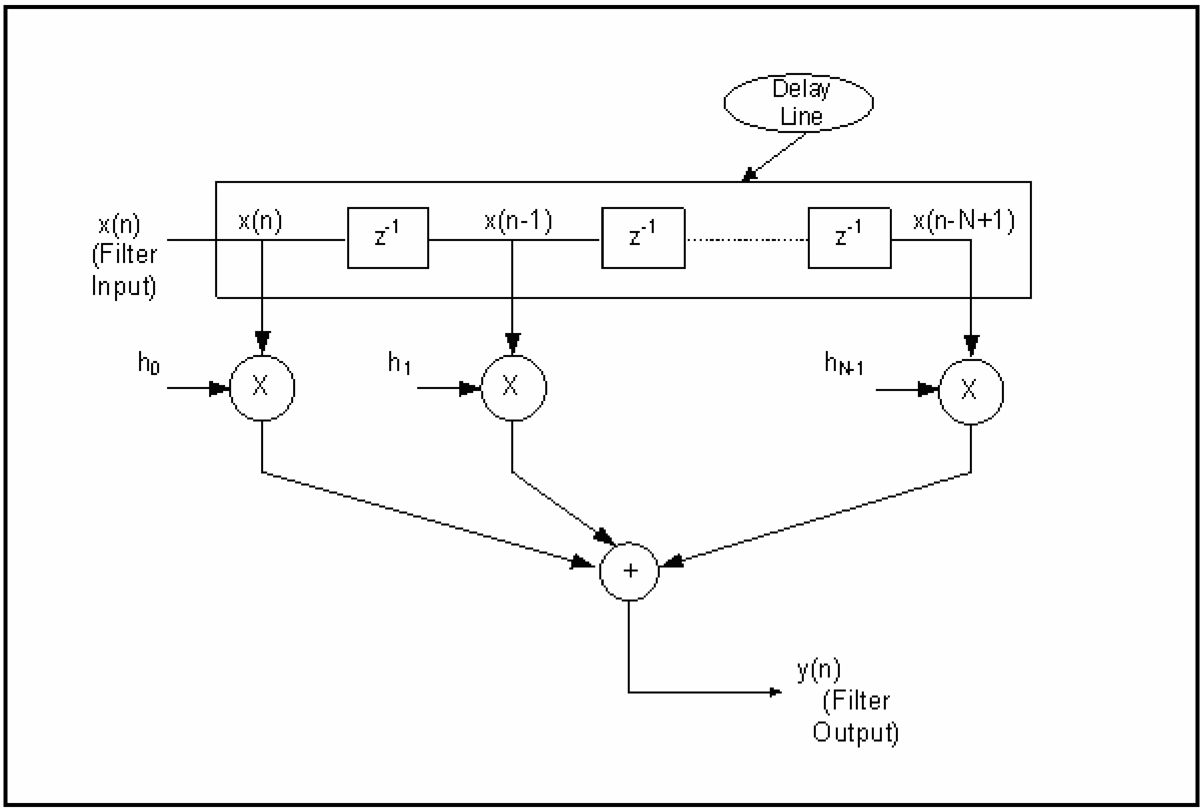

Figure 1: Block Diagram of FIR Filters

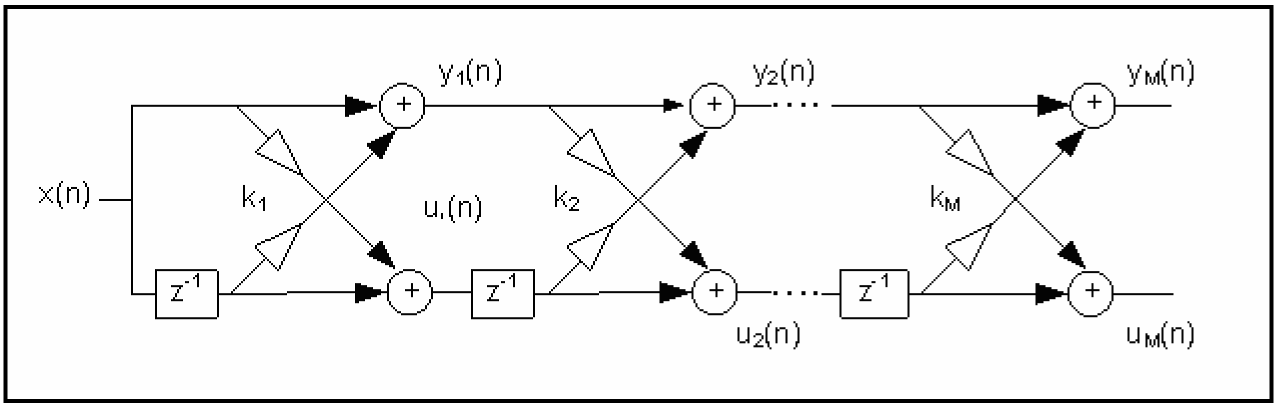

Figure 2: Lattice FIR Filter

The initial values are equal to the filter input x(n):

$$\begin{aligned} y_0(n) &= x(n) \\ u_0(n) &= x(n) \end{aligned}$$

At the last stage we have the output of the lattice FIR filter $y(n)=y_M(n)$.

2.3 Multirate FIR Filters

Multirate filters are a kind of digital filters that change the sampling rate of a digital signal. A multirate filter converts a digital signal with sampling rate M to another digital signal with sampling rate N. The both digital signals represent the same analog signal at different sampling rates. A multirate filter can be realized in an FIR filter or an IIR filter. Due to the advantages of FIR filters, such as linear phase, unconditional stability, simple structure and easy coefficient design, most of the multirate filters are implemented with FIR filters. Here we describe only FIR multirate filters.

The basic operations of the multirate filters are decimation and interpolation. Decimation reduces the sample rate of a signal and can be used to eliminate redundant or unnecessary information contained in the signal. Interpolation increases the sample rate of a signal through filling in missing information between the samples of a signal based on the calculation on the existing data.

2.4 FIR Decimation Filter

The FIR decimation filter can be described using the equation

$$y(m) = \sum_{k=0}^{N-1} h(k)x(mM-k) \tag{4}$$

where h(k) is filter coefficient vector, x(n) is the input signal and M represents the decimation factor. Figure 3 shows the block diagram of an FIR decimation filter. Its equivalent form is showed in Figure 4 .

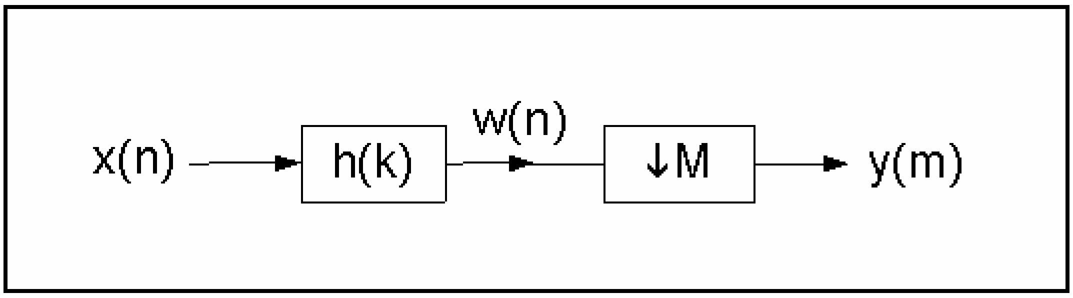

Figure 3: Block Diagram of FIR Decimation Filter

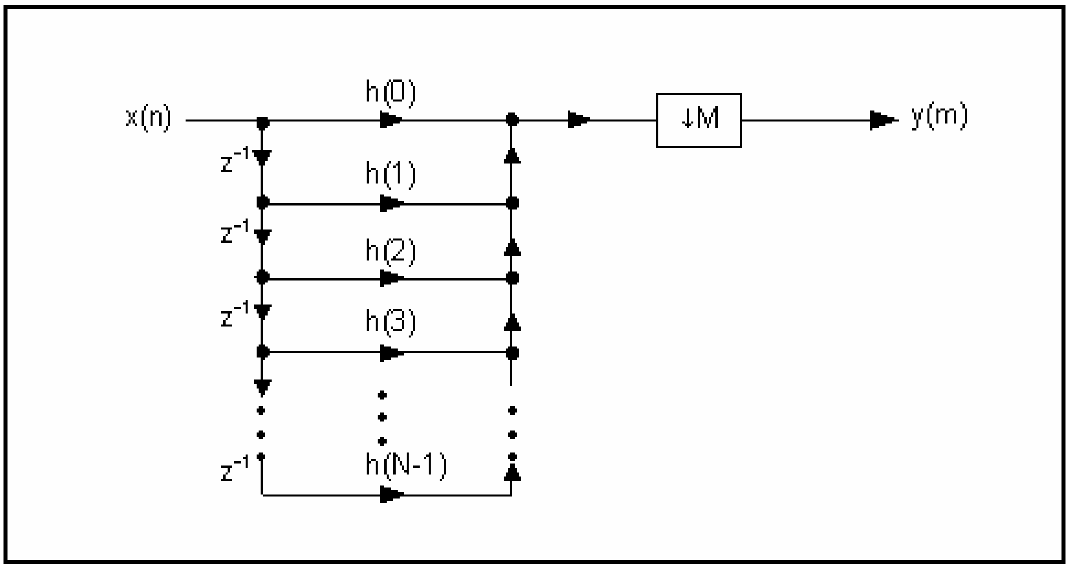

Figure 4: Equivalent Implementation of FIR Decimation Filter

2.5 FIR Interpolation Filter

In comparison with the decimation filter, the interpolation filter can be used in the reconstruction of a digital signal from another digital signal. Figure 5 shows a block diagram of an interpolation filter, where the low-pass filter of the interpolator uses an FIR filter structure. An FIR interpolation filter can be described as

$$y(m) = \sum_{k=0}^{N-1} h(k)x((m-k)/L) \tag{5}$$

for m-k=0, L, 2L, ⋯ . An equivalent implementation of the FIR interpolation filter is showed in Figure 6 .

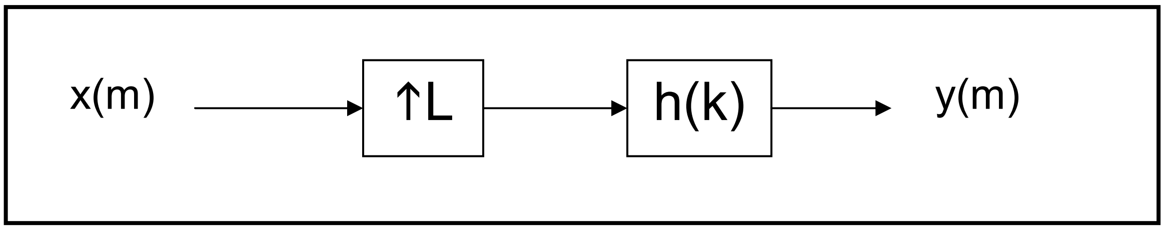

Figure 5: Block Diagram of FIR Interpolation Filter

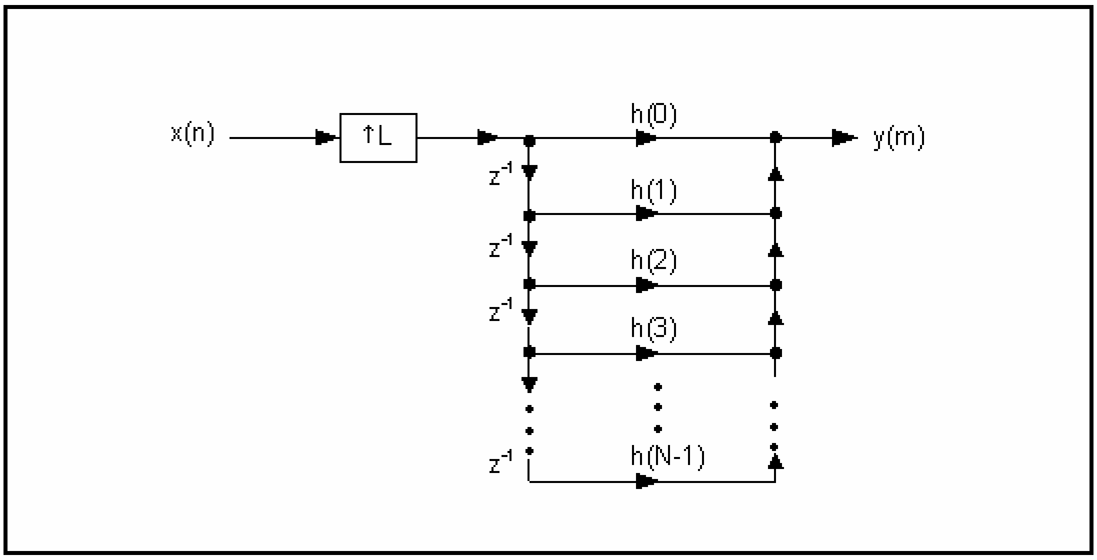

Figure 6: Equivalent Implementation of FIR Interpolation Filter

2.6 Implementation of FIR Filters on XC2000/XE166 Microcontrollers

The XC2000 & XE166 microcontrollers have a MAC unit (Multiply Accumulate) that is dedicated for DSP (Digital Signal Processing) operations. In the following section we describe how to implement an FIR filter using MAC instructions.

The implementation uses the transversal structure (Direct Form) of the FIR filter showed in Fig. 1. A single input is processed at a time and output for every sample is returned. The filter operates on 16-bit real input, 16-bit coefficients and gives 16-bit real output. The number of coefficients given by the user is arbitrary. The delay line is implemented in parallel to the multiply-accumulate operation using instructions CoMACM. The delay buffer will be located in the DPRAM area. The implementation is based on following pseudo C code:

Pseudo C code:

```
{

   short x(N_h)={0,...};      //Input vector
   short Y;                   //Filter result
   short i;

   //Update the input vector with the new input value
   for(i=0; i<N_h-1; i++)
     x(i) = x(i+1);
   x(N_h-1) = IN;                 //move the new input unto X[N_h-1]

   //Calculate the current FIR output
   Y = 0;
   for(i=0; i<N_h; i++)
     Y = Y + h(i)*x(N_h-1-i)      //FIR filter output
  return Y;                       //Filter output returned

}
```

The function is implemented in assembly with MAC instructions due to the optimization requirement. For implementation special attention must be taken to account for the argument (variable) transfer from C function to assembly function, because different compilers have different conventions of variable transfer. Table 1 summarizes the convention of parameter transfer for commonly used compilers.

Table 1: Registers used for parameter passing in different compilers

Parameter type	**Registers used**		
	Viper	**Classic Tasking**	**Keil**
1 bit (bit)	USR0, R2.0..15, R3.0..15, R4.0..15, R5.0..15	Not allowed	R15.0..15
8 bit (char)	RL2, RH2, RL3, RH3, RL4, RH4, RL5, RH5	RL12,RH12,RL13,RH13, RL14,RH14,RL15,RH15	RL8, RH8, RL9, RH9, RL10, RH10, RL11, RH11
16 bit (short, int)	R2, R3, R4, R5, R11, R12, R13, R14	R12,R13,R14,R15	R8, R9,R10,R11,R12
32 bit (long, float)	R2R3, R4R5, R11R12, R13R14	R12R13, R14,R15	R8R9, R10R11
64 bit (double)	R2R3R4R5, R11R12R13R14	R12R13R14R15	R8R9R10R11
64 bit (long long)	R2R3R4R5, R11R12R13R14	Not supported	Not supported

If the FIR filter function is written directly in C, the compiler will automatically take care of the parameter transfer. But, If the function is written in assembly, the parameter passing needs to be handled carefully. So far, just the Viper compiler supports the MAC unit and can automatically generate the MAC instructions.

Here are the assembly codes for FIR filter with the classic Tasking compiler:

```
$EXTEND
$CASE

NAME  Fir_16

;============ Define registers ===================================

h_ptr        LIT    'R12'  ; pointer of coefficient vector h
X_in         LIT    'R13'  ; pointer of new input sample
N_h          LIT    'R14'  ; filter order
buffer_ptr   LIT    'R15'  ; pointer of delay buffer

;-------------- Section for Code Segment Declaration --------------

CODE_FILTER SECTION CODE WORD GLOBAL

GLOBAL _Fir_16

;=====================Procedure Declaration =========================
_Fir_16 PROC FAR

   ASSUME DPP3:SYSTEM
;MAC registers initialization
   MOV    MCW,#0600h     ; MP=1, MS=1
   SUB    N_h,#2h        ;(R14)=N_h-2
   MOV    MRW,N_h        ;(MRW)=N_h-2, repeat counter

;ESFR register initialization
   ADD    N_h,#1           ;(R14)=N_h-1
   SHL    N_h,#1           ;(R14)=2*(N_h-1)
   EXTR #3                 ; Next 2 instructions use ESFR space
   MOV    IDX0,buffer_ptr  ;(IDX0)=DPRAM_add
   MOV    QX0,N_h          ;(QX0)=2*(N_h-1)
   MOV    QR0,N_h

;Read the new input sample and move it
;at x(n-1) address overwriting x(n-1)
   CoMOV   [IDX0],[X_in]          ;x(n-1) <- IN
   CoNOP   [IDX0+QX0],[h_ptr+QR0]       ;return the pointers to beginning
;FIR: first multiplication, h(N_h-1)*x((n-N_h+1)
 CoMUL  [IDX0-],[h_ptr-]                ;(ACC)=h(N_h-1)*x(n-N_h+1)<<1
                                        ;(IDX0)=(IDX0)-2
                                        ;(R12)=(R12)-2
;FIR loop: repeat N_h-1 times the same MAC instruction
loop1:                                ;i=(N_h-2):(-1):0
-USR1   CoMACM [IDX0-],[h_ptr-]       ;(ACC)=(ACC)+h(i)*x(n-i)<<1
   JMPA cc_nusr1,loop1

;return the pointers
   CoNOP   [IDX0+],[h_ptr+]

;Rounding
   CoRND

;write the 16-bit filter output y(n) into registers R4
   CoSTORE   R4,MAH      ;(R4)=(MAH)

   RET
_Fir_16ENDP

CODE_FILTER  ENDS

REGDEF R0-R15
END
```

Another possibility to force the compiler to use MAC instructions is to use intrinsic functions provided by the compiler. All three compilers provide MAC intrinsic functions. Here is an example using the Tasking Viper compiler to implement the FIR filter using intrinsic functions.

Pseudo C code:
```
{

   short x(N_h)={0,...};      //Input vector
   short Y;                   //Filter result
   short i;

   //Update the input vector with the new input value
   for(i=0; i<N_h-1; i++)
     x(i)  = x(i+1);
   x(N_h-1)  = IN;                //move the new input unto X[N_h-1]

   //Calculate the current FIR output
   Y = 0;
   for(i=0; i<N_h; i++)      //FIR filter output
   {
      __CoLOAD(Y);
      __CoMAC(h(i),x(N_h-1-i);
   }

   Y = CoSTOREMAS();
    return Y;                     //Filter output returned

}
```

3 IIR Filters

In comparison with Finite Impulse Response (FIR) filter, the Infinite Impulse Response (IIR) filter is an efficient way of achieving a certain of performance characteristics with a given filter order. This is because the IIR filter incorporates feedback and is capable of realizing both poles and zeros of a system transfer function, whereas the FIR filter can only realize the zeros. Furthermore, because of the recursive structure, the IIR filter can be efficiently implemented with less cycles on microcontrollers and DSP processors. So, IIR filters are often used in embedded applications.

In general, a causal IIR filter can be represented by a difference equation, where the output signal at time n is obtained as a linear combination of samples of the input and output signals at previous time instants. The difference equation of an IIR filter has the form

$$y(n)=\sum_{i=1}^{N}a(i)y(n-i)+\sum_{i=0}^{M}b(i)x(n-i) \tag{5}$$

where x(n) and y(n) are input and output. a(i) and b(i) are filter coefficients corresponding to poles and zeros, respectively.

There are different ways of implementing the above difference equation. The implementations depend on the processor. Each processor has its own implementation approaches with different memory locations, different data formats, and so on. In this application note we want to introduce how to implement an IIR filter using the Infineon XC2000 & XE166 microcontroller families and to provide the performance comparison of different structures.

3.1 Direct Form 1

The simple way to realize an IIR filter is to use the difference equation directly. Fig. 6 shows the signal flow of the difference equation with N=M. Such a realization is called Direct Form 1 implementation of IIR filters.

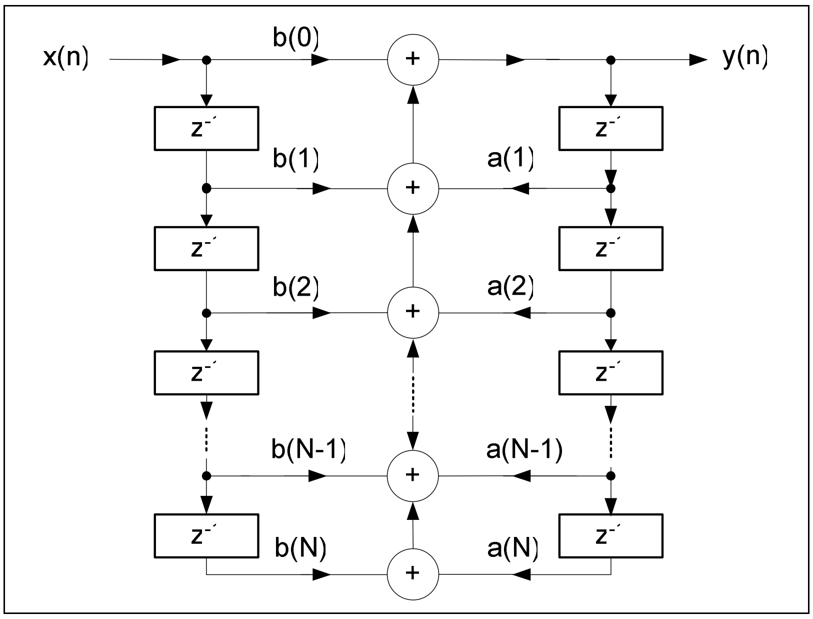

Fig.6. Structure of Direct Form 1 Implementation

The left part of the signal flow implements zeros, and the right part of the signal flow realizes poles. From Fig.6 we can see that the implementation with Direct Form 1 needs 2N memory units to save the state variables. Fig.8 gives the program flow of the implementation with Direct Form 1.

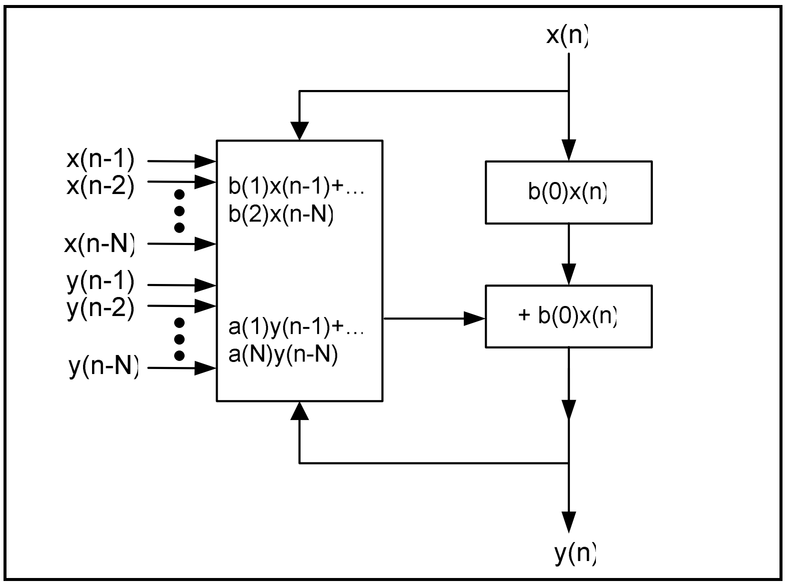

Fig.7. Program Flow of Direct Form 1 Implementation

Following are the C pseudo codes of the program flow in Fig. 7.

```
; x(n) = input signal at time n
; y(n) = output signal at time n
; a(k), b(k) = IIR filter coefficients
; N refer to the filter order

int  a[N], b[N+1];      //Filter vectors
int  y[N], x[N+1];      //input and output signal vectors
int  i, temp;

;Move the new input sample into input vector in DPRAM
for(i=N; i>0; i--)
  x(n-i-1) = x(n-i);
x(n) = IN;                //New input sample

;IIR filtering
y(n) = 0;
for(i=0 to N)
   y(n) = y(n)+b(i)*x(n-i);

for(i=1 to N)
   y(n) = y(n)+a(i)*y(n-i);

Y = y(n)

return Y;          //Filter Output returned
```

The memory location of the data is showed in Fig.8, where 2N past inputs and outputs are located in Dual-Port RAM (DPRAM) as static variables. Two pointers are used in the indirect addressing. Register R12 points to the coefficient vector. The special register IDX0 is used as pointer to the static memory space. Assembly instruction CoMACM executes the multiply-accumulation with additional data moving operation. After one sample IIR filtering the DPRAM is updated with the current input and output samples.

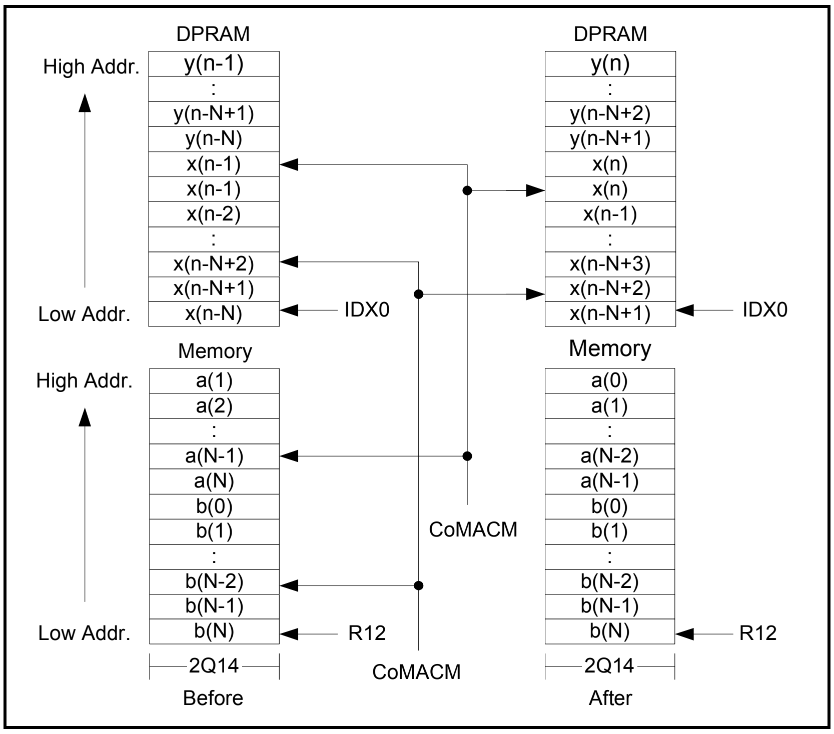

Fig. 8. Memory Location of Direct Form 1 Implementation

3.2 Direct Form 2

We have said that the Direct Form 1 implementation needs 2N static memory units. To reduce the memory occupation and make the computation more efficient, we can manipulate the signal flow in Fig.6. From Fig.6 we know that the structure of Fig.6 is cascaded by two blocks, each being linear and time invariant. Therefore, the two blocks can be commuted without changing the input-output behavior. Moreover, from the block exchange we get the flow graph with two side-to-side stages of pure delays. The two stages can be combined in one. The realization of those transforms is showed in Fig.9 and it is called Direct Form 2 implementation of an IIR filter.

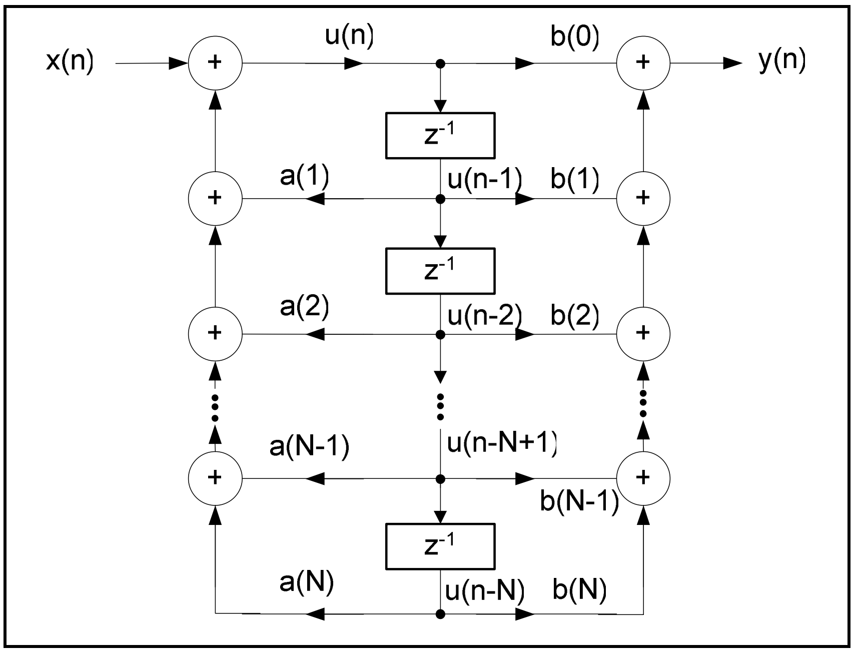

Fig.9. Direct Form 2 Implementation of IIR Filters

The program flow is given in Fig.10, where u(i) is state variable and needs N static memory units.

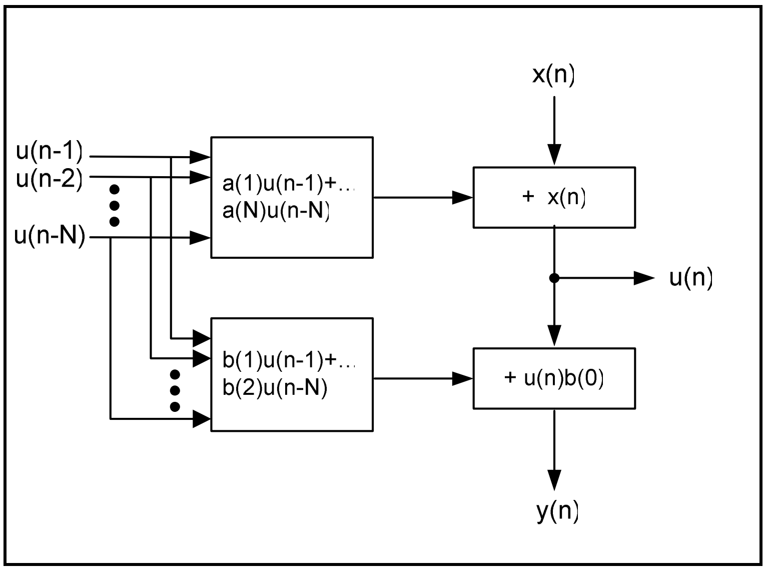

Fig.10. Program Flow of Direct Form 2 Implementation

The C pseudo code of the program flow in Fig.10 reads

```
; x(n) = input signal at time n
; u(n) = state variable at time n
; y(n) = output signal at time n
; a(k), b(k) = IIR filter coefficients
; N = M refer to the filter order

int  a[N], b[N+1];      //Filter vectors
int  u[N+1];            //state variable vector
int  i;

;Calculate the sate variable at time n, u(n)
u(n) = x(n);
for(i=1; i<N; i++)
  u(n) = u(n) + a(i-1)*u(n-i);

;Calculate the output at time n
y(n) = 0;
for(i=0; i<=N; i++)
   y(n) = y(n)+b(i)*u(n-i);

Y = y(n)

return Y;         //Filter Output returned
```

The memory location is showed in Fig.11, where the state variable u(i) is located in DPRAM and needs N memory units. Similar with the Direct Form 1 implementation two pointers R12 and IDX0 are needed for indirect addressing. The assembly instruction CoMAC executes just multiply-accumulation operation without additional data moving. After filtering the static variable memory will be updated. The pointer IDX0 points to the updated state variable space.

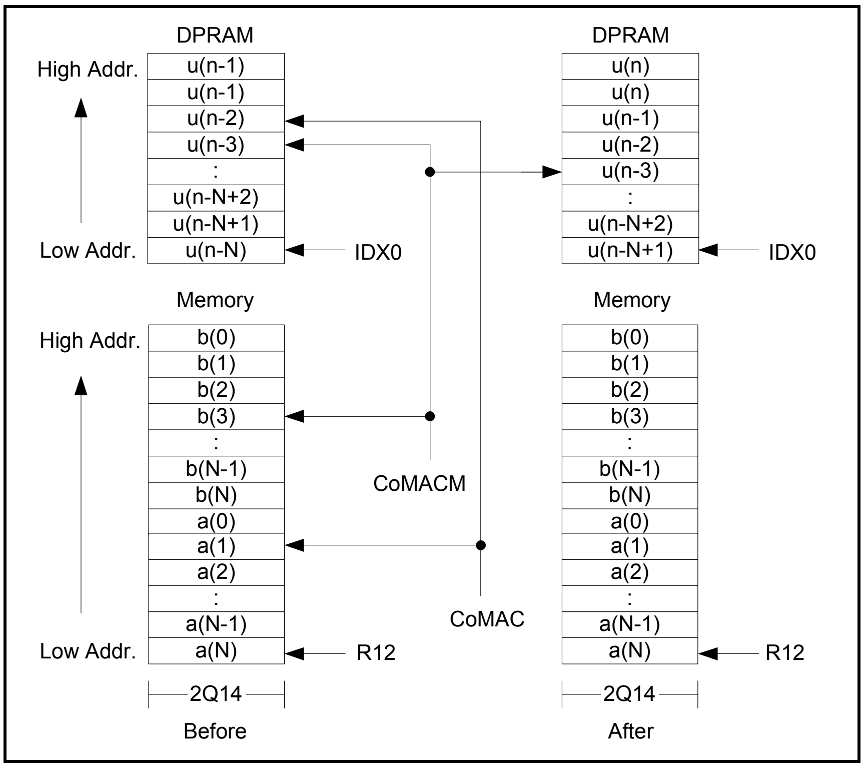

Fig. 11. Memory Location of Direct Form 2 Implementation

3.3 Transposed Form 2

Another transformation that can be done on the signal flow in the Fig.9 without changing its input-output relationship is the transposition. The transposition of a signal flow graph is done with the following operations:

- Inversion of the direction of all the edges
- Transformation of the nodes of addition into branching nodes, and vice-versa
- Exchange of the roles of the input and output edges

The transposition of the realization in Direct Form 2 leads to the Transposed Form 2 implementation of the IIR filter showed in Fig.12. The implementation of the Transposed Form 2 is similar with Direct Form 2.

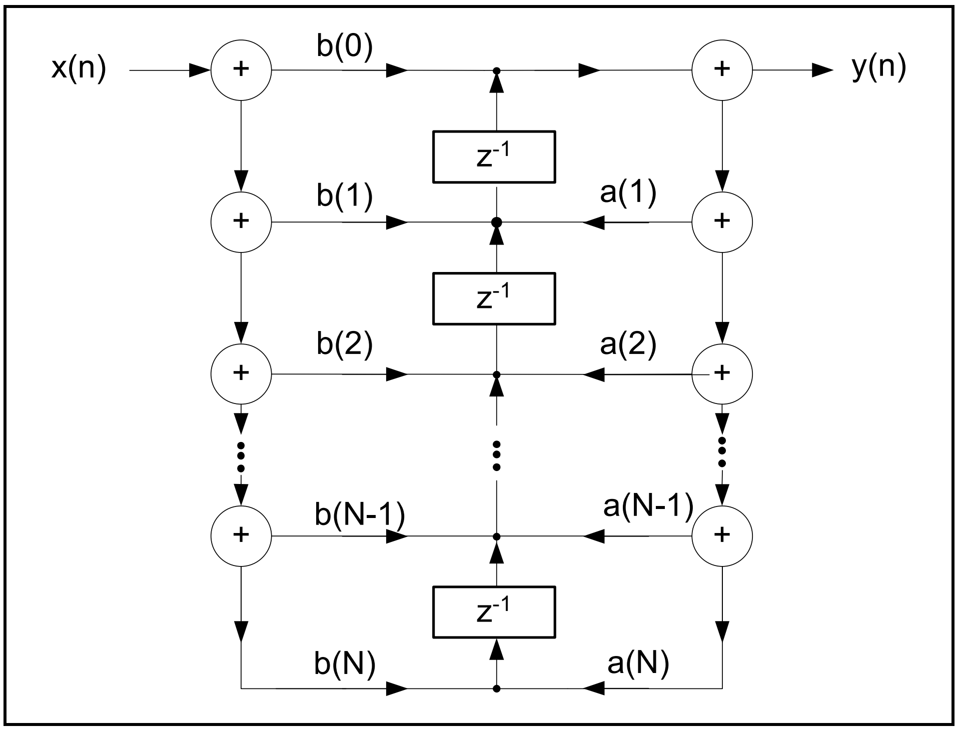

Fig.12. Transposed Form 2 Implementation

3.4 Cascaded Biquad with Direct Form 2

By designing higher-order IIR filters one of the ways is to use the direct form implementation with the complicated simple section. Another way is to cascade several biquad sections with appropriate coefficients. The biquad implementation executes slower but generates smaller numerical errors than the direct form implementation. The biquads can be scaled separately and then cascaded in order to minimize the coefficient quantization and the recursive accumulation errors. The coefficients and data in the direct form implementation must be scaled all at once, which gives rise to large errors. Another disadvantage of the direct form implementation is that the poles of such single-stage high-order polynomials get increasingly sensitive to quantization errors. The second-order polynomials sections are less sensitive to quantization effects.

Fig.13 shows the structure of the cascaded biquads implemented with Direct Form 2 in each biquad section.

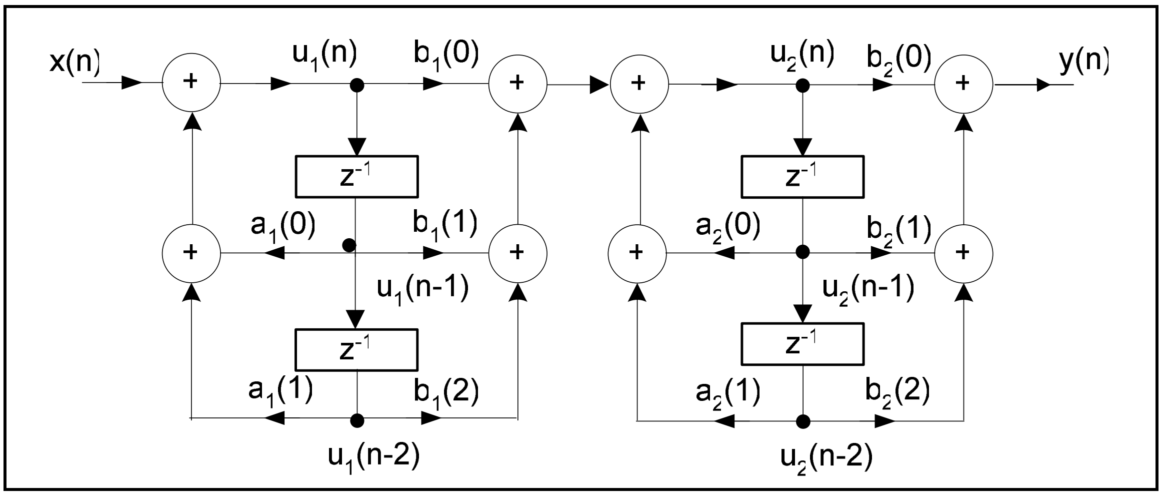

Fig.13. Cascaded Biquad IIR Filter with Direct Form 2 in Each Section

The program flow is described in Fig.14.

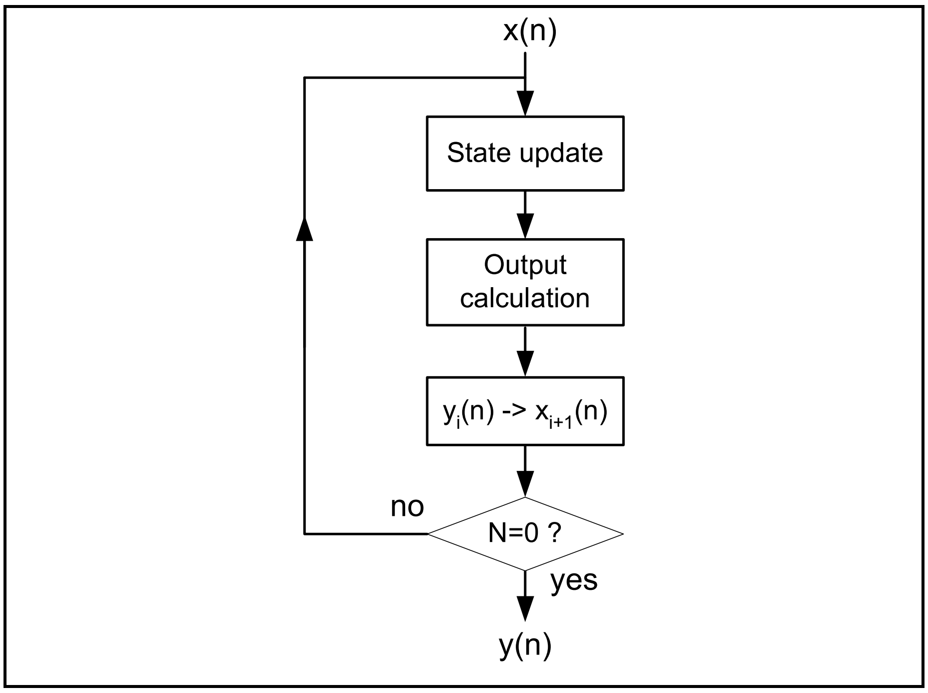

Fig.14. Program Flow of Cascaded Biquad Implementation with Direct Form 2 in Each Section

The C pseudo code looks like

```
; xi(n) = input signal at time n of biquad number i
; ui(n) = state variables at time n of biquad number i
; yi(n) = output signal at time n of biquad number i
; ai(k), bi(k) = Filter coefficients of biquad number i
; N = number of biquads

int  ai(n), bi(n);        //Filter vectors
int  ui(n);              //state variable vector
int  yi(n), Y;           //output
int  i;

for(i=1; i<=N; i++)
{
   ;Update the sate variable u_i(n)at time n
   u_i(n) = x_i(n) + a_i(0)*u_i(n-1) + a_i(1)*u_i(n-2);

   ;Calculate the output at time n
   y_i(n) = b_i(0)*u_i(n) + b_i(1)*u_i(n-1) + b_i(2)*u_i(n-2);

   ;Set i-th biquad output to (i+1)-th biquad input
   x_i+1(n) = y_i(n);
}

;Output of the last biquad
Y = y_N-1(n);

return Y;          //Filter Output returned
```

The memory location is depicted in Fig.15, where the state variable u(i) has the same memory occupation as Direct Form 2 implementation. Two pointers IDX0 and IDX1 are used to access the different state variables.

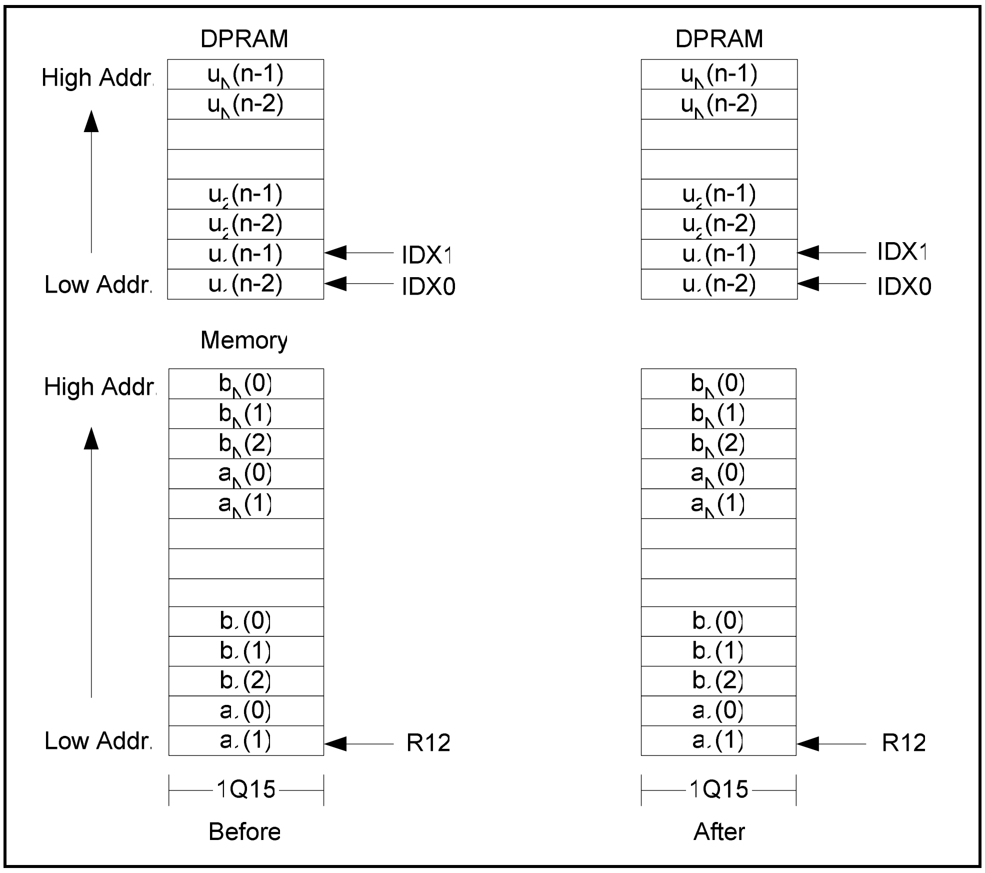

Fig.15. Memory Location of Cascaded Biquad Implementation with Direct Form 2 in each Biquad Section

3.5 Cascaded Biquad with Transposed Form 2

Replacing the biquad section in Fig.13 with Direct Form 2 we get another implementation of the IIR filter showed in Fig.16, and it is called cascaded biquad realization with Direct Form 2. Its implementation is similar with the cascaded biquad with Direct Form 2.

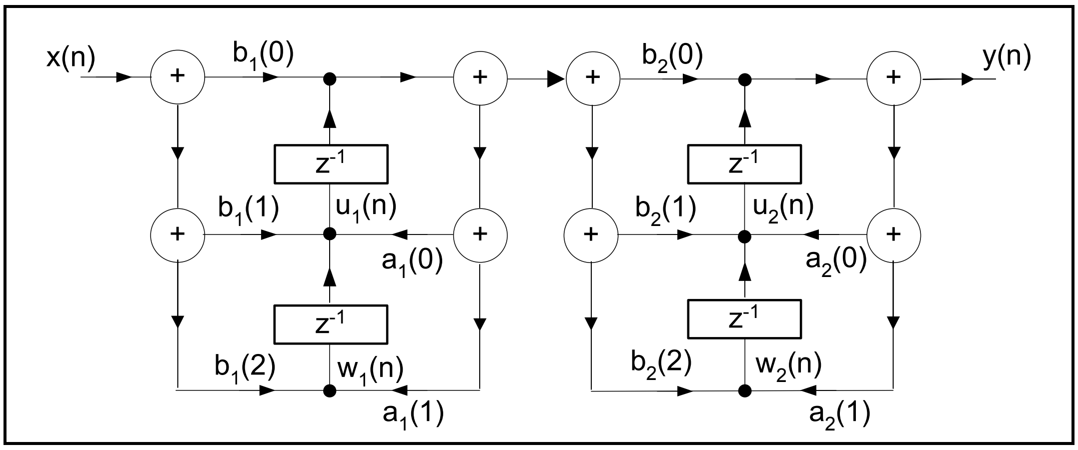

Fig.16. Cascaded Biquad IIR Filter with Transposed Form 2 in each Biquad Section

3.6 Lattice IIR Filters

The lattice IIR filter is used often in the synthesis of speech, most commonly to simulate the vocal tract. Its physical analogue is a series of cylinder of different radii. Each of the filter coefficients respects the amount of energy reflection at a boundary of two cylinders. Fig.17 shows the structure of a lattice IIR filter.

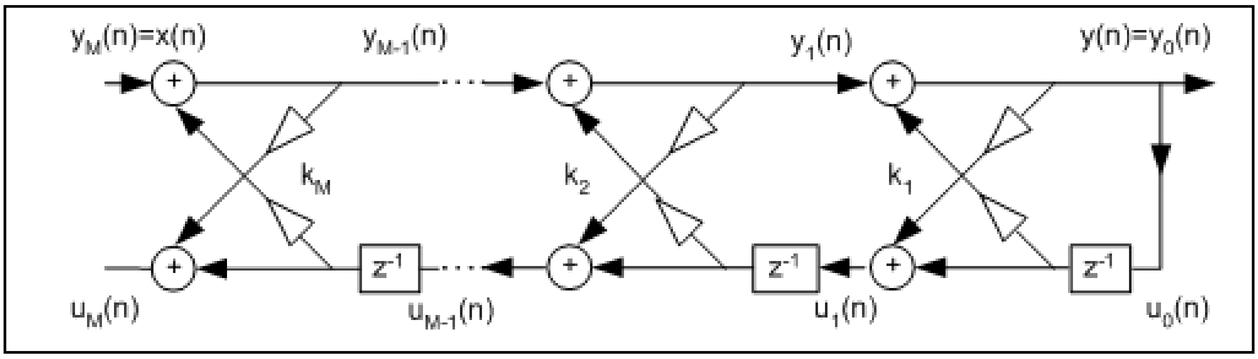

Fig.17. Lattice IIR Filter

The C pseudo code of the implementation of structure in Fig.17 can be written as

```
short x[N_x];              //Input vector
short yn[M]={0,..};        //Output vector of different stages at time n
short K[M];                //Lattice filter coefficient vector
short un[M];          //State variable vector at time n
short un-1[M]={0,...};     //State variable vector at time n-1

short i, j;

for(j=0; j<N_x; j++)
{
  //Initialization
  yn[j] = x[j];
```

```
    //Calculate the output and state vector
    for(i=1; i<M; i++)
    {
        yn[j] = yn[j] - K[i]*un-1[i-1];      //Output
        un[i] = K[i]*yn[j] + un-1[i-1];      //Update the state variable
    }

  }
  return;
```

The memory location of the implementation using the XC2000 is showed in Fig.18, where reflection coefficients are stored in DPRAM, and the static variables u(i) is located in data memory.

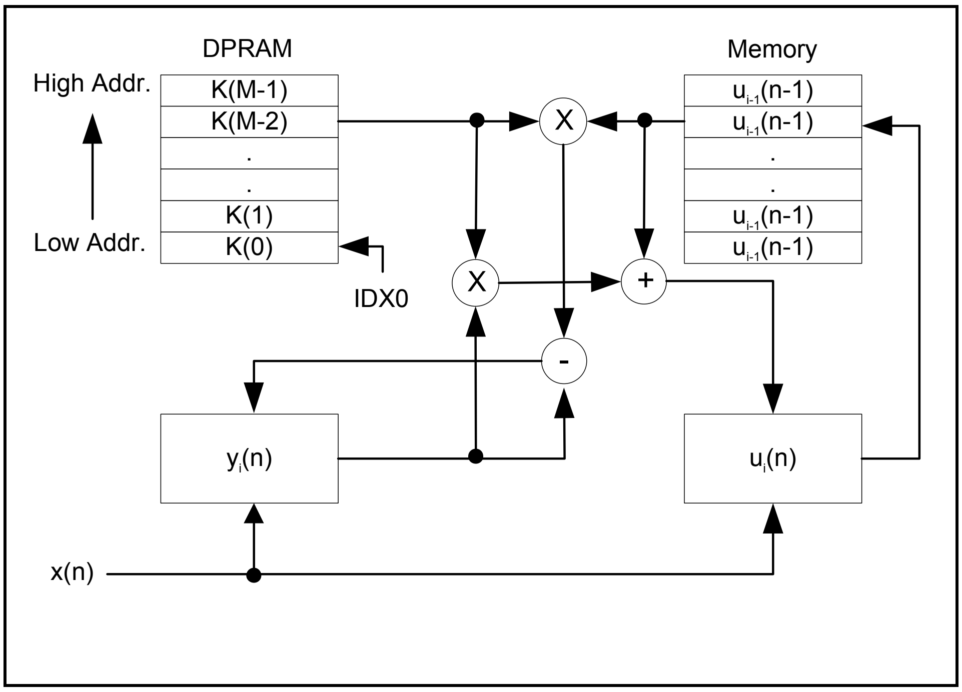

Fig.18. Memory Location of Lattice IIR Filter

4 Conclusion

In this application note we have introduced different implementation structures of the FIR and IIR filters, and their realizations based on XC2000/XE166 microcontrollers. Here we give the performance comparison among six IIR implementations. Following table lists the benchmarks of each implementation on XC2000, where N indicates the filter order, and M is the stage number of the lattice filter.

Table 2: Benchmark of IIR filter on XC2000

	Direct Form			Cascaded Form		Lattice Filter
	Direct Form 1	Direct Form 2	Direct Form 2	With Direct Form 2	With Direct Form 2	IIR
Cycles	2N+23	2N+23	2N+25	12(N/2-1)+23	19(N/2/1)+27	11M+12
Code size	76 bytes	78 bytes	80 bytes	98 bytes	108 bytes	88 bytes
Static memory	2N	N	N	N	N	2M
Quantization errors	Large	large	large	small	small	small
Stability	Sensitive to errors	Sensitive to errors	Sensitive to errors	Less sensitive to errors	Less sensitive to errors	Less sensitive to errors

From the table we can see that the Direct Form 1 and Direct Form 2 are fastest and need minimal cycles. But Direct Form 2 needs less data memory. So, the Direct Form 2 structure is best if the executing speed is critical. However, direct implementations have relatively large quantization errors, and the implemented system is sensitive to quantization errors. Therefore, if the stability and the small quantization errors are required, the cascaded biquad with direct from 2 is the best choice. Infineon provides the free source code for all FIR and IIR implementations introduced in this application note. The routines are realized and optimized on the XC2000 and XE166.

You can download the source code directly from Infineon homepage:
http://www.infineon.com/C166DSPLIB.

5 Reference

[1] Guangyu Wang, User's Manual of XC166Lib, A DSP Library for XC166 Microcontroller Family, V1.1, Feb. 2004.

[2] 16-Bit DSP Library for 16-Bit Microcontroller with C166S V2 Core, V1.1, Feb., 2004, http://www.infineon.com/C166DSPLIB

[3] User Manual: C166S V2 Core, v1.7, January 2001.

[4] XC2200 User's Manual, V1.0 June 2007

[5] Guangyu Wang, “Implementation of DSP Algorithms on Microcontrollers with MAC Unit”, Elektronik, March 2005, Page 80-85.

[6] Guangyu Wang, “DSP Optimization Based on XC166 Microcontroller Architecture”, Electronik, June 2006, Page 50-55.

[7] AP16113, “DSP Optimization Guide for XC2000, XE166 and XC166 Microcontroller Families with MAC Unit”.