



A 16-bit IIR Filter on the ADSP-TS20x TigerSHARC® Processor

Contributed by Rickard Fahlqvist

November 6, 2003

Introduction

This document describes an implementation of a 2nd order 16-bit IIR filter on the ADSP-TS20x TigerSHARC® processor. The implementation uses the direct form II.

Filter Structure

The structure of the 2nd order filter is shown in *Figure 1*. To achieve filters of orders higher than two, several 2nd order filters can be cascaded or connected in a parallel fashion. Note that the resulting filter will have an order that is a multiple of 2. For an odd number of ordered filters, a 1st order filter is needed in addition to one or several 2nd order filters. This is not implemented in the example code presented in *Listing 3*.

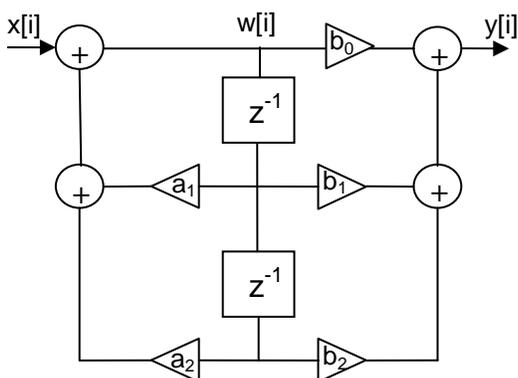


Figure 1. 2nd order direct form II

The equations that characterizes this type of filter structure are described in *Equation 1*.

$$w[i] = x[i] + \sum_{k=1}^2 a_k w[i-k]$$

$$y[i] = \sum_{k=0}^2 b_k w[i-k]$$

Equation 1. Characterizing 2nd order direct form II IIR

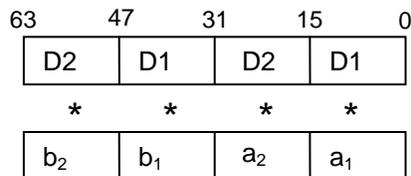
The first numerator coefficient (b_0) can always be set to unity by scaling the input, $x[i]$, with b_0 and dividing b_1 and b_2 by b_0 . This has been used in the example implementation in listing 1 below.

Implementation

Implementing small recursive algorithms on pipelines presents a difficult problem. In this filter, the value of w at iteration i depends on the value of w at iteration $i-1$. This loop-carried dependence limits the degree to which the operations may be pipelined. In this program, all multiplications required to produce $w[i]$ and $y[i]$ are computed in a single cycle. However, to add the input with the partial results in order to get the final output takes 5 cycles. During this time, there is no available parallelism to keep the multiplier busy. As a result, this program can only use one compute unit. The operations used to administer the delay line (as well as the memory fetches) are associated with the ALU, so filling multiple instruction slots per instruction line becomes an issue. As mentioned above, if a

higher order filter is required, you can implement it as parallel 2^{nd} order filters. In this case, two filtered outputs are calculated simultaneously, one in each compute block, with a summation of the results at the end.

The 2-element state is stored in register yR5, and during every iteration each newly computed $w[i]$ is inserted into the state register, which is then shifted left. The delay line is duplicated so that the computation of the 4 multiplications on the feedback and feedforward paths can be performed with one 4-way 16x16-bit multiplication instruction as shown in *Figure 2*.



Dx denotes Delay line element X.

Figure 2. Multiplication of delay line and coefficients

The partial results are collected with the sideways summation instructions. If the coefficients are mixed fraction and integers, the 4-way multiplication must be divided into two separate multiplications, which will impair efficiency.

Because the SDAB (Short Word Data Alignment Buffer) is used, eight 16-bit input words are loaded per iteration where only the first one is used. Memory bandwidth is not the bottleneck, so this seemingly wasteful treatment of the fetched words is appropriate.

Interface

The C style prototype for the filter in the example implementation is noted in *Listing 1*.

```
void iir_16(int2x16 input[],
           int output[],
           int input_len,
           iir_state_t *f_state)
```

Listing 1. IIR function prototype

The typedef “iir_state_t” is a structure that holds coefficients, delay line and the input scaling factor. The structure definition is listed in *Listing 2*.

```
typedef struct
{
  const int2x16 c; /* coefficients */
  int2x16 d; /* start of delay line */
  int2x16 s; /* Input value scaling */
} iir_state_t;
```

Listing 2. Filter state structure

Appendix

In this appendix the assembly source code for the example file is presented.

IIR_16.asm

```

/* *****
 *
 * Copyright © 2003 Analog Devices Inc. All rights reserved.
 *
 * *****/
.section program;
.global _iir_16;

// Local defines
#define Arg0 j4
#define Yout j5
#define Arg2 j6
#define s_p j7
#define Xin j0
#define Xin_tmp j1
#define s_d j2

#define c_offs 0
#define d_offs 1
#define k_offs 2
#define s_offs 3
#define coeff_p j10

_iir_16:
//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
coeff_p = [s_p + c_offs];;
yR24 = [s_p + s_offs]; yR25 = R25 XOR R25;; // Get scale factor
yR21:20 = l[coeff_p += 2];; // Load ai's and bi's
Xin = Arg0 + j31;;
Xin = Xin + Xin; LC0 = Arg2;; // Double it to access shorts; Get # of input samples
yR31 = 0x0010;; // used for FDEP with length=16, position=0
s_d = [s_p + d_offs]; yR4 = R4 XOR R4;;
yR5 = [s_d += j31];; // Load delay line

j10 = j10 - j10;; // Avoid unintended circular buffer
Xin_tmp = Xin;;
yR3:0 = sDAB q[Xin += 8];;
yR4 = R4 OR R5;; // Replicate the two 16-bit states
yR3:0 = sDAB q[Xin += 8];;
Xin_tmp = Xin_tmp + 1; yR11:10 = R5:4 * R21:20;;// Do a1*D1, a2*D2, b1*D1, b2*D2
// and store in R10:11
Xin = Xin_tmp + j31; yR1:0 = R1:0 * R25:24;; // scale

```

```

yR15 = SUM sR10;; // get a1*D1 + a2*D2
yR29:28 = EXPAND sR0(I);; // LSS(R0) -> R28
yR16 = SUM sR11;; // b1*D1 + b2*D2
yR15 = R28 + R15;; // w[i] = x[i] + a1*D1 + a2*D2

.align_code 4;
loop_:
yR5 = LSHIFT R5 by 16; yR4 = R4 XOR R4;; // Shift out oldest element in delay
// line
yR3:0 = sDAB q[Xin += 8]; yR5 += FDEP R15 by R31;; // Get x[i+1] and store the
// new w[i] in delay line
yR3:0 = sDAB q[Xin += 8]; yR17 = R16 + R15;; // y[i] = w[i] + b1*D1 + b2*D2
Xin_tmp = Xin_tmp + 1; yR4 = R4 OR R5;;
yR1:0 = R1:0 * R25:24;; // Scale i/p
Xin = Xin_tmp + j31; yR11:10 = R5:4 * R21:20;; // Do a1*D1, a2*D2, b1*D1, b2*D2 and
// store in R10:11
yR29:28 = EXPAND sR0(I);; // LSS(R0) -> R28
yR15 = SUM sR10;; // get a1*D1 + a2*D2
yR16 = SUM sR11;; // b1*D1 + b2*D2
if NLC0E, jump loop_; yR15 = R28 + R15; [Yout += 1] = yR17;; // w[i+1] = x[i+1] +
// a1*D1 + a2*D2
// 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
_iir_16.end:

```

Listing 3. Source file iir_16.asm

Document History

Version	Description
November 06, 2003 by R. Fahlqvist	First version