DSP-MCOM 2005
Košice, Slovak republic
13-14 September 2005

An Implementation of High Performance IIR Filtration on 2-MAC Blackfin DSP Architecture

Miloš Drutarovský
Department of Electronics and Multimedia Communications
Technical University of Košice
Park Komenského 13, 041 20 Košice
Slovak Republic
Milos.Drutarovsky@tuke.sk

Abstract – **The paper presents a new optimized implementation of IIR biquad filtration routine for high-performance 16-bit fixed point Blackfin DSPs. The core of hand-optimized routine uses both MAC units available in Blackfin architectures and in contrary to other available solutions it does not produce pipeline stalls in the DSP control and MAC units. The proposed core is the fastest currently available IIR biquad code for Blackfin ADSP-BF535 DSP. The core reaches asymptotically a theoretical minimum of 2.5 cycles per biquad and provides features that are not currently available in VisualDSP++ supporting DSP library functions.**

The functionality of the proposed implementation was fully tested in VisualDSP++ simulator and evaluated with ADSP-BF535EZ-Kit hardware.

Index terms – **IIR, biquad, library function, VisualDSP++, pipeline stalls**

1 Introduction

Architectures of all typical Digital Signal Processors (DSPs) have been optimized at least for standard signal processor algorithms – Finite Impulse Response (FIR), Infinite Impulse Response (IIR) and Fast Fourier Transform (FFT). It is well-known that practical implementation of IIR filtration on fixed-point DSPs requires proper decomposition of transfer function in order to overcome finite-precision problems [1].

A biquad realization of IIR filters is one of the best solutions for implementation of many practical IIR filters on typical DSPs. Older DSPs have quite simple data-paths with only 1 single-cycle Multiply and Accumulate (MAC) unit and typically use quite simple pipelining only in their control units (CUs). A general biquad uses 5 coefficients. The core of biquad implementation can reach a theoretical minimum of 5 cycles/biquad for some older DSP with one MAC unit (e.g. Motorola DSP5600x).

Modern fixed DSPs generally use more complicated data-paths and deeper pipelining in CU as well as in data-paths. Analog Devices Blackfin DSPs [2] are a typical example of modern fixed-point DSP with 2 MAC units. A theoretical performance of biquad IIR implementation is thus 2.5 cycle/biquad for 2 MAC Blackfin architecture. Currently available IIR routines in Blackfin VisualDSP++ development tool [3] do not reach this performance and have other serious limitations. It was not known if relatively complex fully interlocked Blackfin pipeline allows to reach the theoretical limit for IIR filter implementation. The paper presents the first known IIR biquad implementation that asymptotically reaches this theoretical limit in Blackfin ADSP-BF535 DSP.

2 Main Features of the Blackfin Architecture

The ADSP-BF5xx (Blackfin) [2] is a family of 16-bit fixed-point dual-MAC processors from Analog Devices. The ADSP-BF5xx combines features common for low-power DSPs with features traditionally associated with general-purpose RISC processors. The ADSP-BF5xx is based on Micro Signal Architecture (MSA) instruction set architecture jointly developed by Analog Devices and Intel. There are two generations of Blackfin processors which have slightly different instruction sets and microarchitectures [2]. Due to these architectural differences, the two generations are only partly assembly-code compatible. The first generation Blackfin processor, the ADSP-BF535, achieves a clock speed of 350 MHz at 1.6 volts. Second-generation Blackfin processors include the ADSP-BF531,2,3,4 ADSP-BF536,7 and ADSP-BF561. The second-generation parts operate at up to 750 MHz at 1.45 volts. Next subsections summarize features of Blackfin DSPs that are relevant to description of IIR implementation. Further information can be found in the corresponding manuals and application notes [4, 5].

2.1 Data Paths

The ADSP-BF5xx contains two fixed-point data paths, two address generation units, and a program sequencer. It also includes a data register file of eight 32-bit registers R0-7, as well as two 40-bit accumulators A0,1. The ADSP-BF5xx uses a RISC like load/store architecture: the data paths generally take inputs from and returns results to the data register file or the accumulators.

2.2 Memory Subsystem

Blackfin DSPs support a hierarchical memory model with different performance and size parameters, depending on the memory location within the hierarchy. The fastest memory, Level (L1), which

includes instruction, data, and scratchpad memory, is accessible in a single cycle. L2 memories, which include on-chip SRAM and off-chip mapping to synchronous, asynchronous, and PCI devices, provide much larger memory spaces, but with higher latencies.

The ADSP-BF5xx core can perform one instruction read and two data transfers in each cycle. Each data transfer can be 8, 16 or 32 bits wide. Both data transfers can access the same data bank if they use different sub-banks, but only one of the transfers can be a store. If data is arranged as 16-bit pairs in memory, the ADSP-BF5xx can transfer four 16-bit values in each cycle.

The ADSP-BF5xx address space is byte addressable. However, instructions must be aligned on 16-bit boundaries, and loads and stores must maintain the appropriate alignment for the transfer: 8-, 16-, and 32-bit transfers must be aligned on 8-, 16-, and 32-bit boundaries, respectively.

2.3 Control Unit

- Interlocked Pipeline

The first-generation ADSP-BF535 pipeline has 8 stages, while the second-generation ADSP-BF5xx pipeline has 10 stages. Both generations feature fully interlocked pipelines, so that data hazards do not cause unexpected results. However, some data hazards are resolved by stalling the processor. Data forwarding has been improved in the second-generation microarchitecture, some stalls that result from data hazards on ADSP-BF535 have been removed by the improved forwarding mechanisms. The ADSP-BF5xx uses static branch predicition for all conditional branches. Instructions generally execute in one cycle in the absence of memory access-related delays and pipeline stalls. The most significant exceptions are the jump, call, and most return instructions, which require four or more cycles.

- Load/Store Concept

The Blackfin DSP architecture supports a RISC-like concept of “load/store machine”. This refers to the characteristic in RISC architectures of intentionally separating memory operations (load and stores) from the arithmetic functions that use targets of memory operations. This prevents memory operations (particularly instructions that access off-chip memory or I/O devices and often take multiple cycles to complete) from stalling the processor thus allowing an instruction execution rate of one instruction per cycle.

The separation of load operations from their associated arithmetic functions allows compilers and assembly language programmers to place unrelated instructions between the load and its dependant instruction(s) [5]. These unrelated instructions execute in parallel while the processor waits for the memory system to return data. If a value arrives before the dependant operation reaches the execution stage of the pipeline, the operation completes in one cycle.

3 Available IIR filter functions for Blackfin DSP

VisualDSP++ is a standard integrated software development tool [3]. Key features include the native C/C++ compiler, C/C++ libraries, advanced graphical plotting tools, statistical profiling, and the VisualDSP++ Kernel. C/C++ DSP library [6] include many optimized block- processing functions. Currently available IIR functions included in the optimized DSP library:

- IIR filter based on the Direct Form II (DF II) biquads

It is available as a standard C-library function

*void iir_fr16(const fract16 x[], fract16 y[], int Ni, iir_state_fr16 *s);*

in all versions of VisualDSP++. The core of iir_fr16 function implements a standard biquad with the transfer function ($i = 1, 2, \dots B$)

$$H_i(z) = \frac{b_0^i + b_1^i z^{-1} + b_2^i z^{-2}}{1 + a_1^i z^{-1} + a_2^i z^{-2}} \quad (1)$$

as shown in Fig.1.

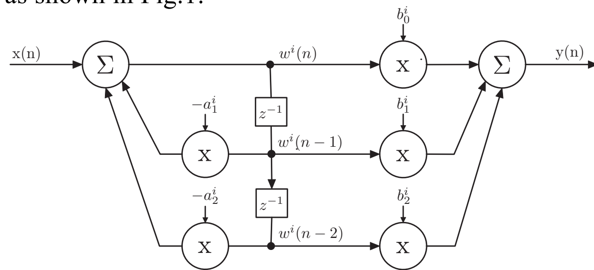

Fig.1 Direct form II biquad implementation

The *iir_fr16* function increases implementation efficiency by passing 2 input samples through biquad cascade in parallel. The core of the *iir_fr16* function requires [6]

$$\sim 3N_e B \quad (2)$$

cycles, where N_e is an even number of input samples and B is a number (can be even or odd) of biquad sections. It is clear that it does not reach the theoretical limit. Moreover, the *iir_fr16* function has a serious practical limitation. It supports only biquad coefficients a_1^i, a_2^i, b_0^i, b_1^i, b_2^i from the fractional interval $\langle -1, 1)$. It is well-known that stable biquads can have $a_1^i \in (-2, 2)$ [1] and thus it cannot be used for implementation of many practical IIR filters.

- IIR filter based on Direct Form I (DF I)

The latest version of C/C++ DSP library (VisualDSP++ v.4.0, February 2005) contains also Direct Form I (DF I) based IIR function

*iirdf1_fr16(const fract16 x[], fract16 y[], int n, iirdf1_state_fr16 *s);)*

that implements the transfer function

$$H_{IIR}(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_N z^{-N}} \quad (3)$$

by using scaled coefficients $A_1 = a_1 / S$, …, $B_N = b_N / S$ as shown in Fig.2.

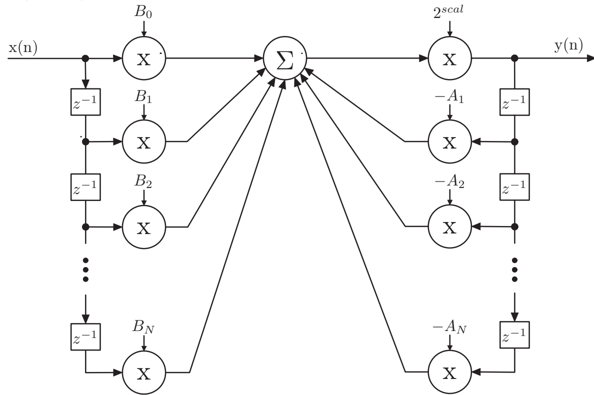

Fig.2 Direct form I IIR filter implementation

The proper scaling factor $S = 2^{scal}$ ensures that none of stored coefficients is outside the fractional $\langle -1,1 \rangle$ interval. The core of the *iirdf1_fr16* function requires [6]

$$\sim N_e N = 2 N_e B \qquad (4)$$

cycles, where N_e is an even number of input samples and $N = 2B$. Probably the main motivation, why this function has been included into the latest DSP library was an attempt to compensate limitations of biquad based *iir_fr16* function. Although from (4) it seems that *iirdf1_fr16* is very efficient, it cannot be used for many practical IIR filters. It is well known that finite word-length effects of 16-bit fixed-point arithmetic cause instability of IIR based on DF I [1].

4 Optimized IIR function

The proposed optimized IIR function *iir_fr16_v2* is based on a non-canoical implementation of biquad using scaled coefficients $A_1^i = a_1^i / 2$, $A_2^i = a_2^i / 2$, $\ldots B_2^i = b_2^i / 2$ shown in Fig.3. It implements the transfer function (1) for $i = 1,2,\ldots,B_e$ by using fixed scaling $S_i = 2^{scal_i} = 2$ in each biquad.

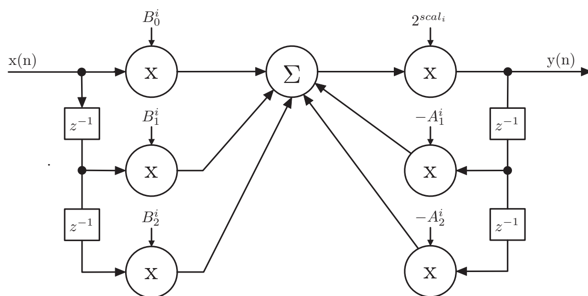

Fig.3 Optimized non-canonic biquad implementation

The core of the *iirdf1_fr16_v2* function requires

$$\sim 2.5 N_e B_e \qquad (5)$$

cycles, where N_e is an even number of input samples and B_e is an even number of biquads. The efficiency of the proposed implementation is based on:

- **Parallel passing** of 2 input samples through biquad cascade.
- **Loop unrolling** of inner IIR filer loop that contains two (referred as odd and even) biquads.
- Efficient use of **special scaling mode** (S2RND) for implementation of post-multiplication by factor $S_i = 2$ in each biquad. Although this feature is quite commonly used with other DSPs (e.g. Motorola DSP56xxx) it was not used in Analog Devices fixed-point DSP IIR library functions.
- **Extensive hand optimization** of inner IIR filter loop code. Optimization had to resolve several pipeline stalls that occur in relatively complex BF-535 pipeline [5]. The final code of optimized inner loop is shown in Tab.1 [7] (understanding of this code requires a very good knowledge of Blackfin DSP architecture and its instruction set). It requires exactly 10 cycles per 2 samples and 2 biquad sections. This is **the theoretical minimum for 2-MAC DSP architecture**.

The code has the following features:

- It is optimized for **an even number** of biquads.
- The code must be **properly aligned** (by .align8 directive) on the 8-byte code memory position.
- Code and data must be located in the L1 code and both data memories in order to reach the optimum performance.
- A short initialization code (step 0 in Tab.1) must be used before the inner loop (steps 1-10).
- It can implement all stable IIR filters.
- Final code is provided as C library function *iir_fr16_v2* with the same prototype as *iir_fr16*.

5 Experimental results

The proposed optimized implementation was fully tested with available software and hardware development tools. An influence of 16-bit fixed point arithmetic was tested in VisualDSP++ simulator. Fig.4 shows difference of *iir_fr16_v2* computed in VisualDSP++ simulator and reference Matlab implementation for 8-th order elliptic band-pass filter with $f_s = 48\,\text{kHz}$, $f_{p1} = 6\,\text{kHz}$, $f_{p1} = 9\,\text{kHz}$, $f_{s1} = 5.3\,\text{kHz}$, $f_{s1} = 9.7\,\text{kHz}$, $\sigma_p = 0.1\,\text{dB}$, $\sigma_s = 80\,\text{dB}$ and $quant = 1.15$ designed by Systolix filter design package [7]. As an input signal a uniformly distributed full scale white random signal was used. Note that the tested filter cannot be implemented by available *iir_fr16* function (many feedback coefficients are outside the interval $\langle -1,1 \rangle$) or *iirdf1_fr16* (instability of DF I structure for 16-bit arithmetic).

It was found out that VisualDSP++ simulator does not provide reliable results for all code alignments in the code memory. For this reason the number of cycles was confirmed by direct measurements on BF-535 EZ-Kit hardware by using internal Blackfin hardware cycle counters. These measurements confirmed validity of (5) for optimized *iirdf1_fr16_v2* internal loop code.

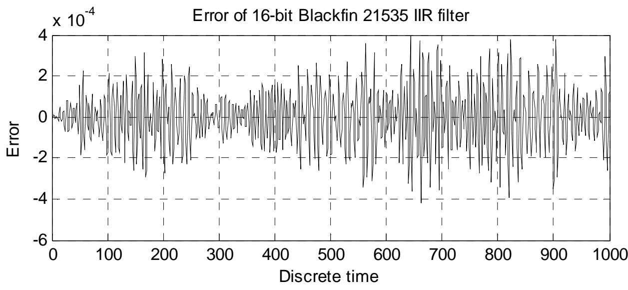

Fig.4 Precision of *iir_fr16_v2* implementation

5 Conclusions

The proposed new biquad IIR filter implementation for Blackfin ADSP BF-535 DSP significantly improves currently available library functions for IIR filtration. It solves all practical aspects of IIR implementation on Blackfin DSP and asymptotically reaches the theoretical limit of dual-MAC Blackfin DSP architecture. It is currently the fastest known IIR implementation for Blackfin BF-535 core and demonstrates that also relatively complex interlocked pipelined Blackfin architecture can execute IIR filter without pipeline stalls.

Acknowledgment: *The author wish to thank Analog Devices, Inc. for providing software and hardware development tools used within this project.*

References

[1] B. Porat, *A Course in Digital Signal Processing*, John Wiley & Sons, New York, 1997.

[2] "*A BDTI Analysis of the Analog Devices ADSP-BF5xx*", Berkeley Design Technology, Inc., pp.1-5, www.analog.com/processors/processors/blackfin/pdf/blackfin_summary.pdf.

[3] "*VisualDSP++*", www.analog.com.

[4] www.analog.com/processors/processors/blackfin/.

[5] "*ADSP-BF535 Blackfin Processor Multi-cycle Instructions and Latencies*", App. Note EE-171, May 13, 2003, pp.1-15, www.analog.com.

[6] "*VisualDSP++ 4.0 C/C++ Compiler and Library Manual for Blackfin Processors*", Revision 3.0, January 2005.

[7] M. Kráľ and M. Nemčík, "*Optimization of IIR filtration on ADSP21535*", Semestral project, KEMT TU Košice, June 2004, pp.1-25, (in Slovak).

[8] "*Filter Express v.5.1*", www.systolix.com.

Table1 Inner loop of optimized *iir_fr16_v2* code

step	accum. A0	accum. A1	register contents		pointer position
initialization code					
0.		b_2*x_{-2} $+b_1*x_{-1}$	**R0:** b_1 **R1:** b_2 $-a_2$ **R2:** x_1 x_0 **R3:** x_{-1} y_{-2}	**R4:** **R5:** **R6:** **R7:**	I1 → x_{-1} I2 → b_0
	r0.l=w[i2++]; r1.l=w[i2++] || r2.h=w[i0++]; r1.h=w[i2++] || r2.l=w[i0++]; r3=[i1++]; a1=r1.l*r3.h || i1+=2; a1+=r0.l*r3.l || r3.h=w[i1--]; i1-=4;				
code for implementation of odd biquads					
1.	b_2*x_{-1}	$-a_2*y_{-2}$	**R0:** b_1 b_0 **R1:** b_2 $-a_2$ **R2:** $[x_1\ x_0]_{->x-1\ x-2}$ **R3:** x_{-1} y_{-2}	**R4:** **R5:** **R6:** **R7:**	incr. I2 by 2 bytes incr. I1 by 4 bytes
	a1+=r1.h*r3.h, a0=r1.l*r3.l || r0.h=w[i2++] || [i1++]=r2;				
2.	$+b_1*x_0$	$+b_0*x_0$	**R0:** b_1 b_0 **R1:** $-a_1$ $-a_2$ **R2:** x_1 y_{-1} **R3:** x_{-1} y_{-2}	**R4:** **R5:** **R6:** **R7:**	incr. I2 by 2 bytes incr. I1 by 2 bytes
	a1+=r0.h*r2.h, a0+=r0.l*r2.h || r1.l=w[i2++] || r2.h=w[i1++];				
3.	$-a_2*y_{-1}$	$[-a_1*y_{-1}]_{s2rnd}$	**R0:** b_{1n} b_0 **R1:** $-a_1$ $-a_2$ **R2:** x_1 y_{-1} **R3:** x_{-1} y_{-2}	**R4:** y_{-2} $[y_0]$ **R5:** **R6:** **R7:**	incr. I2 by 2 bytes incr. I1 by 2 bytes
	r4.h=(a1+=r1.l*r2.h),a0+=r1.h*r2.h (s2rnd) ||r0.l=w[i2++]||r4.l=w[i1++];				
4.	$+b_0*x_1$	$b_{1n}*x_{-1n}$	**R0:** b_{1n} b_0 **R1:** $-a_1$ b_{2n} **R2:** x_1 y_{-1} **R3:** y_{-1n} y_{-2n}	**R4:** y_{-2n} y_0 **R5:** **R6:** **R7:**	decr. I1 by 4 bytes incr. I2 by 2 bytes
	a1=r0.l*r2.h, a0+=r0.h*r2.l || r1.h=w[i2++] || r3=[i1--];				
5.	$[-a_1*y_0]_{s2rnd}$	$+b_{2n}*x_{-2n}$	**R0:** b_{1n} b_0 **R1:** $-a_{2n}$ b_{2n} **R2:** x_1 y_{-1} **R3:** x_{-1n} y_{-2n}	**R4:** $[y_1]$ y_0 **R5:** **R6:** **R7:**	incr. I2 by 2 bytes I1 → x_{-1n} I2 → b_{0n}
	a1+=r1.h*r4.l, r4.l=(a0+=r1.l*r4.h) (s2rnd) || r1.l=w[i2++] || r3.l=w[i1];				
code for implementation of even biquads					
6.	b_2*x_{-1}	$-a_2*y_{-2}$	**R0:** b_1 b_0 **R1:** $-a_2$ b_2 **R2:** **R3:** x_{-1} y_{-2}	**R4:** $[x_1\ x_0]_{->x-1\ x-2}$ **R5:** **R6:** **R7:**	incr. I2 by 2 bytes incr. I1 by 4 bytes
	a1+=r1.l*r3.h, a0=r1.h*r3.l || r0.h=w[i2++] || [i1++]=r4;				
7.	$+b_1*x_0$	$+b_0*x_0$	**R0:** b_1 b_0 **R1:** $-a_2$ $-a_1$ **R2:** **R3:** x_{-1} y_{-2}	**R4:** x_1 y_{-1} **R5:** **R6:** **R7:**	incr. I2 by 2 bytes incr. I1 by 2 byets
	a1+=r0.h*r4.h, a0+=r0.l*r4.h || r1.h=w[i2++] || r4.h=w[i1++];				
8.	$-a_2*y_{-1}$	$[-a_1*y_{-1}]_{s2rnd}$	**R0:** b_{1n} b_0 **R1:** $-a_2$ $-a_1$ **R2:** y_{-2} $[y_0]$ **R3:** x_{-1} y_{-2}	**R4:** x_1 y_{-1} **R5:** **R6:** **R7:**	incr. I2 by 2 bytes incr. I1 by 2 bytes-
	r2.h=(a1+=r1.h*r4.h),a0+=r1.l*r4.h (s2rnd) ||r0.l=w[i2++]||r2.l=w[i1++];				
9.	$+b_0*x_1$	$b_{1n}*x_{-1n}$	**R0:** b_{1n} b_0 **R1:** b_{2n} $-a_1$ **R2:** y_{-2} y_0 **R3:** y_{-1n} y_{-2n}	**R4:** **R5:** **R6:** **R7:**	decr. I1 by 4 bytes incr. I2 by 2 bytes
	a1=r0.l*r4.h, a0+=r0.h*r4.l || r1.l=w[i2++] || r3=[i1--];				
10.	$[-a_1*y_0]_{s2rnd}$	$+b_{2n}*x_{-2n}$	**R0:** b_{1n} b_0 **R1:** b_{2n} $-a_{2n}$ **R2:** $[y_1]$ y_0 **R3:** x_{-1n} y_{-2n}	**R4:** **R5:** **R6:** **R7:**	incr. I2 by 2 bytes I1 → x_{-1nn} I2 → b_{0nn}
	a1+=r1.l*r2.l, r2.l=(a0+=r1.h*r2.h) (s2rnd) || r1.h=w[i2++] || r3.l=w[i1];				