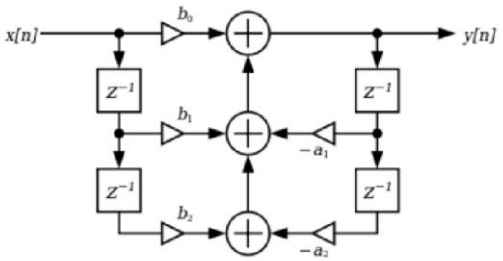
ULTRA-FAST BIQUAD FILTERING OPTIMIZED FOR CORTEX-M4/M7



firmware-developments.com
The developer's place

Overview

This program implements the IIR/BIQUAD subroutine as described at https://www.keil.com/pack/doc/CMSIS/DSP/html/group___biquad_cascade_d_f1.html

Subroutine name "**arm_biquad_cascade_df1_fast_q15**". The initialization subroutine "arm_biquad_cascade_df1_init_q15 " is identical to CMSIS.

The code delivery consists in three folders:

- "DOC" : this documentation
- "KEIL" : uVision V5.12 project used as test-bench of the subroutine and CMSIS's
- "C_BIQ_M4" : bit-exact arithmetic C-code simulator in VisualC2010
- "encrypted_file" : source code tar'd and coded with a key we send by email.

```
$ ls -l
total 84
drwxrwx---+ 1 FirmwareDevelopments None     0 16 mai   21:23 C_BIQ_M4
drwxrwx---+ 1 FirmwareDevelopments None     0 16 mai   21:23 DOC
drwxrwx---+ 1 FirmwareDevelopments None     0 16 mai   21:23 encrypted_file
drwxrwx---+ 1 FirmwareDevelopments None     0 16 mai   21:23 KEIL
-rwxrwx---+ 1 FirmwareDevelopments None 74715 16 mai   17:42 p_biquad_m4.jpg
```

The folders are located at http://firmware-developments.com/WEB/P6x/BIQ_M4/

Details

The code reuses the same data structures, data format and APIs of the original CMSIS library.

When compiled with option -O3 for Cortex-M4 this program runs **16% faster** than the original CMSIS library (KEIL ARM C-compiler V5.05).

The new API name is "arm_biquad_cascade_df1_fast_q15_fwd". It assumes the even-order samples are aligned on **four-bytes boundaries**. There must be an **even number of samples** to process (the constraint can be relaxed on demand).

The code is written in two files:
- one in C holding the loop on BiQuads (`arm_biquad_cascade_df1_fast_q15_fwd`)
- one in assembly processing one BiQuad (`arm_biquad_cascade_df1_fast_q15_fwd_asm`) .

Here is the extract of the compilation MAP file for the code size:

```
arm_biquad_cascade_df1_init_q15          0x0000015d   Thumb Code    26  arm_biquad_cascade_df1_fast_q15.o(.text)
arm_biquad_cascade_df1_fast_q15          0x00000177   Thumb Code   204  arm_biquad_cascade_df1_fast_q15.o(.text)
arm_biquad_cascade_df1_fast_q15_fwd      0x00000243   Thumb Code    94  arm_biquad_cascade_df1_fast_q15.o(.text)
arm_biquad_cascade_df1_fast_q15_fwd_asm  0x00000349   Thumb Code   168  arm_biquad_cascade_df1_fast_q15_fwd_asm.o(.text)
```

CPU load

The code speed is benchmarked on the KEIL simulator assuming 0 wait-state.

We advertised 7.5 cycles per sample with the following computations.

The critical loop takes **17 cycles per 16bits sample pairs** (15 cycles on Cortex-M7 where the Load/Store unit can execute in parallel with ALU), on top of which you add **1 cycles for loop-counter** increment and **3 cycles for the conditional loop branch**.

The code complexity is then **21 cycles per sample pairs without unrolling the loop**.

When a loop of 1000 samples was processed through 2 BiQuads with the original CMSIS code in 25.1kcycles this subroutine takes 21.2kcycles with the same saturation control

The subroutine uses 36 bytes of stack more than the original one. The stack usage goes then from about 136 bytes to 172bytes. This can be substantially optimized on request.

API

Documentation extracted from https://www.keil.com/pack/doc/CMSIS/DSP/html/group___biquad_cascade_d_f1.html

void arm_biquad_cascade_df1_fast_q15_fwd (

const arm_biquad_casd_df1_inst_q15 * S,
q15_t * pSrc, q15_t * pDst,
uint32_t blockSize

Parameters

[in]	*S	points to an instance of the Q15 Biquad cascade structure.
[in]	*pSrc	points to the block of input data.
[out]	*pDst	points to the block of output data.
[in]	blockSize	number of samples to process per call.

Returns

none.

Scaling and Overflow Behavior:

This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around and distorts the result. In order to avoid overflows completely the input signal must be scaled down by two bits and lie in the range [-0.25 +0.25). The 2.30 accumulator is then shifted by postShift bits and the result truncated to 1.15 format by discarding the low 16 bits.