DSP functions in DAC8PRO

Summary:
The digital board of DAC8PRO is built around an XMOS XU216 device which has extra cores and powerful instructions to perform DSP functions like filtering, cross over or delay. By using the AVDSP framework and its utilities, it is possible to create a DSP program to be uploaded in the DAC8PRO. This document explains how this works and provides step by step examples.

DAC8PRO DSP firmwares:
By default, DAC8PRO runs with two firmwares. One for XU216 in charge of USB interface and data transmission with DAC and AES inputs. One for managing the front-panel (volume, display…) and adjusting DAC registers according to the USB host sample rate.
In order to enable DSP functionalities, a specific version of the XU216 firmware must be installed with the DFU upgrade utility. Then it will be possible to manually upload DSP programs that will interact with internal data flows existing between DAC, AES and USB host in and out.
The DSP enabled DAC8PRO firmware is fully compatible with the latest standard DAC8PRO firmware and can also run without any DSP program. It does not include any front-panel firmware, so when installing it, the front-panel firmware will be kept as it is. Minimum version V1.6 is required to select a DSP program in the DAC **Filter** menu.

Firmware version and capabilities:
The maximum sample rate is configured at 96khz instead of 192k. If the target DSP program takes too long to accommodate the time between two audio samples (10us at 96k), then the DAC is dynamically reconfigured for half the sample rate and a decimation is done with a preconditioning FIR filter. To avoid THD, it is always better to send the audio at a sample rate compatible with (or below) the maximum rate of the DSP program.
Two versions are provided. When 4 AES inputs are required, the power available for the DSP program is limited to 250 MIPS spread over 4 cores max. When a single AES input is acceptable, the power available for the DSP program is 437 MIPS spread over 7 cores max.

Summary of firmwares capabilities:

Name (USB device and file name)	USB OUT	USB IN	DAC	AES	CORES x mips	MIPS (total)	total instructions 48k	total instructions 96k
DAC8PRODSP4	**8**	**8**	**8**	**8**	**1 x 100 2 x 83 3 x 71 4 x 62**	**100 166 214 250**	**5208**	**2600**
DAC8PRODSP7	**8**	**8**	**8**	**2**	**1..4 x 100 5 x 83 6 x 71 7 x 62**	**400 416 428 437**	**9113**	**4552**

A specific 192000 version could be proposed on demand for integrators, depending on requirement and feasibility.
Remark : the DAC8PRO DSP firmware also accepts 44100 or 88200 hz.
DSD playback is not possible (silence) when a DSP program is selected.

Creating a DSP program:
At the time of writing, no graphical user interface is provided to create a DSP flow.
It will require the following steps:

1. defining and organizing the actions expected as a flow of macro instructions that will be executed at each audio sample
2. evaluating cpu load for this flow and grouping / distributing macro-instructions per core
3. using a text editor to write this program in sequences with parameters and code sections
4. converting this file as a binary file with “dspcreate” utility from AVDSP framework
5. uploading the resulting binary file with “xmosusb” command line utility.
6. verifying DAC8PRO status and XMOS load for each core using “xmosusb” utility
7. testing the expected behavior with third party tools like REW using loopback instructions to verify for each channel response, total gains, and verifying clipping behaviour.

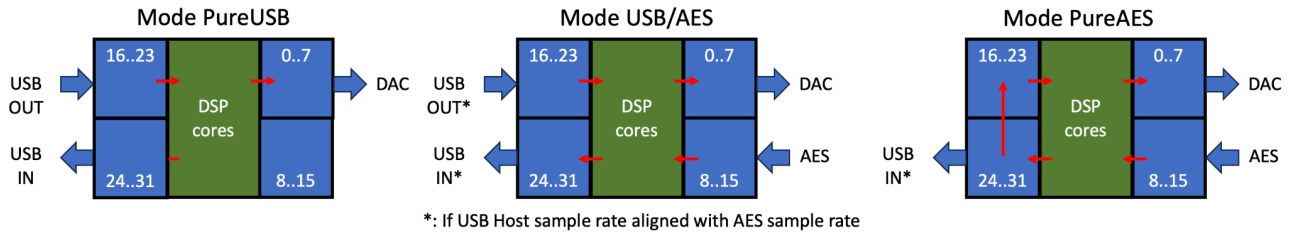
This looks like an intensive effort at first but this gives the possibility to optimize the treatment and to fit a comfortable crossover solution within a limited MIPS quantity. Routing and mixing is also very flexible and efficient as there is no predefined matrix or framework to follow.

The generic behavior of a dsp flow is to : load a sample, treat it, eventually delay it, store it.
A sample can be loaded in the DSP accumulator from any AES input or from any channel provided by the USB host. 16 memory locations are predefined for this.
Transformation can be applied to the DSP accumulator with a set of predefined macro instructions like gain or biquad or saturation.
Then the accumulator can go through a fifo pipeline for applying delays (in microseconds).
The resulting accumulator will be stored as an output sample either for the DACs or for the USB host, represented by 16 other predefined memory locations.
Any core can work with any of these 32 (16 input + 16 outputs) predefined memory locations.

Additional memory locations can be created to share value across cpu cores. For example a core can prepare a stereo input signal with gain and peak correction and save the result in a shared memory location for the other cores to act as a crossover for 8 channels.

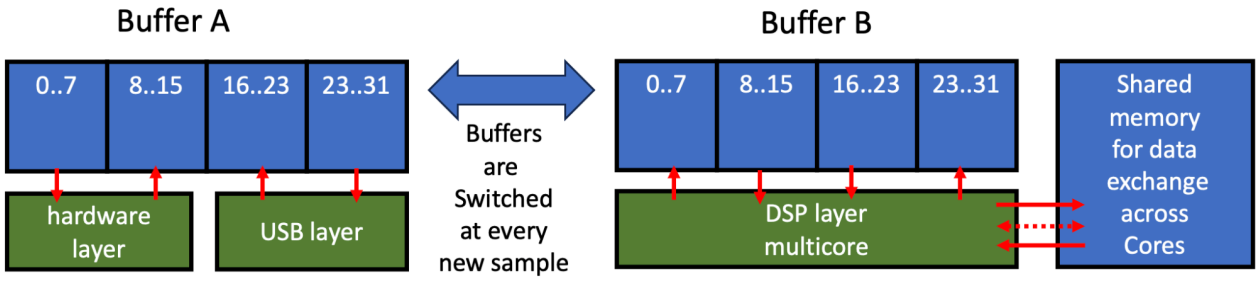
In order to interface the DSP layer with the USB-DAC-AES layer through these 32 memory locations, a mechanism of double buffering is implemented. This guarantees perfect timing across channels and cores (at the expense of 2 samples delay).
The hardware layer interacts only with buffer A, and the DSP layer interacts only with buffer B. and buffer A and B are switched at every sample.

The following diagram represents the 32 memory location according to the 3 DAC modes:



*: If USB Host sample rate aligned with AES sample rate

the following diagram represents the buffer management:



MIPS needed for a typical project

A table in this document represents the list of macro instructions available and the time taken to execute them in native cpu core instructions, which is to be compared directly with MIPS available per core or in total.

As an example, a typical flow for filtering and delaying one channel needs 4 macro instructions, and requires 176 core instructions for 6 biquad sections (e.g. Linkwitz-Riley 4th order followed by 4 peak corrections).

The table below summarizes the cpu instructions available per core depending on the sample rate and depending on the MIPS allocated to a core (which depends on the total number of active cores). An indication is given for a maximum number of typical treatments as above.

MIPS per core	48000hz		96000hz		192000hz	
100,0	2083	11	1042	5	521	2
83,3	1736	9	868	4	434	2
71,4	1488	8	744	4	372	2
62,5	1302	7	651	3	326	1

Remark : To maximize the total MIPS capacity, it is recommended to use multiple cores if possible when the program can be split in similar sections (like a 2x3 crossover solution).

DSP program:

A DSP program built with the AVDSP framework is a list of sequential binary values (called “opcode”) that will be interpreted one-by-one by a specific runtime library optimized for XMOS XU216 and included within the DAC8PRO DSP firmware.

The opcodes are low-level DSP-like instructions (about 60), mainly working on an accumulator or transferring data across memory.

The binary file also contains constant definitions like filters parameters or mixer/gain/delay, which are all computed upfront to cope with all possible audio sample rate (no any SRC involved).

The DAC8PRO front-panel does not provide any mechanisms to modify the filters or gain dynamically during audio listening. All values are embedded in the DSP program binary file and this can be changed only by modifying and uploading a new version of the DSP program.

In order to generate a DSP binary file, a specific command-line utility called `dspcreate` is provided. It accepts either inputs:

A. A text file representing the dsp flow with a predefined set of simple macro-instruction, or
B. An object file generated by a C compiler like GCC using the core AVDSP library, enabling full capabilities and extended syntax.

Both methods are compatible with the DAC8PRO DSP firmware but for simplifying the process, this document describes and focuses on using a text file with a limited set of predefined macro-instructions as an input.
Analyzing a text file and generating a DSP binary program is done with the following command line:

```
OS prompt > dspcreate -dsptext myprogram.txt -binfile
mybinfile.bin <param>
```

where `<param>` is optional and describes a list of couples Label=value which can be used as preprocessing information for the DSP program. For example a cutoff frequency can be passed as `Fc=800` on the command line and will be treated before analyzing `myprogram.txt`.

As explained with more details in next chapters, the binary file generated will be uploaded in the DAC8PRO RAM memory with the command-line utility “xmosusb” with the following command:

```
OS prompt > xmosusb --dspload mybinfile.bin
```

the status of the DAC after this upload will be given by the command:

```
OS prompt > xmosusb --dacstatus
```

providing the following informations as an example:

maximum dsp tasks = 4
dsp 1: instructions = 131
dsp 2: instructions = 66
dsp 3: instructions = 66
dsp 4: instructions = 0
maxi instructions = 131 / 520 = 25%fs

DSP program structure:
The text file passed as a parameter with the selector `-dsptext` has to be structured with following sections:

- label definitions for documenting constant parameters (not stored in the final binary file)
- label definitions for filters, or memory parameters which are stored in the binary file
- code sections with macro-instruction for each core.

This structure can be repeated as much as needed according to the maximum number of cores available. The 2 first optional sections can be grouped at the beginning of the file or spread along if they are specific to a core.

Label value definition:
A DSP program is easier to write and verify when defining labels which represent physical locations or constants to be used in different points of the code section. As an example:

```
LeftIn       16 ; RightIn 17 ; LeftOut 10 ; RightOut 13 ### USB and DAC IOs
LeftLowOut   =LeftOut ; LeftMidOut =LeftOut+1 ; LeftHighOut =LeftOut+2
RightLowOut  =RightOut[0] ; RightMidOut =RightOut[1] ; RightHighOut
=RightOut[2]
LeftHeadphone 8 ; RightHeadphone  9
firstGain    -3db
Low_fc       400 ; Mid_fc 2000
```

As seen in this example, a label can be defined directly with a numerical value (eg `LeftIn 16` or `firstGain -3db`), or with an expression (+,-,*,/) starting with “=”. Multiple definitions can be done on a line with “;” separator. Comments are possible with “#” preceding character.

Remark: when an expression starts with a decibel value, then operators are limited to + and -, and next values in the expression are expected to be also decibel.
Numbers can be written in decimal, binary starting with a letter “b” or hexadecimal with “x”.

Label parameter definition:
There are 2 types of static parameters :filters, or memories.

Filters describe a list of first and/or second order filters to be computed in a row with a biquad routine. The biquad coefficients are computed upfront according to a list of predefined filter names covering most of the usual requirements (Bessel, Butterworth, Linkwitz-Riley, All-pass, peak, notch…). Generic filters can also be combined to form other filters. as an example:

```
LowPass      LPLR4(Low_fc)
MidPass      HP2(Low_fc, 0.7) HP2(Low_fc, 0.7) LPBU3(Mid_fc)
HighPass     HPBU3(Mid_fc, -2db) HS2(5000, 0.5, +2db)
RoomMode     NOTCH(50, 10) PEAK(80, 0.5, -2db)
```

This represents the settings for a typical 3 way crossover:

- `LowPass` is simply a Linkwitz-Rilley 4th order low pass filter with a cutoff frequency of “Low_fc” (at -6db by design)
- `MidPass` is also a Linkwitz-Rilley 4th order high pass filter but made of 2 successive 2nd order high-pass filters (by definition of an HPLR4 using two HP2 with Q=0.7), followed by a low pass Butterworth 3rd order filter with cutoff frequency Mid_fc (at -3db by design).
- `HighPass` is an high-pass Butterworth 3rd order with an embedded attenuation of -2db after it, followed by a second order high-shelf filter (with Q=0.5 here) with a gain of+2B after 5000hz.
- `RoomMode` defines a notch filter at 50hz with a strong Q=10, and a -2db reduction with a peak correction at 80hz with a relatively large Q=0.5.

These 4 filter labels can be used later in the core code section as parameters of the “biquad” macro instruction. Grouping them at the beginning of the program is easier to maintain.
Remark: any filter can be set with an optional gain (or reduction) parameter. Then its biquad coefficients (b0,b1,b2) will be scaled accordingly by the encoder.

Memory describes a target memory location which can be used to exchange data between cores. As an example, a core can be used for pre-conditioning a stereo signal and then the other core will use these values instead of directly using the 32 Input/output memory locations.

```
stereoMem    MEMORY2
monoChan     MEMORY
```

This example defines 2 memory locations to store 2 channels related to a stereo signal and a single memory location to store a mono channel.

Code and core section:
a DSP code section starts with the keyword `core`. All the macro instructions are written one by one on the following lines. multiple instructions can be written on a single line with a “;” separator if this brings better visibility. This is an example for treating the Low channels as per the example parameters above:

```
core
      inputgain    (LeftIn, firstGain)
      biquad       LowPass
      output       LeftLowOut
      inputgain    (RightIn, firstGain)
      biquad       LowPass
      output       RightLowOut
end
```

The same code could be duplicated for treating the Mid channels, inside this core or in another core code section. The keyword `end` must be unique in the program. End of a core section is implicit when a new core instruction is encountered.

This program below provides an example with one preconditioning core in charge of stereo and headphones, and one other core in charge of the Low filtering. This can be merged or duplicated for mid and high filtering.

```
core
      mixer        (LeftIn, -6db) (RightIn, -6db)
      savemem      monoChan
      inputgain    (LeftIn, firstGain)
      biquad       RoomMode
      savexmem     stereoMem[ 0 ]
      inputgain    (RightIn, firstGain)
      biquad       RoomMode
      savexmem     stereoMem[ 1 ]
      transfer     (LeftIn, LeftHeadphone) (RightIn,RightHeadphone)
core
      loadxmem     stereoMem[ 0 ]
      biquad       LowPass
      output       LeftLowOut
      loadxmem     stereoMem[ 1 ]
      biquad       LowPass
      output       RightLowOut
end
```

By default, any `output` or `outputgain` instruction will clip the signal between (+1..-1) if the value of the accumulator is outside these bounds. Still this behavior is not satisfying as it generates a high level of harmonics. Therefore it is highly recommended to tune the gain chain to avoid any clipping situation. See also possibility to use saturate instruction as explained later in the document.
Remark : the DAC volume is independent of the DSP and acts directly on the DAC outputs.

Instruction details:

keyword	description	parameters
core	Define the beginning of a new code section for a new DSP core. A core can be active or not depending on a flexible 32 bits condition checked with 1 or 2 parameters. See detailed information later in this document.	optional 1 or 2 parameters representing core condition. First parameter represents the bits set to 1 compatible with this core. The second represents the bit set to 0 compatible with this core.

end	mark the end of the whole DSP program.	no parameter
transfer	load a sample from a memory location and save it to another.	(in, out) …
input	Load a sample from a memory location directly in the main DSP accumulator, without applying conversion or gain.	immediate numerical value or expression representing one of the 32 locations.
output	Save the main DSP accumulator in a memory location directly. Value is always clamped between -1 .. +1	immediate numerical value or expression representing one of the 32 locations.
inputgain	Load a sample from a memory location in the main DSP accumulator and apply a gain or reduction. When multiple couples (in,gain) are provided, they are added all together.	(in, gain) … 0<= in <32 -8 < gain < +8 -18db < gain < +18db
mixer	Execute an equivalent but faster version of multiple inputgain instructions. same parameters as inputgain.	(in, gain) …
loadxmem **loadymem**	Load the DSP accumulator X or Y with a value from a given memory location. The unique parameter is a label defined as MEMORY, eventually followed by an index number	memory label name with an optional [value] which will be added (as an index of an array)
savexmem **saveymem**	Store the value of the DSP accumulator X or Y in a given memory location. The unique parameter is a label defined as MEMORY	same as loadxmem
delayus **delaydpus**	Propagate the DSP accumulator msb through a FIFO data line in order to create a delay line. The parameter represents the number of microseconds. delaydpus perform the same operation on 64bits original data and thus requires twice memory.	delay in microseconds as a numerical value or expression
delayone	Switch the current DSP accumulator with its value at the previous sample, thus creating a delay of one sample. Used to synchronize outputs across cores when savexmem or loadxmem are used.	no parameter
dcblock	Provides a high pass first order to eliminate any continuous DC signal. The unique parameter is a minimum frequency cutoff. To optimize performance, it is better to include a HP1 filter at the beginning of a biquad filter.	frequency as a value or expression 10 < f < 100
saturate	Eventually modify DSP accumulator so that the value is constrained between -1..+1 . If a saturation is detected, a gain reduction is applied by steps of -6db.	no parameter
saturategain	Combined instruction, applying a gain or reduction to the DSP accumulator and	gain as an immediate numerical value or as an

	saturating the result immediately (as done with saturate instruction described above)	expression.
biquad	Compute multiple biquad sections based on the filter defined by a label and given as a parameter	filter label name
param	Restart a label parameter section, enabling definition of labels memory or filters below.	no parameter
sine	Generate a precise sine wave. Using MCFO algorithm. Remark the thd is almost perfect but amplitude will reduce when frequency increase close to fs/8	frequency as a numerical value or expression, followed with amplitude
dirac	Generate a one sample pulse. Typically used to test biquad response.	frequency as a numerical value or expression, followed with amplitude
square	Generate square waves.	frequency as a numerical value or expression, followed with amplitude

typical cpu requirements for dsp macro instructions:

avdsp macro instructions	**cpu instruction**
initialization and core start	21 + 5 when saturate flag is raised
basic input or output	10
basic transfer between one input-output	6 + 6 per transferred in-out
load sample and apply a gain (inputgain)	14
load multiple sample and apply specific gains (mixer)	14 + 6 for each (input,gain)
biquad filtering	40 + 19 for each filter (first or second order)
verify accumulator saturation and eventually apply -6db reduction steps	12 + 2 in case of reduction
same with a gain applied before (saturategain)	21 + 2 in case of reduction
fifo pipeline to delay a signal (delayus or delaydpus)	20 for delayus 23 for delaydpus
share data across core through memory (savexmem or loadxmem)	10
fifo pipeline for delaying by one single sample to synchronize core access to memory (delayone)	13

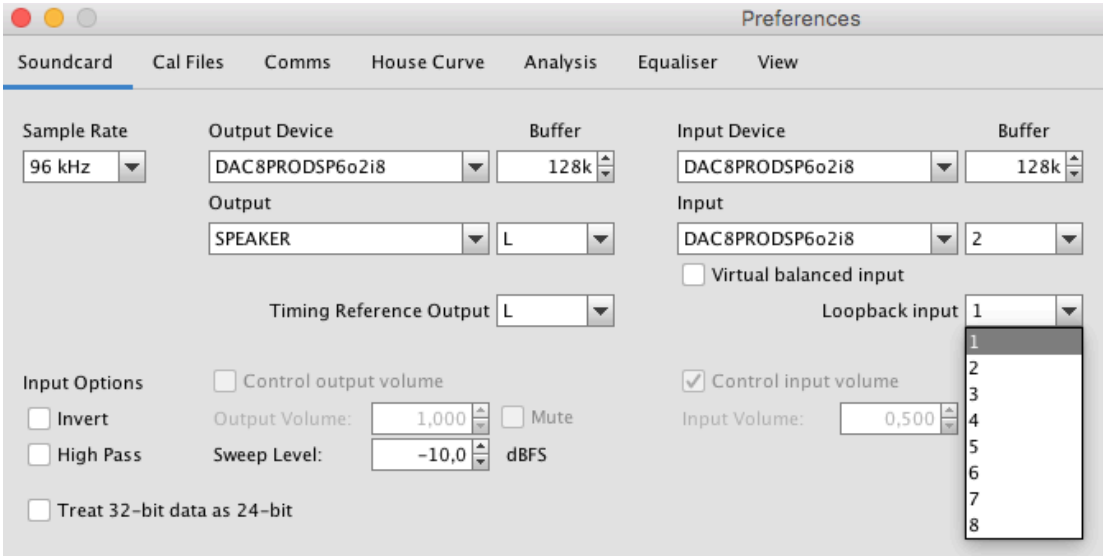
typical basic program for 1 channel: input x biquad 4 sections (eg LR4 + 2 x PEAK) delayus y output z	**138** or **214** for **8** sections

List of supported filter names:

type	**label name** (LP = low pass, HP = highpass)	**parameters**
Bessel	LPBE2, LPBE3, LPBE4, LPBE6, LPBE8 HPBE2, HPBE3, HPBE4, HPBE6, HPBE8	Frequency and optional gain
Bessel computed for fc cutoff at -3db	LPBE3db2, LPBE3db3, LPBE3db4, LPBE3db6, LPBE3db8, HPBE3db2 HPBE3db3, HPBE3db4, HPBE3db6, HPBE3db8	
Butterworth	LPBU2, LPBU3, LPBU4, LPBU6, LPBU8, HPBU2, HPBU3, HPBU4, HPBU6, HPBU8	
Linkwitz-Riley	LPLR2, LPLR3, LPLR4, LPLR6, LPLR8 HPLR2, HPLR3, HPLR4, HPLR6, HPLR8,	
classic first order	FLP1, FHP1, FAP1 (allpass)	Frequency and optional gain
High and Low shelf first order	FLS1, FHS1	Frequency and gain
High and Low shelf second order	FLS2, FHS2	Frequency and Q and gain
classic 2nd order (can be cascaded to create any filter)	FLP2, FHP2, FAP2, (all pass)	Frequency and Q and optional gain
special 2nd order	FPEAK, FNOTCH, FBPQ, FBP0DB (bandpass)	Frequency and Q and optional gain

Using REW to test a basic program with loopback:
It is important to verify the result of a DSP program with a tool like REW, displaying the response with a frequency sweep, the time response with an impulse or square wave, and to measure group delay of the solution. REW can provide signal on each of the 8 USB-out channels and display FFT or scope view for any of the 8 USB-in channels. It is good practice to provide a time reference signal, with a loopback of USB-Out and USB-in for channel 1 (also called Left).

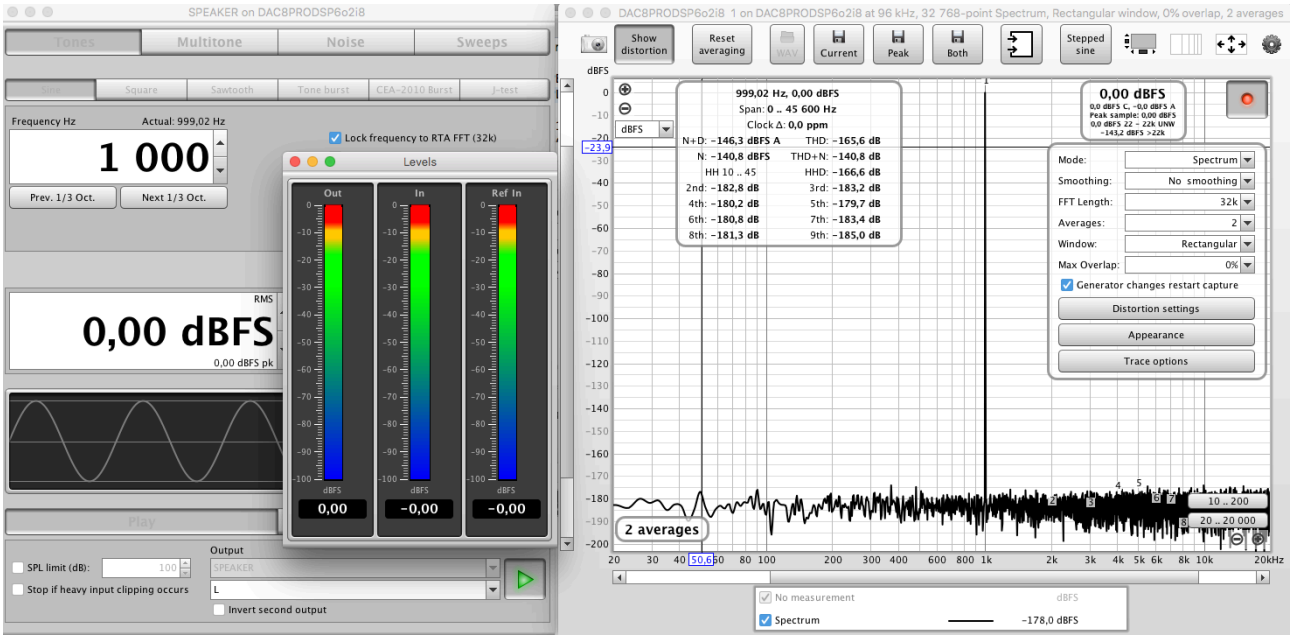
Example of Preference screen :



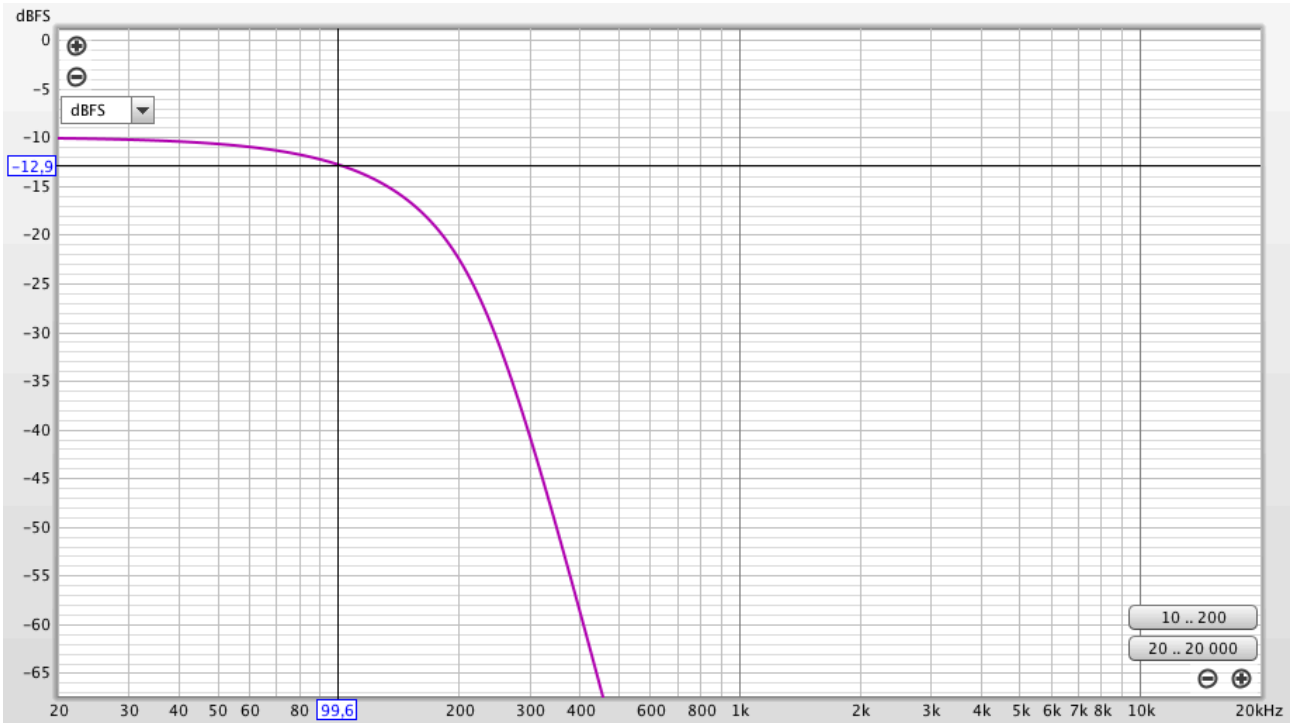
Here is a simple test program with a `transfer` function at the beginning of the program to provide loopback between USB-out 0 and USB-in 0 (16,24). The original signal is also transferred on DAC0 (16,0), a lowpass filter is provided on DAC 1 and USB-in 1, a high pass filter is provided on DAC2 and USB-in 2:

```
lowpass      HPBE8(200)   #bessel filter 8th order (constant group delay)
highpass     HPLR4(400)   #linkwitz-rilley 4th order
core
      transfer (16, 24)(16,0)  #this provides fast loop back on USB channel 0
      inputgain (16, 0db)      #load sample from USB out 0
      biquad lowpass           #apply low pass bessel
      output 1 ; output 25     #output result on DAC1 and to USB for test
      inputgain (16, 0db)      #load sample from USB-out 0 (once again)
      biquad highpass          #apply highpass
      output 2 ; output 26     #output result on DAC2 and to USB for test
end
```

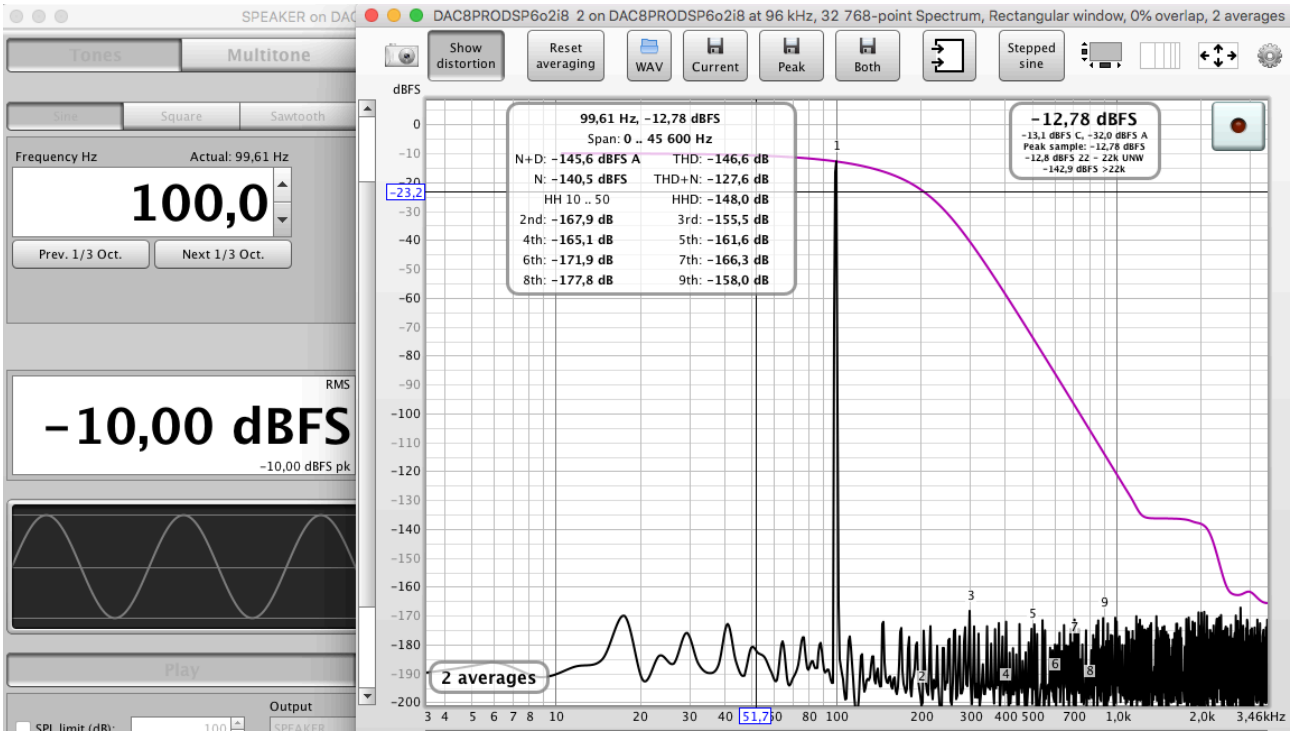
screen when testing loop back with a 0db sine wave as input: no clipping , no THD



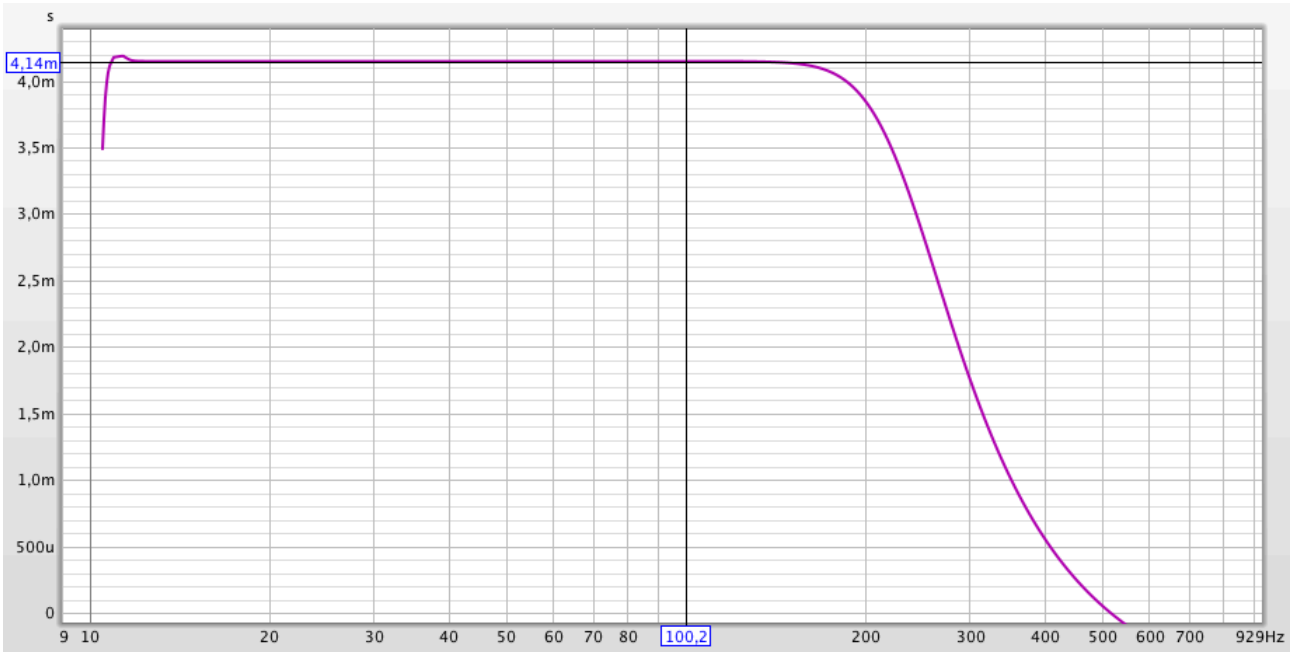
Sweep Measurement on channel 1 with -10db signal in range 10-20000:
LP Bessel 8th response is -2.9@ 100hz, -31@300hz, -77@600hz so 46db/octave (300..600)



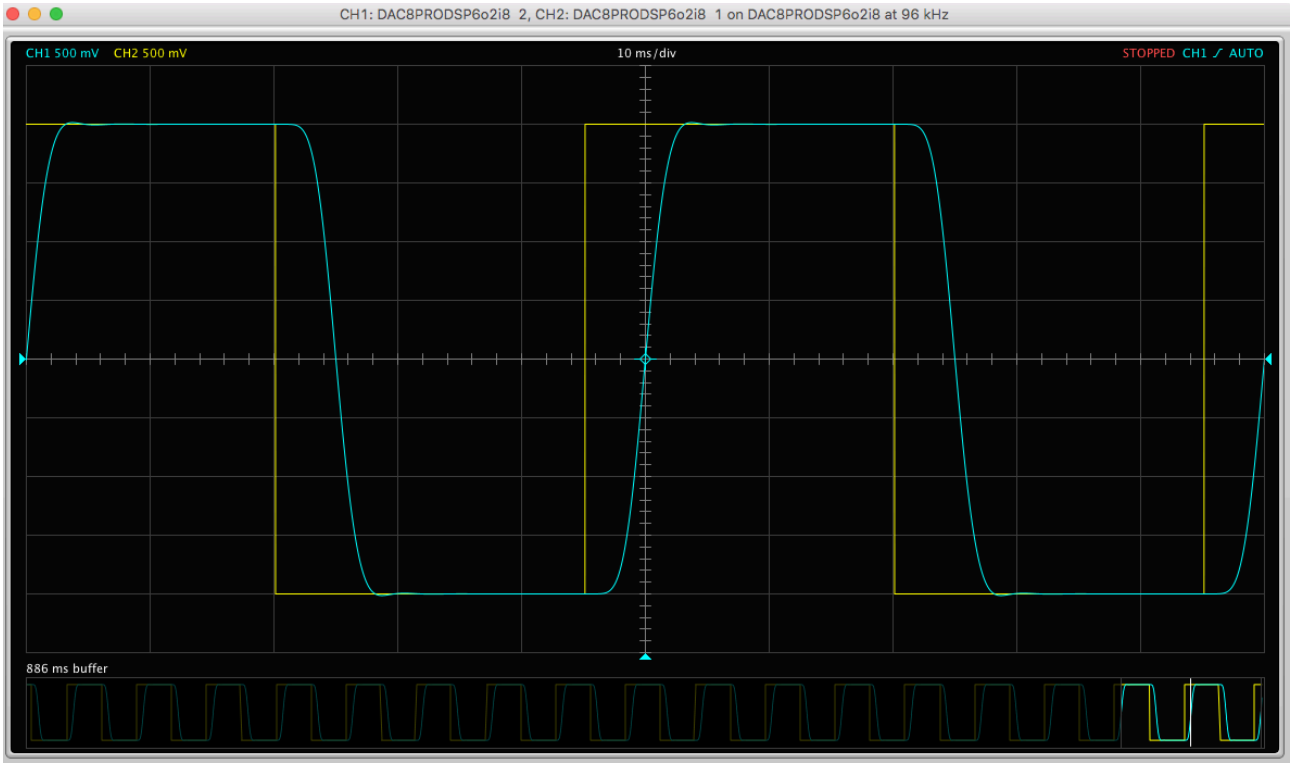
There is no THD for a 100hz test tone (thanks to our specific biquad routine):



As expected, the group delay of the bessel filter is flat below the cutoff frequency, at 4.14ms.



The scope view show a perfect damped step response to a square wave (0db 20hz here) with no clipping:



For the channel 2, response of the high pass LR4:

TBD

Newly introduced tpdf instructions

The `tpdf` instruction provides the possibility to apply N bits dithering with triangular noise.
To use this functionality, the `tpdf` instruction must be defined preferably at the beginning of the first core, with a numerical parameter defining the bit position (8 to 30). This instruction will then compute a signed 31 bits noise value and will apply a mask to keep only (32-N) useful bits. This is done at each sample rate cycle and the noise value can be used in any core.

Then, anywhere in the program and in any core it is possible to use a specific `outputtpdf` instruction which will add this random number to the sample, before storing the value in the target memory location.

It is possible to use the `tpdf` instruction in any core, in addition to the first core : then the number of dithering bits will be specific within this core. for example:

```
core
      tpdf 20
      input 16
      output 24             #original value
      outputtpdf 25         #20 bits dithered value
core
      input 16
      outputtpdf 26         #20 bits dithered (taken from first core)
      tpdf 18
      input 16
      outputtpdf 27         #16 bits dithered value
core
      input 16
      outputtpdf 28         #20 bits dithered (taken from first core)
end
```

In fact, `outputtpdf` instruction always use the number of dithering bits defined by a previous `tpdf` instruction in its own core, otherwise defined in the first core. If no `tpdf` instruction is defined in the program, the encoder will add one automatically when the first `outputtpdf` instruction is encountered. Then the default dithering is arbitrarily chosen to 24 bits.

Saturate instruction details and example
The following program can be tested with the third party tools REW with “Scope” view, by switching the input channels 1 to 4 and by generating a square wave at 0dbfs.

```
highpass   HPBU2(400)
core
      input 16
      output 24         #loopback for reference
      biquad highpass #the result in the DSP accumulator exceed +1..-1
      copyxy            #save result in accumulator Y
      output 25         # signal is clamped sharp between +1 and -1
      copyyx            #retreive accumulator Y
      gain -3db         #accumulator is reduced between +0.8..-0.8
      output 26         #clean as biquad routine provides 18db headroom
      copyyx            #retreive accumulator Y
      saturate          # the result is reduced by 6db automatically
      output 27
end
```

When a `saturate` instruction detects an overflow situation (inside any core) it raises a global saturation flag and clamps the signal between +1..-1. Then at the next sample cycle, the first core checks this flag and eventually increments a saturation counter register.
If a `saturate` instruction sees a number N in this counter register, it reduces the value of the accumulator by N x -6db (shifting N bit right). So the hard clipping happened only on one single sample and this is not audible and has no any consequence on the audio chain.
The only way to reset this saturation number register is to switch on/off the DAC, or reload a DSP program, or switch the sampling rate frequency, or mute/unmute the DAC.

It is recommended to use the `saturate` instruction until all demonstrations are done that no clipping is produced by the user DSP program. Extensive tests can be done with REW square waves at 0dbfs and reviewing each output with the “Scope” view.

Using conditional `core` and `section`

The AVDSP catalog of macros does not include any conditional testing but there is a possibility to dynamically activate or not a core depending on external conditions provided by the DAC firmware in a global status flag and reflecting the DAC modes and status.

By default any `core` declaration is valid and this will use a physical cpu core in the XU216 processor. But it is possible to set a condition with one or two parameters after the keyword `core`. These optional parameters are one or two binary masks, which can be provided as a decimal value or hexadecimal by using the “x” prefix or binary by using the “b” prefix.
The first parameter defines a mask for a bitwise “and” operation with the DAC status flag, and any matching bit set to 1 will enable the core.
The second optional parameter also defines a mask for a bitwise “and” operation with the DAC status flag, but the result of this “and” must be zero to validate the core, otherwise no physical core is allocated during runtime and then the total number of mips is redistributed.

The status flag is reflecting various status and mode as described below:

bit 15..12	dspcores	bit 11..8	AES inputs	bit 7..4	rate	bit 3..0	mode
0001	4	0001	only 1	0001	44/48k	0001	PureUSB
0011	5	0010	2 inputs	0010	88/96k	0010	PureUSB2
0111	6	0100	3 inputs	0100	176/192k	0100	AES/USB
1111	7	1000	4 inputs	1000	352/384k	1000	PureAES

b31..28	chanadc	b27..24	chandac	b23..20	toUsb	b19..16	fromUsb
0..15	ADC/AES	0..15	DAC/DSD	0..15	chan/2	0..15	chan/2

As an example, to activate core when the DAC is configured with a sampling rate below 176khz and only when the mode PureAES is selected, the core statement must be written as
`core 0x30 7` or
`core 0b0011000 0b0111`
then all the code below this core statement will be executed or not.

By extension, the instruction `section` gives the possibility to enable a code section within any active core if the conditions are met. example:

```
core
      section 0b1000   #check if PureAES mode is selected
      input 8          #if yes, read first AES channel
      section 3        #check if any USB mode is selected
      input 16         #if yes, read first USB channel from host
      section          #do nothing but mark the end of the section area
      output 0         #code continues here in all case and update DAC
      section 1        #check if PureUSB mode is selected
      output 24        #if yes provide the host with a loopback
end                    #end also marks the end of the previous section
```

Optimized fixed point 64 bits math explained

The XMOS XU216 is a powerful micro processor with a 32bits instructions sets and some specific instructions able to handle 64 bits. This is used extensively in the AVDSP runtime (using dual-issue assembler programing) in order to provide a PureDSP treatment with almost no THD.

The audio samples provided from the USB host are either 16, 24 or 32 bits signed integers. The sample provided by the AES inputs are 24bits only. They are all represented as 32 bits signed data in memory with format called q31 or s0.31:

original 32 bits audio samples in q31 or s0.31 format	
b31	**b30..0**
sign	31 usefull bits (192db)

The AVDSP runtime is using two 64bits Accumulators called X and Y in this document.
With the standard input instruction the audio samples are loaded in accumulator X with an 8 bits headroom so the 64bits presentation is q56 or s7.56:

audio samples converted to 64 bits accumulator		
b63...b56	**b55...b25**	**b24..b0**
extended sign (00 or FF)	31 usefull bits	25 zeros

A gain value is coded as a 32 bits signed integer, with 1 bit for the sign, 3 bits for the integer part, 28bits for the mantissa, also called q28 or s3.28. This gives the possibility to represent values between +7.999 and -8.0:

32 bits coded values or gain or biquad coefficients as q28 or s3.28		
b31	**b30..28**	**b27..0**
sign	3bits = +18db	mantissa (28 usefull bits 168db)

The inputgain instruction is multiplying a sample s0.31 by a gain s3.28 with a 32x32=64bits multiply instruction which provides a s4.59 base result. The accumulator is then shifted right by 3 to come to s7.56 format, which will be the standard used during all next treatments.

64 bits accumulator as q56 or s7.56 (+/-127.999 = +42db)		
b63	**b62...b56**	**b55..0**
sign (0/1)	integer (7bits)	56 bits accuracy

An output instructions use the specific cpu instruction lsat and lextract to quickly saturate and reduce the s7.56 accumulator to a 32bits s0.31 value which is stored in the output location.

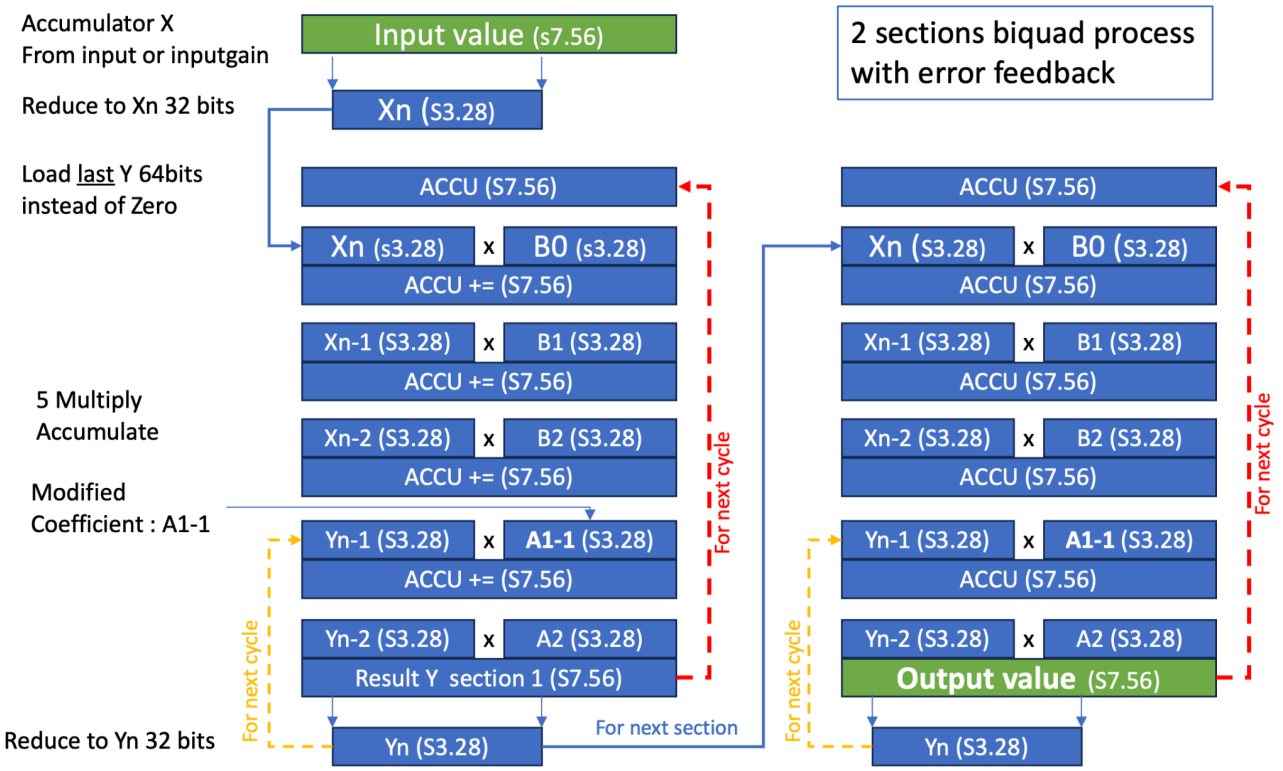
A gain instruction multiplies the 64bits X coded s7.56 by the 32bits gain coded s3.28 with three multiply instructions in a row resulting in a 96bits value coded s11.84. This is then shifted right by 28bits with two lextract instructions to reduce the accumulator to the regular s7.56 format.

An outputgain instruction also combines three multiply instructions and then reduce the 96bits result with the lsat and lextract instructions to get 32 useful bits before storing them in memory.

The mulxy instruction multiplies the two 64bits accumulator X and Y with 5 multiply instructions and keeps only the usefull 64bits so that a s7.56 signal multiplied by a s7.56 signal is reduced as a s7.56 value in the destination accumulator X (or Y with mulyx).

The biquad instruction requires a 32bits Xn sample and produces a 64bits Yn result by applying the five coefficients multiplication (form II): Yn= Xn.b0+Xn-1.b1+Xn-2.b2+Yn-1.a1+Yn-2.a2

Usually a filter is a cascade of multiple biquad sections, and each Yn output is reduced to a 32bits sample before being fed as the input of the next section.
The approach is to reduce the precision of the accumulator X coded s7.56 to a 32bits value Xn coded s3.28 before applying the five multiplies with the coefficient (b0,b1,b2,a1,a2) coded s3.28. The resulting Yn value is coded 64bits s7.56 and will be reused asis in the next cycle, instead of clearing the accumulator as usually done. To compensate for this, the a1 coefficient is reduced by 1.0 during the encoding phase. This is an optimized approach to “truncation error feedback” as explained by Jon Dattorro in this document : https://ccrma.stanford.edu/~dattorro/HiFi.pdf



The benefit is to get a much larger precision at low frequency or low signal, without requiring 64x32 bits multiply instructions as usually done. The resulting THD for a low pass LR4 filter for high fs (192k) is better than -120db in the passband.
This approach is better than using 32bits IEEE float (24bits mantissa+7bits exponents).

The 28 bit base mantissa used for biquad inputs or coefficient or gain value, generating a 56 bit mantissa as described for instructions above, can be changed between 16 and 30 simply by using the option selector -dspformat x in the dspcreate command line. The user can test the benefit of optimizing this by using third party software like REW with the RTA view, with the requirement of using 32 bits audio drivers (like asio or alsa but not java) to better evaluate the benefit of this tuning. Mantissa of 28bits is a good tradeoff to get maximum performance while providing a 42db headroom during basic gain computation and 18db for biquads (remark, all-pass filters require larger coefficient and headroom is then limited to +12db).