EEC0055 - Digital Systems Design

2020/2021

Laboratory project 3 – V0.4
20 November – 23 December 2020

No title

Revision history

date	notes	author
Nov 23, 2020	V0.1 Preliminary version	jca@fe.up.pt
Dec 4, 2020	V0.2 - Figure 1 has been corrected to include a final multiplexer for bypassing module **interpol** - Added the new sections 2.4 – Block **interpol** 3 - Implementation of the FIR coefficient RAM 4 - Matlab/Octave reference model	jca@fe.up.pt
Dec 15, 2020	V0.3 - Section 2.1 has new text with details on the fixed-point arithmetic operations to perform (new text in dark blue)	jca@fe.up.pt
Dec 19, 2020	V0.4 - Section 2.4 has been rewritten to match the current version of the interpolator module (revised text in dark blue)	jca@fe.up.pt

1 – Introduction

This project will implement a digital audio processing system for demonstrating the degradation in the subjective audio quality by changing the sampling frequency and the quantization levels. The input is a high-quality stereo audio signal sampled at 48 kHz with 18 bits per sample, provided by the audio interface system provided with the reference project. The final output signal has the same format as the input signal because this is required to feed the audio CODEC inputs to the DAC.

Figure 1 presents a block diagram of the system to develop. The shaded blocks will be provided later. The inputs **Ncoefs**, **N**, **M** and **K** will be connected to serial interface ports and the selection bits of the multiplexers can also connect to those ports or to the slide switches in the board. Two instances of this system will process independently the left and right audio streams with different configuration parameters.

The detailed specification of each block and hints for their implementation is presented in the next sections. The first block **lowpass** is a digital FIR filter to act as an anti-aliasing low-pass filter. The filter coefficients that will dictate the filter's frequency response will be read from an internal RAM memory whose contents can be uploaded via the serial interface. The second block **downsample** reduces the original 48 kHz sampling frequency to an integer fraction of that, creating a signal sampled at Fs1= 48 kHz / **N**, where **N** is an integer between 2 and 10. Next block **requantize** reduces the number of bits of the audio samples from the original 18 bits to **M** bits, where **M** can be any number between 17 to 3. Last block **interpol** is a **K**-order interpolator that will increase the sampling frequency to the 48 kHz required to feed the digital to analog converters of the audio CODEC. This block will be provided as verified Verilog RTL model (or *IP-block*).

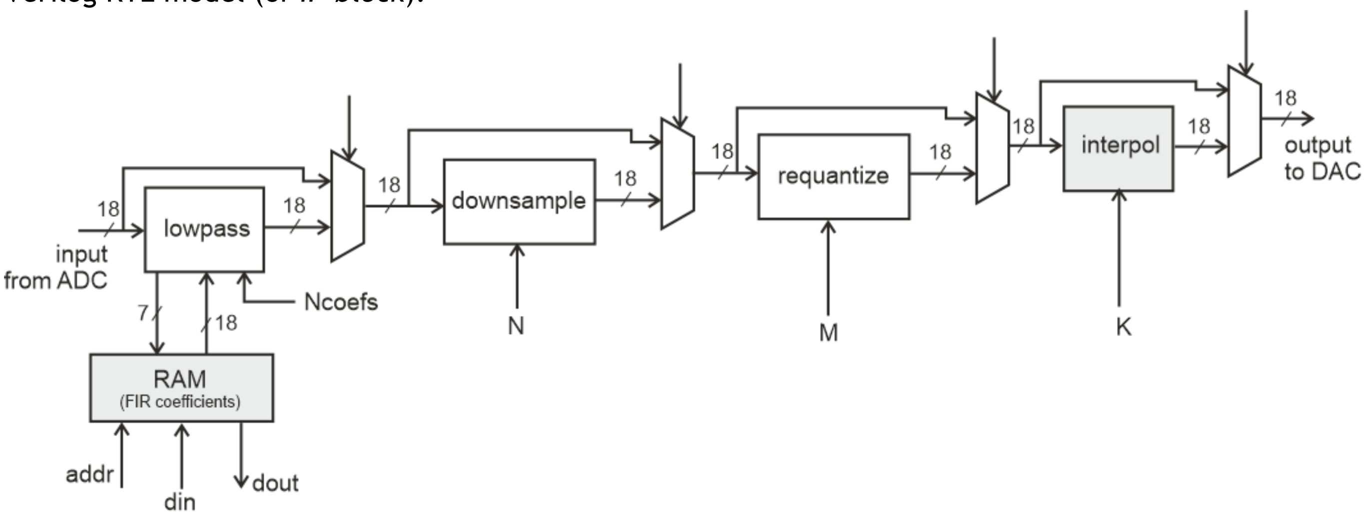

Figure 1 – Block diagram of the system to develop. Two instances of this system will process independently the left and right audio streams with different configuration parameters.

The multiplexers between each block will allow to insert or bypass individually each function in the audio processing chain. Both the left and right audio streams will be processed by similar systems, although the configuration parameters will be independent for each channel (the filter coefficients, the **M**, **N** and **K** parameters and the selection bits of the multiplexers).

The whole digital system should be synchronous with a single clock signal running at 12.288 MHz. The input audio samples are synchronized by the 48 kHz clock enable signal **DIN_RDY**, generated by the audio CODEC controller. This signal lasts for a single clock cycle and must be used as an enable of the master clock:

```
always @(posedge clock)
if ( DIN_RDY )
   // do whatever has to be done with the new audio sample
```

and never as a clock signal:

```
always @(posedge DIN_RDY)
  // this cannot be done, you should be able to explain why!
```

The example **psdi_dsp.v** provided in the reference project illustrates the process of using this signal. As the block **downsample** reduces the sampling frequency by N, it may also generate a similar clock enable signal with the frequency of the downsampled signal generated by this block (with frequency 48/N kHz).

As the clock frequency is already defined, the main design goal is to minimize the circuit size, measured as the number of look-up tables and flops used by your module, when synthesized alone. The whole circuit must be assembled and verified, at least at the functional level, using a testbench adapted from the testbench given in the reference project.

2 – Functional description and suggestions for implementation

This section presents the detailed functional description of each block (except module **interpol** that will be provided for free) and some suggestions for building their implementation.

2.1 – Lowpass FIR filter

The FIR (Finite Impulse Response) digital filter calculates the discrete convolution between the input signal x_k and the filter impulse response h_i:

$$y_k = \sum_{i=0}^{M-1} x_{k-i} \times h_i$$

The filter coefficients h_i (or the finite impulse response) is formed by M 18 bit words stored in a RAM memory. Each coefficient is represented by a fixed point word, with 12 bits for the fractional part and 6 bits for the integer part (more details about fractional binary arithmetic will be presented later). The minimum length of the FIR filter is M=65.

This specification enforces that the memory holding the filter coefficients (and consequently the memory storing the previous samples of the input signal) must be implemented with a minimum size of 65 words. The RAM memory provided in the design kit is configured as a 128 word memory, which will support filters up to 128 coefficients, but the designer is free to modify it in order to make a better implementation using a smaller RAM. Note that the shorter FIR filters can always be implemented by setting to zero the higher-order coefficients.

The input signal is formed by 18-bit signed integer samples and the output of the filter must be represented in the same format. If the filter is designed in order to guarantee a gain less than or equal to 1 (or 0 dB), then the result calculated from the expression above will fit in 18 bits without overflowing. However, the intermediate accumulated results while performing the summation may require more than 18 bits to prevent overflow from occurring. Considering that the filter coefficients are represented by 6 integer bits and 12 fractional bits, each result of x_{k-i} x h_i will require 36 bits (24 integer bits and 12 fractional bits). If the maximum filter length is 128, then more 7 integer bits will be required to guarantee that overflow will not happen. The summation should thus be computed with 43 bits, where 12 are fractional bits that can be discarded (or truncated, not rounded) and 31 are integer bits. However, if the maximum gain is 1, the result will fit in only the least significant 18 bits of the integer part.

The example below illustrates the calculation process using fixed point arithmetic, where the "bb…bbb", "hh…hhh" etc. represent bits of each element involved in the calculation process.

x_k = bb bbbb bbbb bbbb bbbb   (signed integer, two's complement)

h_k = hh hhhh . hhhh hhhh hhhh (signed fixed-point, two's complement; the value represented as a fixed-point number is the value of the 18-bit signed integer divided by 2^12)

x_k x h_k = pppp pppp pppp pppp pppp pppp . pppp pppp pppp (12 fractional bits, 24 integer bits)

Sum(x_k x h_k) = sss ssss ssss ss **ss ssss ssss ssss ssss** . ssss ssss ssss (the fractional part – "red" bits - can be discarded! The green "s" bits will contain the final output value)

This example may suggest that a 43-bit accumulator will be required to accumulate the products x_k x h_k. However, if the final result will be contained in the 18 "green" bits, the 13 leftmost "black" bits do not need to be implemented in the accumulator. Actually, if they are implemented but not connected to any part of the circuit, the synthesis optimization processes will remove (or trim) all the parts of the circuit implemented to compute exclusively those 13 "black" bits.

For this implementation to work it is necessary that the filter coefficients uploaded to the RAM define a filter with a magnitude gain not greater than 1. If this condition is not met overflow may occur with unpredictable results. We assume it is the responsibility of the final user to guarantee this condition.

The digital implementation of a FIR filter requires another memory, or a bank of registers, to hold the M previous samples of the input signal x_k. This can be done with another RAM or a register array organized as a shift register. As there are 256 clock cycles between the arrival of each new sample, this computation can be done as a sequential process requiring only one multiplier and one accumulator, besides the storage elements referred above.

A memory can be implemented in Verilog as an array of registers. An example of declaration of a register array with 64 words of 18 bits each is:

```
reg [17:0] mymemory[0:63];
```

The write and read access to the memory must be synchronous with the clock. Only one data can be read and written in the same clock cycle but both operations can occur in the same clock, as the memories can be built as dual-port memories with two independent read/write ports. The Verilog code below illustrates the implementation of a synchronous write to a memory and a synchronous read of the memory to a register. For more information on how to model memories in Verilog for the Spartan6 FPGA family refer to the XST user manual (google "Xilinx ug687", chapter 7 – HDL coding techniques)

```
always @(posedge clock)
begin
  if ( write_enable )
     mymemory[ write_address ] <= data_to_write;

  if ( read_enable )
    data_read_from_memory_to_a_register <= mymemory[ read_address ];
end
// in alternative reading the memory asynchronously:
assign data_read_from_memory_combinational = mymemory[ read_address ]
```

The algorithm to implement the FIR filter sequentially can be summarized by the following pseudocode. Note this is not Verilog code neither it is intended to represent the structure of Verilog code. Also, although the cycle `for( )` exists in Verilog and is synthesisable (only for certain specific constructs), the functionality of the `for( )` loop must be implemented by using a finite state machine and not a Verilog `for( )` loop.

```
Mcoef[0:M-1] is the coefficient memory holding the filter coefficients
Mxin[0:M-1] is the memory holding the x(k-i) input samples
Repeat forever:
  Wait for DIN_RDY == 1, this means a new sample x(k) has arrived
  Set Y(k) to 0
  Store x(k) to memory Mxin and delete the oldest input sample
  for i=0 to M-1
        Read h(i) from Mcoef memory and read x(k-i) from Mxin memory
        Calculate Y(k) + h(i) * x(k-i) and store the result to Y(k)
  endfor
```

The memory **Mcoef[ ]** will be an external block to your module. From the point of view of your module, this will operate as a read-only memory behaving as a combinational circuit: your module generates a 7-bit address and the memory responds the 18-bit coefficient stored in the addressed location. The memory will actually be implemented as a dual-port RAM with a read/write access port connected to the serial

interface for supporting the upload of the filter coefficients. The other memory holding the input samples will be created inside of your module and can also be implemented with a shift-register structure.

The Verilog module interface of this block should be:

```
module lowpass(
              input clock,   // Master clock
              input reset,   // master reset, synchronous, active high

              input  [17:0] datain,  // input data
              input         endata,  // data clock enable
              output [17:0] dataout, // output data

              // Interface to the coefficient memory:
              output [ 6:0] coefaddress,  // 7-bit address
              input  [17:0] coefdata      // 18-bit data
            );
```

2.2 – Block downsample

This module receives a sequence of data samples at the sampling frequency of 48 kHz, synchronized by signal DIN_RDY, and reduces the sampling frequency to an integer multiple N of the input sampling frequency (N must be between 2 and 10). This module outputs the input sample received at each N sampling cycles, as illustrated by the following example for N=3:

```
Input data @ 48 kHz:  x0  x1  x2  x3  x4  x5  x6  x7  x8  x9  x10  x11  x12  x13
Output data @ 16 kHz: x0          x3          x6          x9             x12
```

This module should also generate a clock enable signal with frequency equal to 48 kHz / N.

The interface of this module should be:

```
module downsample(
              input clock,   // Master clock
              input reset,   // master reset, synchronous, active high

              input [3:0]   Nfreq,    // sampling rate divide factor

              input  [17:0] datain,    // input data
              input         endatain, // in clock enable, Fs=48kHz
              output [17:0] dataout,   // output data
              output        endataout // out clock enable, Fs = 48kHz/Nfreq
            );
```

2.3 – Block requantize

Module **requantize** reduces the number of bits of the input samples to a smaller number M between 17 and 3 and scales the output to the 18 bit dynamic range to keep the output samples represented in 18 bits. This module performs the following operation, where x_k represents the input sample and y_k is the output sample:

$$y_k = round(x_k / 2^{(18-M)}) \times 2^{(18-M)}$$

The rounding process should be implemented as follows, depending on the magnitude of the fractional part of the results of the division $x_k / 2^{(18-M)}$:

- If less than 0.5 round down;
- If greater than 0.5 round up;
- If exactly equal to 0.5 round to the nearest even (or round down if the integer part is even and round up if the integer part is odd)

Note than in binary the division by 2^K corresponds to moving the fractional point K positions to the left and the bits that remain at the right of the fractional point represent the fractional part of the quotient. If those bits are 0.0xx…xxx the fractional part is less than 0.5 and the result should be rounded down; if the fractional bits are 0.1bb…bbb and at least one of the "b" bits is one, then the fractional part is greater than 0.5 and the result should be rounded up by adding one unit to the integer part; if the fractional bits are 0.100…000 than a one must be added to the integer part if it is an odd number. A few examples of 8-bit words requantized to 4 bits and implementing this rounding process:

```
1010_0110 -> 1010_0000  // rounded down
0101_1001 -> 0110_0000  // rounded up
0011_1000 -> 0100_0000  // rounded up to the nearest even
0010_1000 -> 0010_0000  // rounded down.
```

The interface of this module should be:

```
module requantize(
                input clock,   // Master clock
                input reset,   // master reset, synchronous, active high

                input [4:0]   Nquant,    // No. of output quantization bits

                input  [17:0] datain,   // input data
                input         endatain, // input data clock enable
                output [17:0] dataout   // output data

               );
```

2.4 – Block interpol

As referred before, this module is available as a synthesisable RTL module. This performs a first-order (or linear) interpolation between the input samples received at a lower sampling rate and generate an output sampled at the original 48 kHz. This module receives the output of the **requantize** module and the data enable defining the sampling frequency of the signal at this stage (the input **endatain** of module **requantize**), performs the linear interpolation and outputs the signal that will connect to the DAC output ports in the top level module (signals **RIGHT_out** and **LEFT_out**).

Figure 2-a shows the expected output after performing a linear interpolation on a signal sampled at 12 kHz to increase the sampling frequency to 48 kHz and figure 2-b shown simulation waveforms for a decimation factor (input Nfreq) equal to 3 and 6. Figure 2-c waveform shows the latency between receiving one input sample and outputting the corresponding output sample. This is equal to the decimation factor (Nfreq) plus 1.

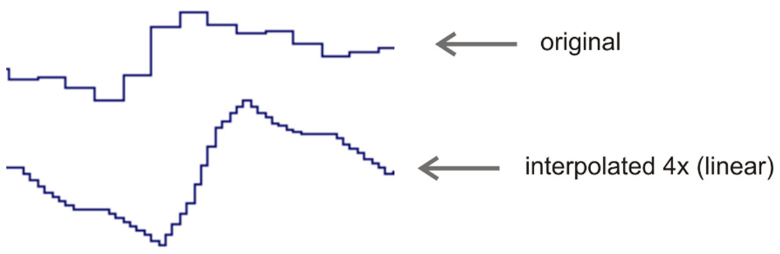

Figure 2-a – Linear interpolation to increase the sampling frequency by a factor of 4

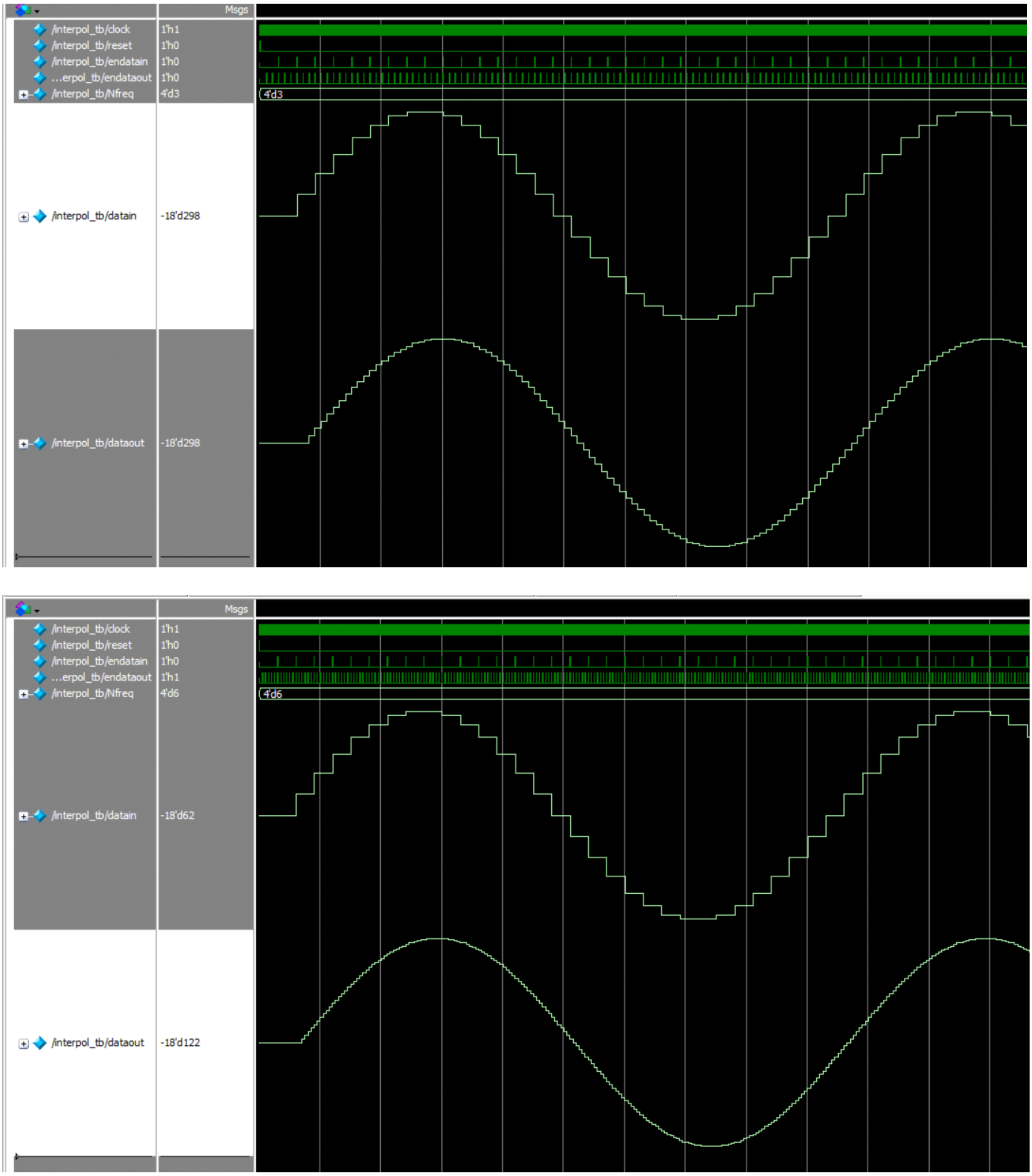

Figure 2-b – Example of simulation waveforms for decimation factors 3 (top) and 6 (signal Nfreq).

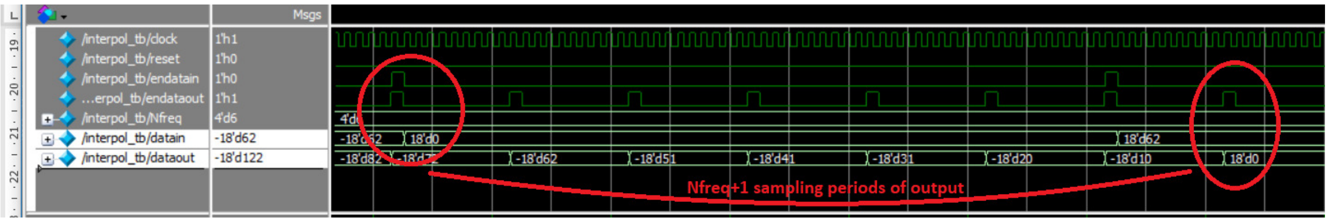

Figure 2-c – The latency of interpolator is equal to a number of sampling periods of the output signal (48 kHz) equal to the decimation factor plus 1 (in this example this is 7 periods of 48 kHz).

The interface of this module is:

```
module interpol (
    input clock,     // master clock
    input reset,     // master reset, synchronous, active high
    input endatain, // enable input data @ low sampling frequency
    input endataout,// output enable, this should be signal DIN_RDY (48kHz clk enable).
    input [ 3:0]  Nfreq, // Connect this input to the decimation factor Nfreq

    input
      signed [17:0] datain, // input data
    output
      signed [17:0] dataout // output data
    );
```

3 - Implementation of the FIR coefficient RAM

The design kit V0.4 includes now a RTL module of a dual-port RAM memory with 128 words of 18 bits, to hold the coefficients of the low-pass FIR filter (module **./src/verilog-rtl/DPRAM.v**). The memory is implemented as a synchronous RAM that is pre-loaded at synthesis time with the contents of file **./simdata/FIR.hex**. This file is generated by running the Matlab/Octave script **./matlab/psddsp.m** (see section 4) with the FIR coefficients quantized to 18 bits and 12 fractional bits.

The toplevel design **s6base_top.v** instantiates this module with the read/write port already connected to the serial interface to allow uploading the memory and change the FIR filter in real-time (lines 300 to 315). Your circuit should use only the ports **addr2[6:0]** and **dataout2[17:0]** that are presently connected to module **psdi_dsp** only to exemplify how to connect them. Note that this memory is not used by **psdi_dsp** and this module just assigns the address bus to zeroes.

You are free to adapt this code for implementing other memories that may be needed for your design.

4 - Matlab/Octave reference model

The reference Matlab/Octave model ./matlab/psddsp.m has been updated to generate simulation data from an audio .wav file. The differences to the previous version are:

- Runs transparently in Matlab and in Octave. The interpolation order is forced to 1 (linear interpolation) when running in Octave;
- Creates hex files in folder **./simdata** to be read by the Verilog tasks **$readmemh()** included in the top level testbench **s6base_tb.v**, with the streams of data extracted from the input audio file and the golden output files with the expected results. The number of samples to write to these files is defined by variable **NhexSamplesForTb** (line 54) and its value must not be greater than the number of samples corresponding to the duration of the signal used to create the final output audio file (variable **duration**, line 50);
- Creates the hex file ./simdata/FIR.hex with the coefficients of the low-pass FIR filter. This file is loaded at synthesis time to the coefficient RAM model.

To configure a simulation you may adjust the parameters between lines 27 and 75 to select the input audio file, define the duration of the audio file to be processed, configure the operation of the signal processing path and select to include or bypass each one of the 4 blocks.

… to be continued …