2IN35 – VLSI Programming – Lab Work

Assignment 4: Audio scaler

Hrishikesh Salunkhe, h.l.salunkhe@tue.nl,
Alok Lele, a.lele@tue.nl

June 2, 2015

Contents

1 Introduction

In this assignment you will design and implement an audio scaler, also known as a sample rate converter. For this assignment you need the following files from the website:

- Dummy scaler implementation: `scaler.zip`
- Multi rate EDK systems: `L4_audio-multi-rate-offline.rar` and `L4_audio-multi-rate-realtime.rar`
- Example audio files: `input.bin`

2 Background

A scaler, also known as sample rate converter, is used to convert a signal from one sample rate to another, while changing the information in the signal as little as possible. The scaler may convert a signal to a higher sample rate, in which case we call it an upscaler, or to a lower sample rate, in which case we call it a downscaler. In this assignment we will be converting from a sample rate of 44.1 kHz (used on audio cds) to 48 kHz (used on digital audio tape), which requires an upscaler.

The main problem of scaling is that we have to construct samples at positions for which there is no data available in the original signal. This is illustrated in figure 1 for input signal $x[n]$ and output signal $y[k]$. To solve this, we need a scheme to determine a value in between 2 adjacent input samples.

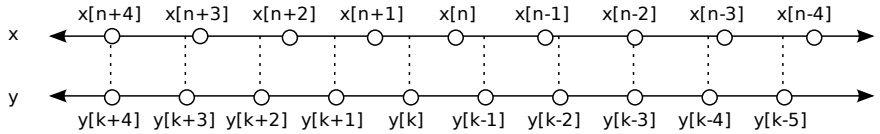

Figure 1: The input signal x[n] has lower sampling frequency than output signal y[k], so interpolation is necessary.

Assume that the input sample rate is F_x and the output sample rate is F_y. If the following relation holds $\frac{F_y}{F_x} = \frac{L}{M}$, for some integers L and M, then this problem can be solved in 2 steps: first convert the input signal to an intermediate signal with sample rate F_xL and then convert that signal to a signal that has sample rate $\frac{F_xL}{M} = F_y$. That leaves us with 2 simpler problems to solve.

First, we need to generate the intermediate signal from the input. Increasing the sample rate to F_xL is done using an upsampler, that introduces $L-1$ zeroes in between each pair of consecutive input samples. However, this will introduce a lot of high frequency noise outside of the frequency range that can be represented in the input signal. So we need to low-pass filter the resulting signal to only pass the frequencies that are representable in the input.

Second, we need to generate the output signal from the intermediate signal. Decreasing the sample rate to $\frac{F_xL}{M}$ is done by a downsampler, that discards $M-1$ input samples for

every output sample. However, if the intermediate signal contains frequencies that cannot be represented in the output, we have to low-pass filter the intermediate signal first.

To build the the $\frac{L}{M}$ scaler, we connect the above 2 systems in series, resulting in a series connection of 4 components: an upsampler, 2 low-pass filters, and a downsampler. The 2 low-pass filters can be combined into 1 low-pass filter: simply take the one that has the lowest cut-off frequency, i.e. the one that removes the largest number of frequencies. The block diagram of this system is illustrated in figure 2, and its behavior in figure 3.

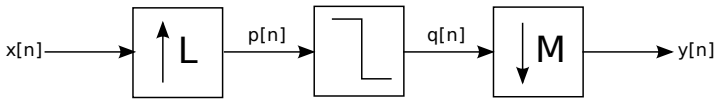

Figure 2: A $\frac{L}{M}$ scaler with input stream $x[n]$ and output stream $y[n]$

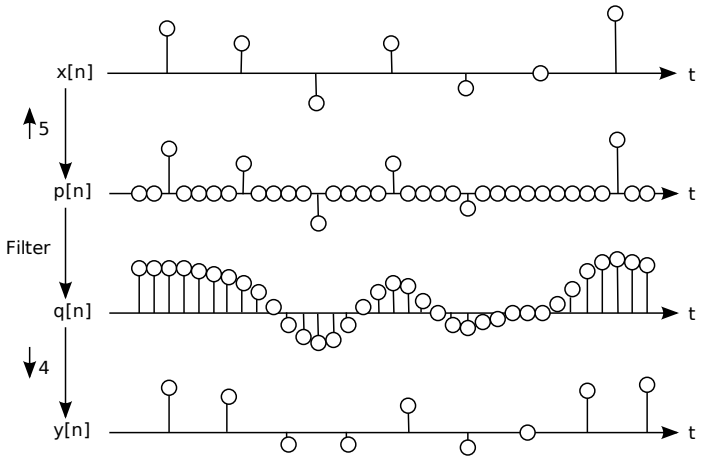

Figure 3: Behavior of a $\frac{5}{4}$ scaler on an input stream $x[n]$

The requirements for the combined low-pass filter are that all frequencies above $\frac{F_x}{2} \downarrow \frac{F_y}{2}$ Hz (the maximum frequency that is representable in both the input and output stream) should be removed, and all frequencies below it should be passed. This ideal behaviour can only be obtained using a sinc function to generate the filter coefficients. The definition of the sinc function is:

$$\text{sinc}(t) = \begin{cases} 1 & t = 0 \\ \frac{\sin(\pi t)}{\pi t} & t \neq 0 \end{cases} \tag{1}$$

For the upscaler, the coefficients for the filter are $\text{sinc}(\frac{n}{L})$ for $n \in \mathbb{Z}$. Because the sinc function goes to zero only in its limits, we get an infinite amount of non-zero coefficients. It is impossible to implement such a filter, so what is used in practice is a windowed sinc function, which is only non-zero for a finite length interval. The lanczos2 function is an example of a

windowed sinc function (see also figures 4, 5):

$$\text{lanczos2}(t) = \begin{cases} 0 & t \leq -2 \\ \text{sinc}(t)\text{sinc}(\frac{t}{2}) & -2 < t < 2 \\ 0 & t \geq 2 \end{cases} \tag{2}$$

Using the lanczos2 function, the coefficients for the upscaler's low-pass filter become lanczos2($\frac{n}{L}$) for $n \in \mathbb{Z}$. Because the lanczos2 function is zero only at the interval $(-2, 2)$, we only need $4L$ filter coefficients, and is therefore implementable by a simple FIR-filter.

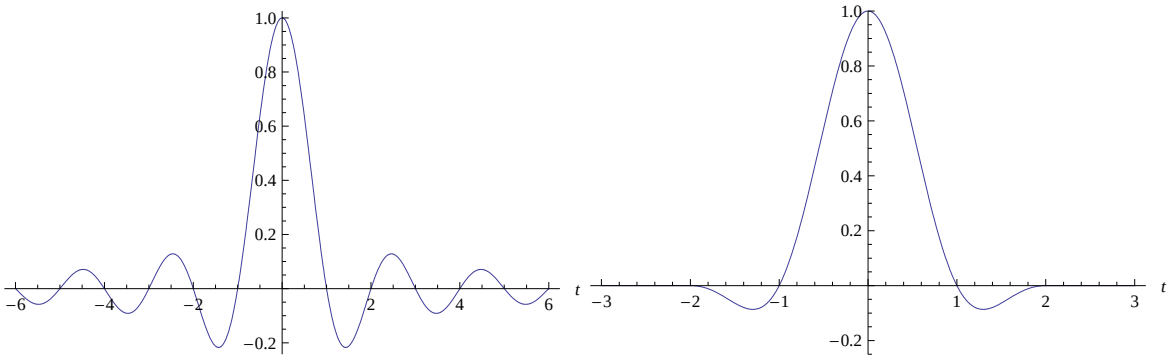

Figure 4: The sinc function (left) and the lanczos2 function (right)

The sum of the coefficients of the low-pass FIR filter should always add up to 1. This is easy to see if we a apply a signal of constant value (a DC signal). This signal can be seen as having an infinitesimal frequency, so should be passed undistorted. According to the FIR filter equation, the only way to get this input signal back is by letting the coefficients sum to 1. If the coefficients are generated using the sinc function, this property holds. However, if the lanczos2 function is used, it doesn't hold. This means that for the lanczos2 function an additional normalization step is requires to force the coefficients to sum up to 1.

However, if we look at the average intensity of the upsampled signal, it is only $\frac{1}{L}$ times that of the input signal. So to get the input signal intensity back, we have to multiply the normalized coefficients by L, effectively letting them sum to L.

3 Optimizations

The zeroes introduced by the upsampler are processed by the FIR filter, which means that of every L multiplications performed by the FIR filter, $L-1$ of them are multiplications between a zero and a coefficient. Furthermore of all of the samples leaving the FIR filter only $\frac{1}{M}$ are actually used in the output of the scaler. Clearly, this is a waste of resources if implemented naively.

If we write down the equation for the system and then expand and simplify it, we obtain the

following equivalent formula:

$$y[n] = \sum_{j=0}^{3} h[jL + nM \ \mathbf{mod}\ L]x[nM \ \mathbf{div}\ L - j] \tag{3}$$

$$h[n] = \text{lanczos2}(\frac{n}{L} - 2) \tag{4}$$

Note that $h[n] = 0$ only for $n \in \{0, L, 3L\}$, so that there are still some multiplications with zero taking place, but not many.

Scalers can be implemented by chaining together several scalers to form a composite scaler. To implement a scaler with a ratio of $\frac{15}{4}$ one might use two scalers of ratio $\frac{3}{2}$ and $\frac{5}{2}$. In such an implementation care has to be taken that the sample rate of the signal between the individual scalers does not fall below the output sample rate of the composite scaler. Otherwise information in the signal will be lost.

Now we basically have 2 options with 2 choices each when implementing a scaler. First of all, we can either implement a scaler directly, or compose it from smaller scalers. Secondly, we can either use the naive implementation with upsampler, FIR filter and downsampler or use the direct equation for $y[n]$. This leads to four possible designs, and which of the designs is best may depend on the application.

Note that all equations given above are for upscalers only. So if you build a composite scaler, make sure every individual scaler is an upscaler. For the ratio $\frac{160}{147}$ (which is used for upscaling from 44.1 to 48 kHz) this can be accomplished by using scalers of ratios $\frac{64}{63}$ and $\frac{15}{14}$.

Questions

1. Verify the direct equation for $y[n]$
2. The equation for $y[n]$ uses different coefficients for each output $y[n]$, however the pattern of coefficients repeats. What is the period of this pattern?
3. Examine for each of the the four possible designs:
   - The number of multiplications performed per output.
   - The total number of coefficients required.
   - The highest sample rate inside the system.

4 Design & Implementation

You have to design 1 of the 4 upscaler implementations that you have examined in the previous section. The specific requirements are:

- The output sample rate must be $\frac{48000}{44100}$ times the input sample rate.
- The output represents a resampled version of the input.
- The design must support an input sample rate of at least 44.1kHz.

- The design must be able to run at a clock frequency of at least 100Mhz.
- The design may produce start-up noise.
- The top-level Verilog module must be called `filter`.

If possible, try to optimize your design for minimum resource utilization, or for maximum throughput.

The interface for this assignment is the same as for the previous assignment, except that there is no coefficient input. There still is an input port and an output port that both use the 4-phase handshake protocol to exchange data items. However, the behaviour is different for a scaler, because the rate at which inputs are consumed is not the same as the rate at which outputs are produced.

The formulas for the scaler assume that all numbers are reals. However, we cannot work with reals in practice, so we need to *quantize* all values. We can use either floating point or integers. Floating point is very slow (unless heavily pipelined) and expensive to implement in an FPGA, which is why it isn't supported by the Xilinx synthesizer. Furthermore, the large dynamic range offered by floating point is not necessary for most signal processing applications. So integers is usually the best choice.

The input samples that the system receives are already signed 16 bit values. Only the coefficients remain to be generated and quantized. This can be done by multiplying the floating point values by a power of 2, say ($2^{16} = 65536$) and then round off. The resulting coefficients should lie in the range $[-32768, 32768]$. After computing the sum, the result should be divided by 65536 and rounded again to obtain the proper output. Division by 65536 is trivial in hardware: discard the 16 least significant bits. Another way of looking at this is that 16 fractional bits are used during the computation.

Make sure your intermediate values have enough bits. Adding 2 n-bit numbers results in an $n + 1$-bit number, and multiplying 2 n bit numbers results in an $2n$-bit number. If you do not reserve enough bits, the resulting samples might be wrong. Before outputing the result of the computation, also make sure to check for overflows: if it occurs, set the output to the closest representable output value. Overflows are impossible to avoid using the lanczos2 filter (can you see why?).

Once you have quantized the coefficients, they can be stored in read only memory. Refer to the ISE language templates (found in the edit menu) to find out how to instantiate a ROM and how to load the contents from a file using the `$readmemb` and `$readmemh` functions. The coefficient files themselves should specify the coefficients in ASCII text (not in binary!), one coefficient per line. E.g. 255 would become 0ff in 10-bit hexadecimal.

Even if you use the direct form equation for $y[n]$ as your starting point, it might still be a good idea to break the problem down into smaller components that communicate with each other using the same 4 phase handshake protocol that is used to communicate with the environment.

5 Testing & Running

The procedure to test and run your design is analogous to the previous assignments: `scaler.zip` contains a test bench that you can use to validate your design, and `L4_audio-multi-rate-offline.rar` and `L4_audio-multi-rate-realtime.rar` contains the EDK systems in which you can plug your design to compile it to a bit-file.

If you are going to compile your design on the N-grod servers, don't forget to copy the files that hold the content of the ROMs to the servers as well.

The terminal-interface that is presented to you over the serial connection is the same as for the previous assignments, but you can ignore everyhting that has to do with coefficients.

6 Reporting

In your final report, answer the questions that are stated in this assignment. Furthermore, compare input with output both in the time domain and in the frequency domain to assure that the output is a properly interpolated version of the input. If there are any differences, explain them. Audacity and Matlab are very useful to help you with this. Further reporting guidelines are on the course's website.

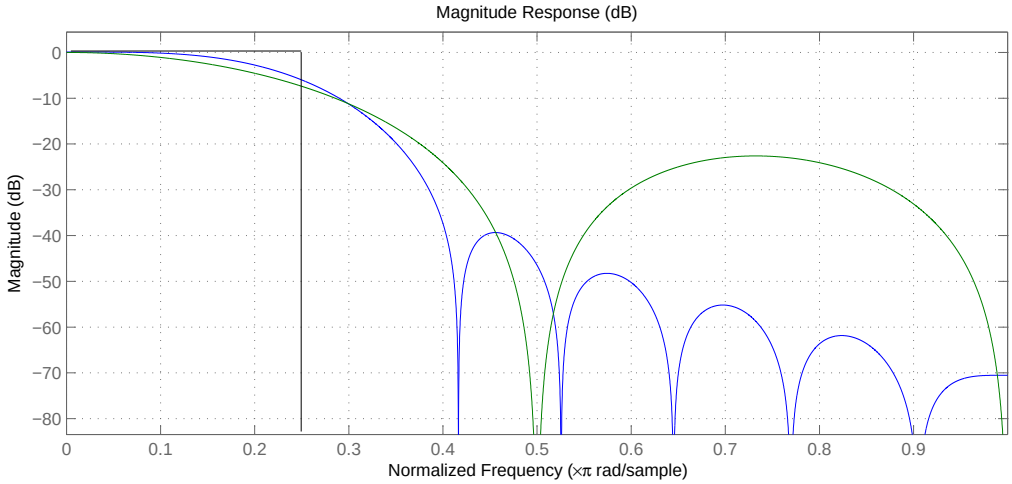

Figure 5: The magnitude response of the sinc function (black), lanczos2 function (blue) and of linear interpolation (green)