HW/SW Co-Design
Main Task
Realtime Reverb Filter with Zero Delay

Vienna University of Technology

October 29, 2018

1 Assignment

The goal of the main task is to implement a real-time stereo reverb filter with zero delay. The basic algorithm for this task is quite simple. In essence it is just a very long FIR filter.

$$y[n] = \sum_{i=0}^{N-1} b_i * x[n-i] \tag{1}$$

To achieve a minimal filter delay, you have to generate a new output sample immediately[1] after reading an input sample. Hence, it is not possible to simply collect a certain number of samples and perform an FFT-based convolution on this block, because this would introduce a delay of at least the time it takes to record the samples plus the execution time of the FFT-based algorithm. A direct FIR filter (i.e. a filter that directly implements Equation 1) fulfills the low-delay property and is quite straight forward to implement. However, with a large number of taps direct FIR filters grow very inefficient, which excludes them as an option. Consult [2] and [1] to find out how to tackle this problem.

The sample bit-width is 16, i.e. each stereo sample consists of 32 bits of data. Choose an appropriate (fixed-point) number format for your implementation.

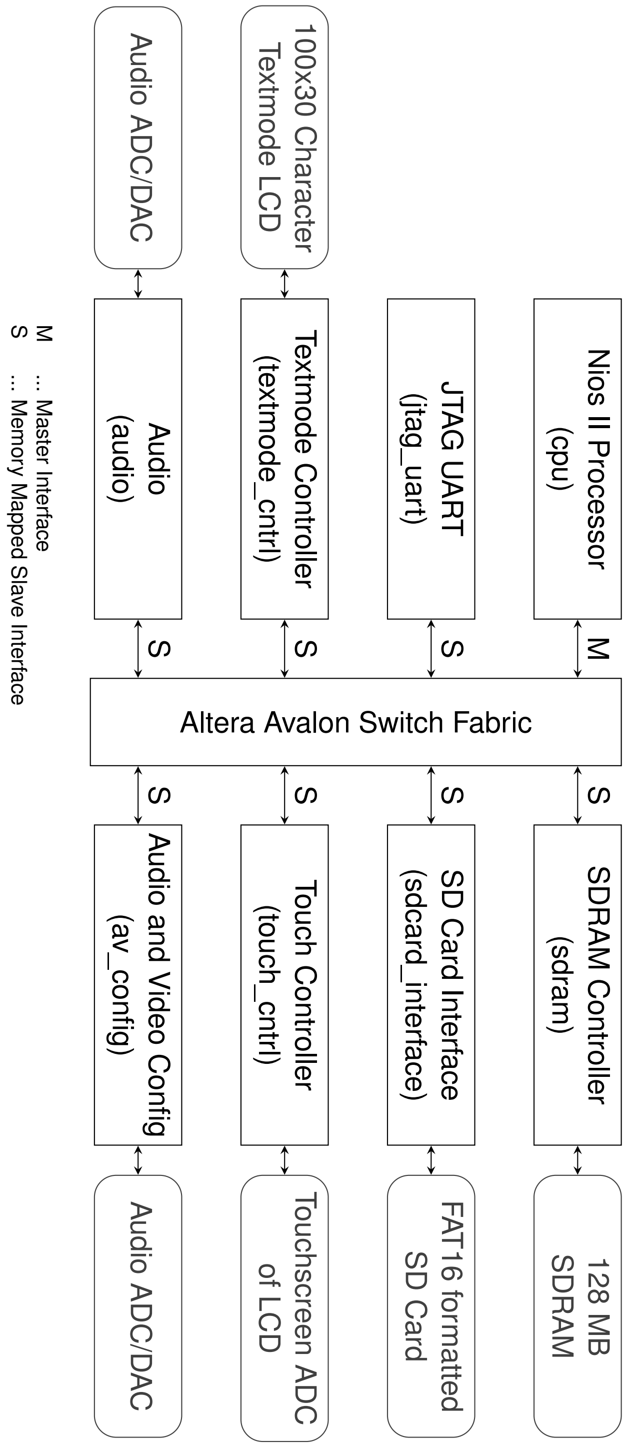
1.1 Template

You are provided with a template Nios II system that is already capable of

- Reading and writing a FAT 16 formatted SD card
- Playing wave files using the board's audio DAC (line-out, green audio jack)
- Recording from the board's audio ADC (line-in, blue audio jack) to a wave file
- Passing through audio data from line-in to line-out

[1]with 1-2 sample periods tolerance

Figure 1 shows an overview of the system's components. For the sake of clarity the PLLs have been omitted from the figure. The main system operation frequency is 100 MHz generated by a PLL from the 50 MHz board clock. On top of that the system also requires special clocks for the display as well as the audio ADC/DAC. Note that for the SDRAM interface there is an additional 100 MHz clock, which must have a particular phase relation to the main system clock. Hence changing the 100 MHz clock also involves changing this phase shift (otherwise the system will not work anymore)! Further details on this topic can be found in the manual of the SDRAM controller.



Nios II Processor (cpu)
JTAG UART (jtag_uart)
Textmode Controller (textmode_cntrl)
Audio (audio)
100x30 Character Textmode LCD
Audio ADC/DAC
M
S
S
S
Altera Avalon Switch Fabric
S
S
S
S
SDRAM Controller (sdram)
SD Card Interface (sdcard_interface)
Touch Controller (touch_cntrl)
Audio and Video Config (av_config)
128 MB SDRAM
FAT16 formatted SD Card
Touchscreen ADC of LCD
Audio ADC/DAC
M ... Master Interface
S ... Memory Mapped Slave Interface

Figure 1: System Overview

- Nios 2 Processor
  The fast version of the Nios II processor is equipped with a hardware multiplier as well as data and instruction caches. Since the template does not contain an on-chip memory the caches are vital for system performance, since everything (data and instructions) is stored in SDRAM. However, you are free to change any settings of the processor and the system as a whole. Hence you can add additional memories for the processor (in this context also have a look at the so called tightly coupled memories) or deactivate unneeded features to free up resources for other parts of your design.
- JTAG UART
  The UART interface to communicate with the `nios2-terminal`.
- SDRAM Controller
  The SDRAM Controller enables your system to interface the 128 MB SDRAM of the DE2-115 Board. The bitwidth of this memory is 32 bits. The controller is configured to run at 100 MHz. Note that if you change the operation frequency of your system, the parameters of this core also need to be adjusted.
- Textmode Controller and Touchscreen
  To interface with your application the template provides a simple textmode controller. Note that this core stores the "framebuffer" internally, thus it does not put any load on the memory bandwidth of the SDRAM. The touchscreen controller enables the design to get inputs from the display.
- Audio & Audio and Video Config
  These cores implement the interface to the audio ADC/DAC. Everything is already setup for the required sampling frequency (48 kHz), hence there should not be made any changes to the settings of these cores.

Hint: This year's main task is quite demanding on the memory interface. Keep in mind that the FPGA board also contains 2 MB SRAM, which is quite simple to interface. The Platform Designer also contains an Avalon compatible SRAM controller.

1.2 Operation Modes

Your design should support two operation modes:

- real-time Mode:
  In this mode the system should constantly read samples from the input ADC, filter them and write them to the output DAC. This mode will be used to assess the performance of your filter (i.e. what is the maximum length of the impulse response that can be processed in real-time). After the real-time mode is activated, your system may take some time to setup and initialize the hardware and software components, read the impulse response from the SD card as well as perform some pre-calculation with if necessary. After this startup phase it should immediately proceed with the actual filter operation.
- Offline Mode:
  To verify the quality and accuracy of your of filter implementation, it shall be possible to read a file form the SD card, perform the filter operation and write it back to the SD card. Make sure that this operation mode uses the exact same hardware and software modules as the real-time mode (i.e. it is not allowed to use e.g. a slower but more accurate implementation for this task).

For both operation modes the impulse response file shall be loaded from the SD card (this can be a hard-coded file name in your program like `ir.wav`). The mode switch shall be possible during run-time (i.e. no recompilation), either through a `nios2-terminal` command, a user interface on the display or any other method you may come up with. For the offline mode you can either make the input selectable or simply read the file `input.wav`.

1.3 Reference Implementation

The template includes a reference implementation of a reverb algorithm in the form of an Octave/Matlab script, which can be started using the bash scipt located in the `tools` directory.

```
./reverb_reference INFILE OUTFILE IRFILE
```

This script loads two wave files (the impulse response and the actual input file) and applies the filter operation by processing fixed-size chunks using an FFT-based convolution based on the overlap-add method[2]. Recall that a convolution in the time-domain can be expressed as a multiplication in the frequency domain.[3] Consider the following Octave code, where a convolution of the signals `x` and `h` is performed in three different ways.

```
x = [1,2,3,4,5,6,7,8]
h = [1,2,1]

result1 = conv(x,h)
result2 = ifft(fft([x,zeros(1,length(h))]) .* fft([h,zeros(1,length(x))]))
result3 = filter(h,1,x)
```

All three methods yield the same result, the only difference is the length of the result vector. Notice that in order to perform the FFT-based convolution the input vectors must be zero-extended, such that they have the same length and their total length has (at least) the size of the result of the convolution (i.e. `length(x)+length(h)-1`).

The files created by the reference implementation will be used to assess your solution. To check your implementation against the reference the template provides the `checkwav.sh` script which basically calculates the correlation between two wave files. To achieve all points you have to reach a correlation coefficient of 0.9995. Note that if you use the same scale value for the reference scripts and for your own implementation the maximal sample deviation should be within one LSB. Note that the reference implementation normalizes the resulting samples to a value range of -1 to 1. However, this is not possible for the real-time filter so you may change this to a constant factor. Be aware that the Octave script uses a very naive and completely unoptimized algorithm! Its purpose is mainly to demonstrate the general idea behind the processing steps. Search for additional material (papers, signal processing books, etc.) for possible optimizations. Think about what can be precalculated.

Hint: Start out by understanding the algorithm and try to come up with an improved Octave filter. Then port your code to a fixed-point C version. To read and write wave files you can simply use the C module (`wav.[ch]`) located in the Nios II software directory. Also don't forget about the zero-delay requirement and plan for that feature early on in your development process!

[2]`https://en.wikipedia.org/wiki/Overlap-add_method`
[3]`https://en.wikipedia.org/wiki/Convolution#Discrete_convolution`

2 Build Environment

The build environment is basically the same as for the Get-To-Know task. Note that we will ONLY use the makefile-based build process when we check your submission. This means that a simple make in the project directory must successfully run the build process and download the hardware and software to the board. So keep in mind that if you change the build environment or use the Eclipse IDE, this "feature" must still be functioning! If you are working on your own computer make sure that the build works on the TI lab computers! However, during the implementation process of your solution you are of free to use the graphical interface of Quartus/Qsys as well as Eclipse for software development (this can especially be useful if you what to use the debugger).

3 Submission and Grading

The goal is to implement a real-time reverb filter with zero delay, good audio quality and a maximal number of taps. You may use all resources on the board and FPGA without restrictions. The task is divided into two parts:

Midterm presentations In the first part we ask you to gain a good understanding of the algorithm, the computational effort required for each part of the calculation and the memory requirements. Based on this, you should prepare architectural plans for your solution. You will present and we will discuss your profiling insights and solution concept in a lab appointment, and additionally you will make a slide show presentation to show them to the other groups.

As a group, make an appointment at hwsw@ecs.tuwien.ac.at for the presentation/discussion in the lab with all group members present. Before coming to the appointment, upload your slides for the public presentation in TUWEL. You do not need to show working HW/SW at this time.

You can get 35 points for the lab and public presentations based on the quality and feasibility of your concept, your expertise during the discussion and the quality of your presentation.

Final submission By the end of the term, you will submit your solution and present it in a lab appointment (like the get-to-know task). Again, make an appointment each group member can attend at hwsw@ecs.tuwien.ac.at and upload your solution to TUWEL 24 hours in advance.

The grading will be based on the achieved performance and the submitted solution quality. The target value for the number of (stereo) taps your filter should be able to handle is $t = 96000$ (i.e. 2 seconds). Based on the actual number x achieved by your solution you will then be awarded

$$max\Big(\frac{x}{t} * 30, 30\Big) + min\Big(\big(\frac{x}{t} - 1\big) * 12, 0\Big)$$

points, i.e. there are bonus points for solutions that exceed the target value. Note however, that the bonus point will only be awarded if you reach the quality criterion. For code and optimization quality, you can get a maximum of 20 points. If you don't reach the required quality (i.e. the correlation coefficient as determined by the `./checkwav.sh` script) 15 points will be deducted.

References

[1] Guillermo Garcia. Optimal filter partition for efficient convolution with short input/output delay. In *Audio Engineering Society Convention 113*, Oct 2002.

[2] W. G. Gardner. Efficient convolution without input-output delay. *J. Audio Eng. Soc.*, 43 (3):127–136, March 1995.