TECHNISCHE HOCHSCHULE DEGGENDORF

Department: Computer science

Course of study: Applied computer science

# Project report “Awesome Audio Mixer” on FPGA Acceleration

## Group 3

Harald Keller

Chetan Kandya

Submission: 29.06.2021

# Table of contents


**1 Project abstract ..... 3**

1.1 Motivation for project ..... 3

1.2 Overview of project ..... 3

**2 Fundamentals wave format and FIR filter ..... 4**

2.1 Wave audio format ..... 4

2.2 FIR filter ..... 5

2.3 FFT ..... 8

**3 Overlay AMIX (Audio Mixer) ..... 9**

3.1 AMIX IP ..... 9

3.2 CSIM ..... 10

3.3 Block diagram ..... 11

**4 Jupyter Notebook results ..... 12**

**5 Comparison PS versus PL ..... 14**

5.1 Runtime Comparison ..... 14

5.2 Speedup ..... 15

**6 Conclusion ..... 16**

# 1 Project abstract

## 1.1 Motivation for project

The motivation of the project is to compare a pure software implementation of an audio mixer algorithm with a combined software-hardware implementation using a FPGA board in terms of speedup.

## 1.2 Overview of project

In this project we have created an Audio Equalizer, which can be used to alter the properties (like bass and treble) of an audio file. To achieve this, the audio file is divided into three frequency ranges. These are then individually amplified and finally merged back into one file (cf. Figure 1).

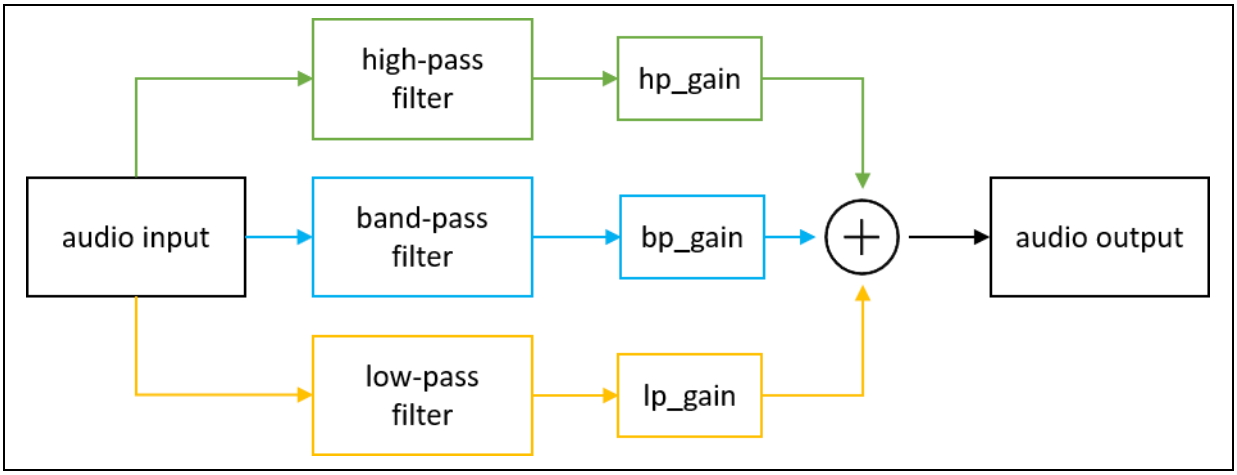



Figure 1: Overview audio mixer

# 2 Fundamentals wave format and FIR filter

## 2.1 Wave audio format

The wave format is used to store audio files digitally. It is the main format on Microsoft Windows to store uncompressed audio data. The format is organized in chunks (ct. Figure 2).

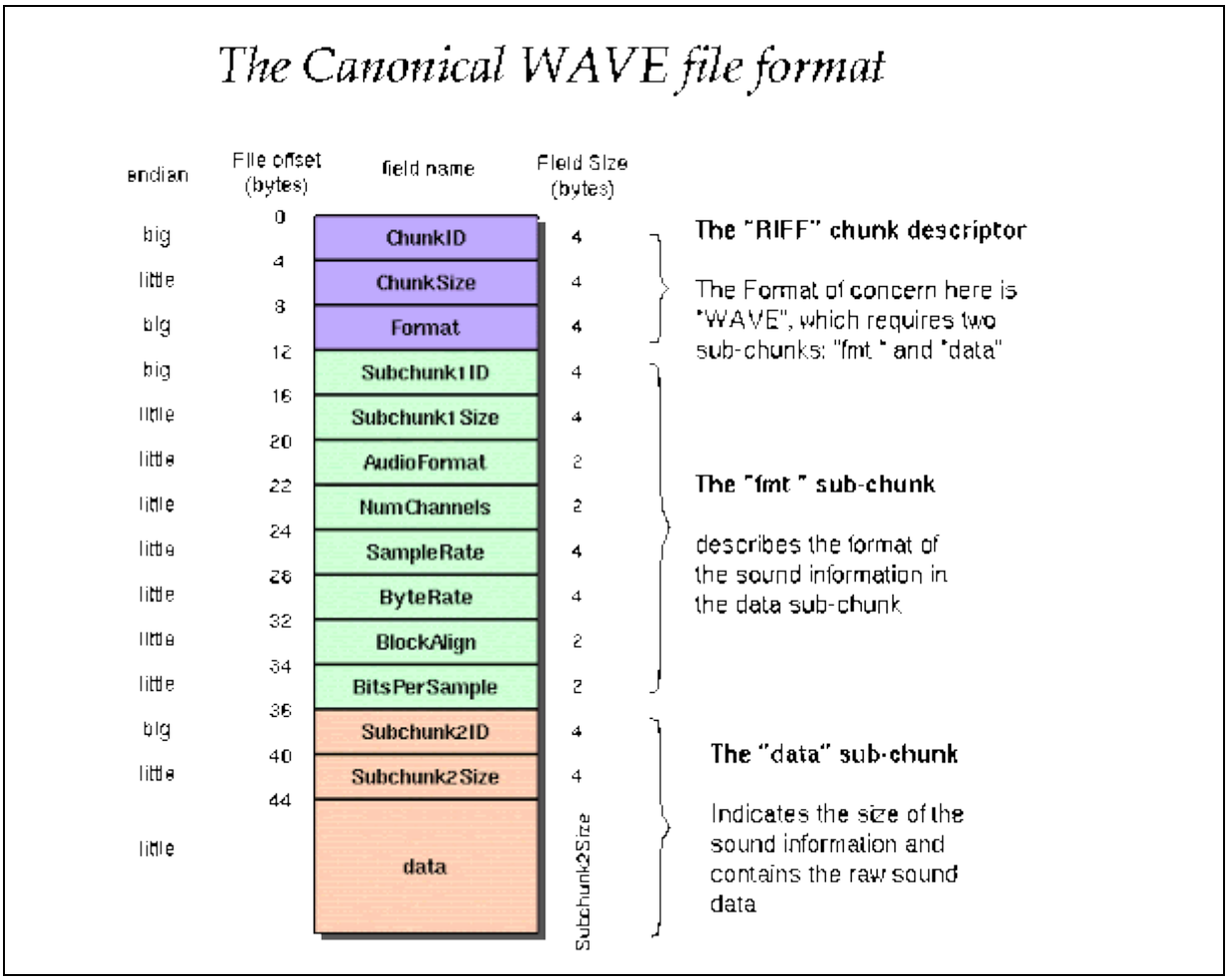



Figure 2: Wave format

Source: http://soundfile.sapp.org/doc/WaveFormat/

When a wave file is opened with a Hex editor (ct. Figure 3) the individual chunks become visible. Following can be seen by analyzing the bytes:

- ChunkID = „RIFF“ (char[4])
- Format = „WAVE“ (char[4])
- ChunkID = „fmt“ (char[4])

- AudioFormat = 0x0001 = 1 = PCM = (unit16, canonial uncompressed format)
- NumChannels = 0x0002 = 2 (uint16)
- SampleRate = 0x0000ac44 = 44100 (uint32, 44.1kHz)
- BitsPerSample = 0x0010 = 16 (uint16)
- Data = 0x49000000 = 1224736768 bytes

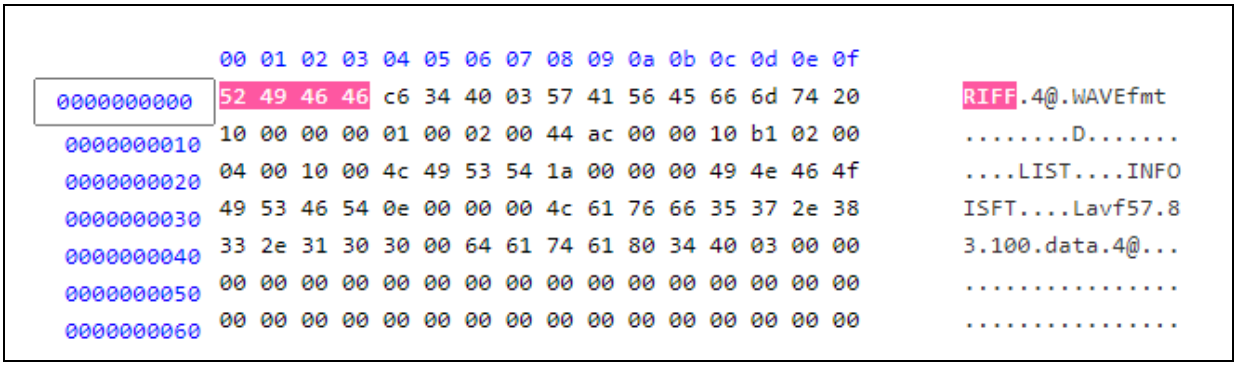



Figure 3: AC_DC - Hells Bells Wave file hexadecimal representation

It can be seen that the first samples are zero. It should be emphasized that the sample values of the two channels are always stored alternately (ct. Figure 4).

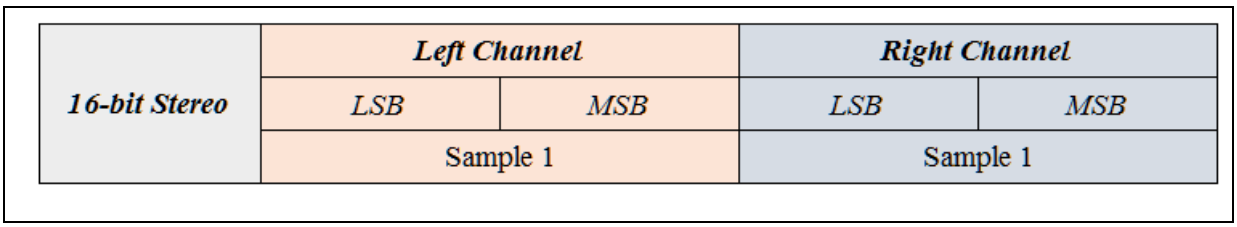



Figure 4: Samples arrangement 16 Bit stereo audio file in data section

## 2.2 FIR filter

An essential part of the project is to divide the piece of music into different frequency parts to be able to amplify them individually afterwards. A finite impulse response (FIR) filter is used for this purpose.

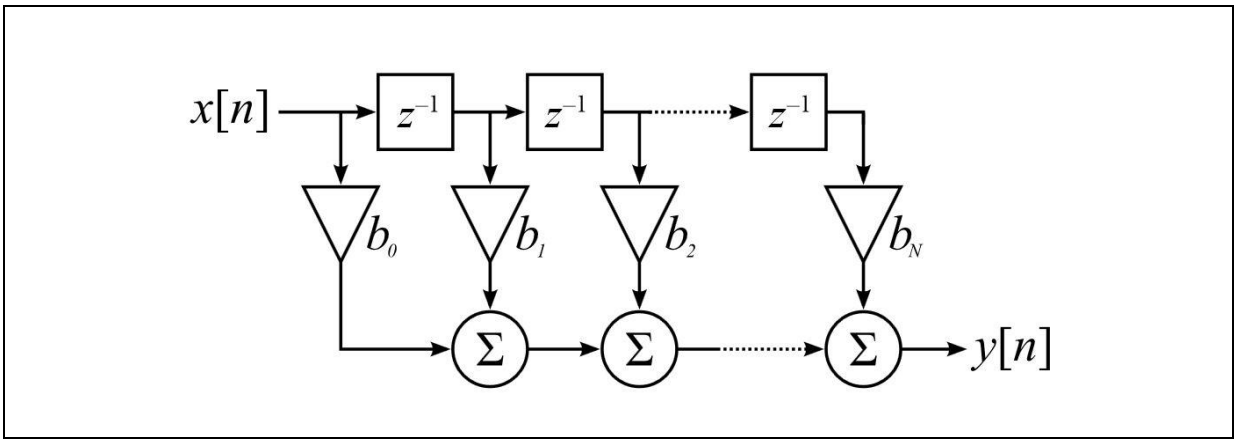



Figure 5: FIR filter

Source: https://www.all-electronics.de/elektronik-entwicklung/fir-filter-im-cpld.html

An FIR filter always has an order of N. The greater the order of the filter is selected, the steeper the transition range or the attenuation in the stopband is pronounced.

The upper range of the filter (cf. Figure 5) represents a delay chain with N+1 links. The input value x[n] is the n-th sample value of the audio file. To calculate the output value, successive audio values are pushed through the delay chain, multiplied by the coefficients $b_i$ and then accumulated.

A low-pass filter (0 Hz – 3 kHz), a band-pass filter (3 kHz – 10 kHz) and a high-pass filter (> 10 kHz) are used for the project. The upper limit of the high pass filter is limited by the human ear, which can only perceive frequencies up to about 20 kHz. Therefore, audio files usually have frequencies only up to 22 kHz. In combination with the Nyquist theorem, this results in the typical sampling rate of 44.1 kHz for audio files.

The filter designer from MATLAB is used to determine the $b_i$ coefficients. For all filters, the order of 50, the sample frequency of 44.1 kHz and hamming windows FIR filters are used.

The magnitude responses of the various filters are shown below. It can be seen that they have an attenuation of greater than 60 dB in the stopbands.

If the sample values are represented with signed 16-bit values as shown before, the following attenuation results for the highest possible value in the stopband:

$$a = (2^{16-1} - 1) * 10^{\frac{-60}{20}} = 32.767 = 32$$

Normally this value is no longer audible.

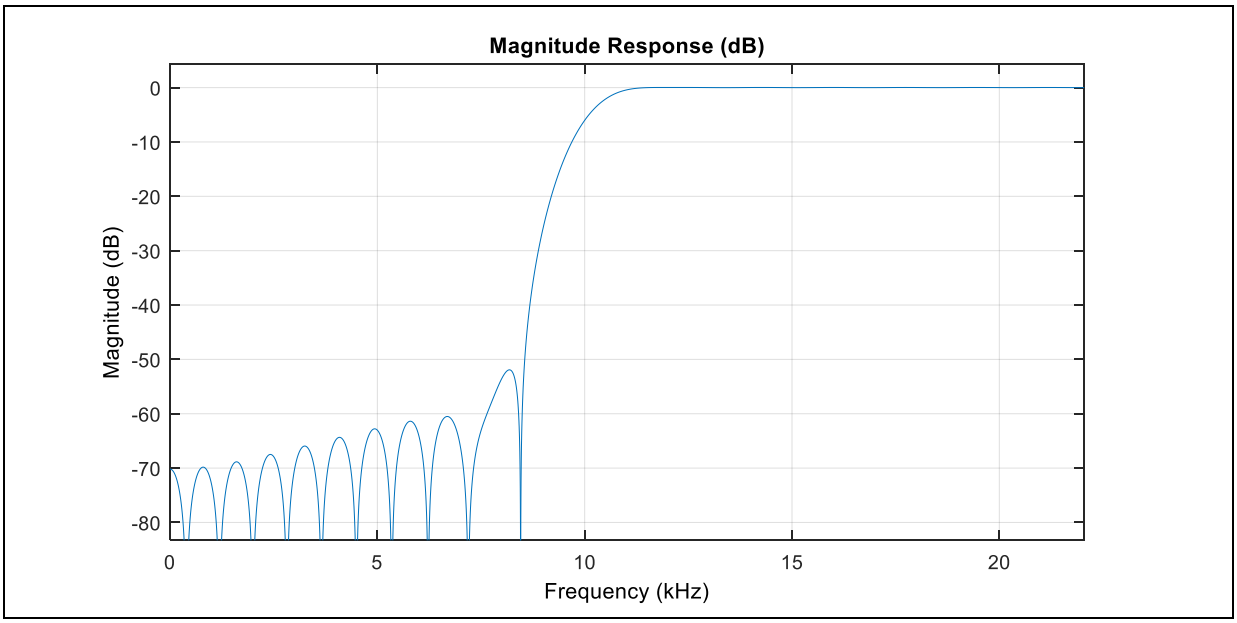



Figure 6: Low-pass filter

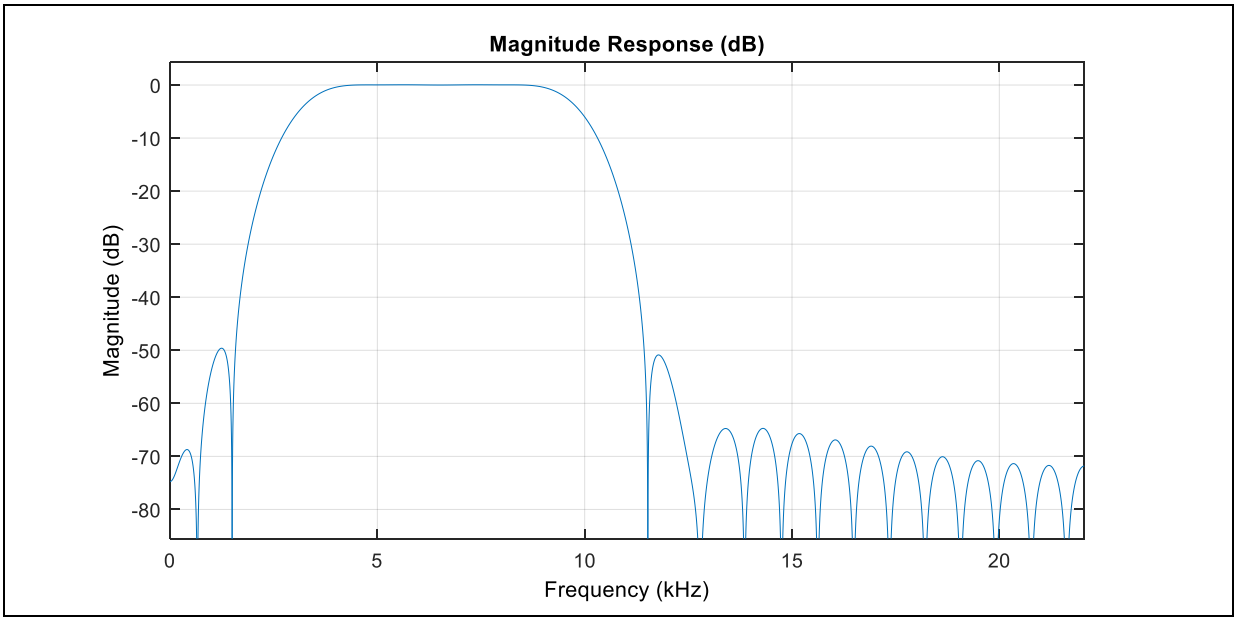



Figure 7: Band-pass filter

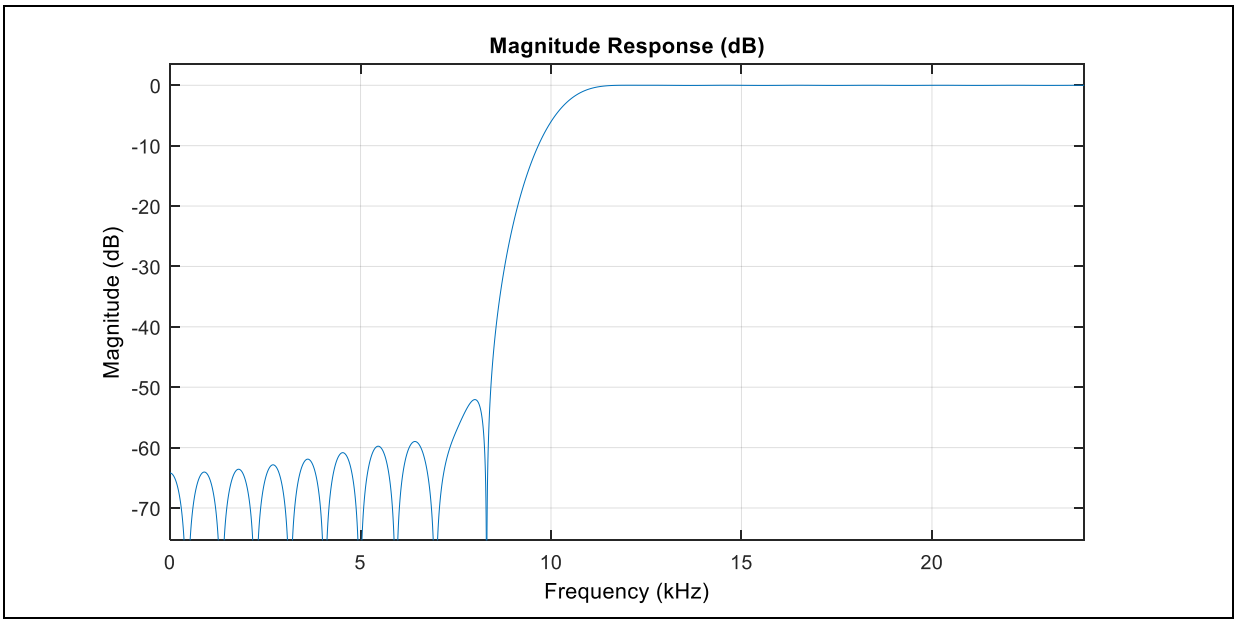



Figure 8: High-pass filter

## 2.3 FFT

The Fast-Fourier-Transformation (FFT) is an instrument to decompose a discrete-time signal into its frequency components. During the project, the FFT is used to evaluate the filter effect and to display the final result in the frequency domain. The result of FFT of the raw audio data are shown below.

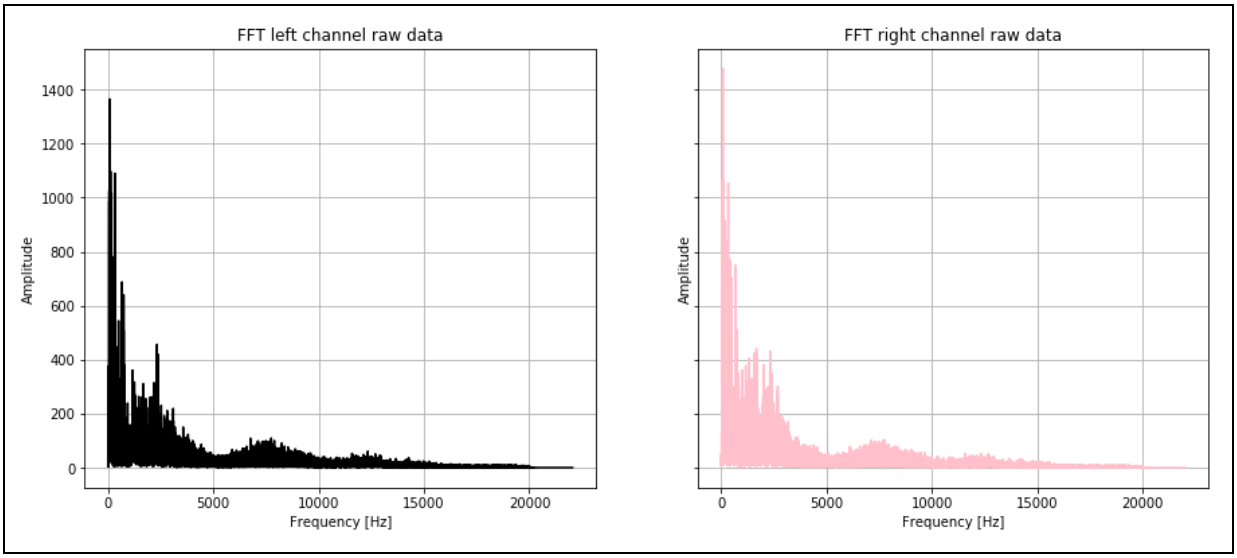



Figure 9: FFT raw audio data

# 3 Overlay AMIX (Audio Mixer)

For the combined software-hardware implementation the filtering and the multiplication with the amplification values will be attained by means of FPGA. For this purpose, an IP had to be created.

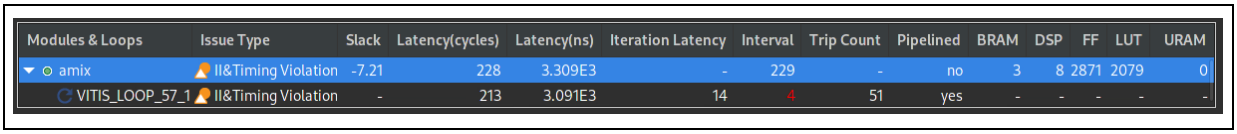
To achieve this, a C++ function is first created and then translated into VHDL code or an IP block using the HLS of Vivado Vitis. In the next section the results of the IP generation are summarized.

## 3.1 AMIX IP

The created function "amix" can be found in the file amix.cpp. In the following we will refer to it. The function takes a sample value for each call and returns a sample value. For the connection of the IP to the processing system only the AXI Lite bus is selected, because the inout values consist only of one floating point number. The AP_CTRL_HS interface is used as a block level I/O protocol of the IP.

The main part of the function consists of a For loop, in which the buffer for the filter is continuously updated. Furthermore the buffer values are multiplied with FIR coefficients and accumulated afterwards.

The compiler instruction “HLS PIPELINE” was used to realize the operations of the loop side by side. This also causes an unrolling of the for loop. By default the Initiation interval (II) is set to 1. However during the creation of the IP the II had to be set to 4. This causes that only every 4th clock cycle new input values are read. Consequently, one process takes 40 ns.



| Modules & Loops | Issue Type | Slack | Latency(cycles) | Latency(ns) | Iteration Latency | Interval | Trip Count | Pipelined | BRAM | DSP | FF | LUT | URAM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| amix | II&Timing Violation | -7.21 | 228 | 3.309E3 | - | 229 | - | no | 3 | 8 | 2871 | 2079 | 0 |
| VITIS_LOOP_57_1 | II&Timing Violation | - | 213 | 3.091E3 | 14 | 4 | 51 | yes | - | - | - | - | - |

Figure 10: Performance & Resource Estimates

The schedule viewer shows that first the gain values and the sample input value are read.

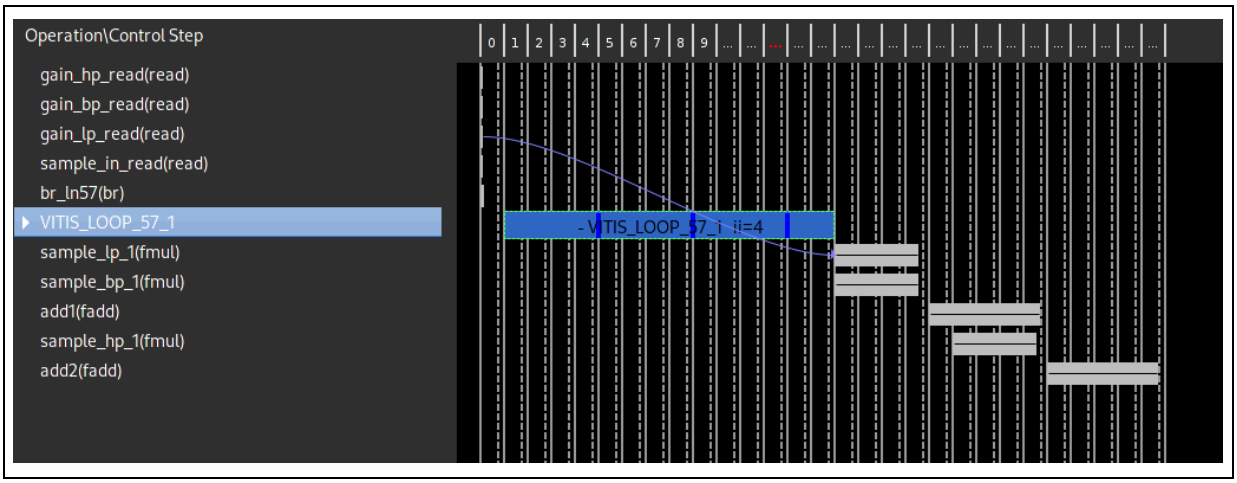



Figure 11: Schedule Viewer AMIX

In the VITIS_LOOP_57_1 it can be seen that a shift register was created from the buffer for the sample values and that this is "rotated" once.

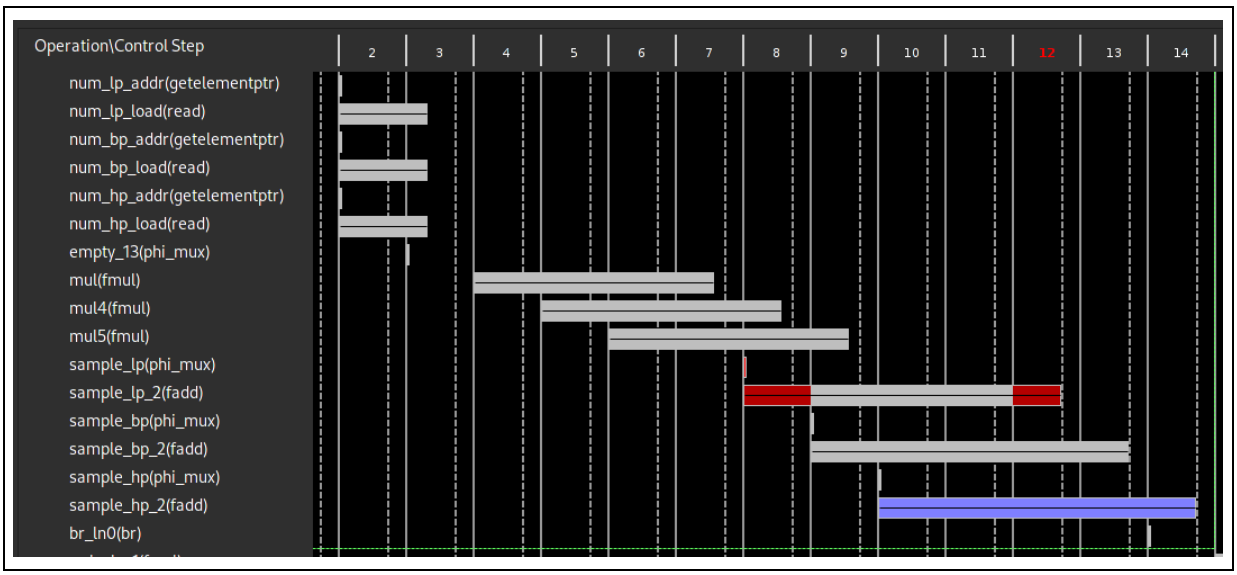



Figure 12: Cutout Schedule Viewer VITIS_LOOP_57_1

After that the numerator values are read, a floating-point multiplication is performed. At the end, the filtered samples are multiplied by the gain values and finally accumulated.

## 3.2 CSIM

Before RTL code is generated, the C function must be tested. This is done using the C simulation in Vitis HLS. The simulation of amix function can be found in the main.cpp file.

```
INFO: [SIM 2] ***************** CSIM start *****************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../main.cpp in debug mode
   Generating csim.exe
0.18674
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] ***************** CSIM finish *****************
```

Figure 13: Output C Simulation

The test returns the value 0.18674, which is calculated as follows:

$$sample_{out} = numlp[0] * sample_{in} * gain_{lp}$$

$$+numbp[0] * sample_{in} * gain_{bp}$$

$$+numhp[0] * sample_{in} * gain_{hp}$$

$$= -9.70552162e-04 * 100.25 * 1.00 + 8.10099084e-05 * 100.25 * 2.00$$

$$+ 8.90425301e-04 * 100.25 * 3.00 = 0.18674$$

## 3.3 Block diagram

Once the IP has been created, it can be linked to the processing system in the block diagram. This is done with the help of an AXI Interconnect component.

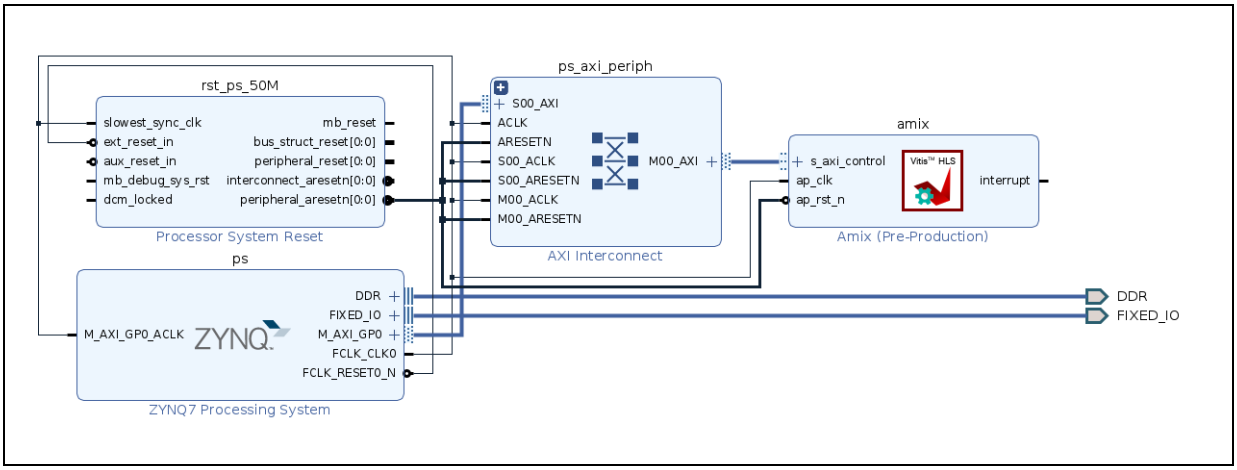



Figure 14: Block diagram

# 4 Jupyter Notebook results

In the notebook, the desired audio settings can first be adjusted with the help of sliders (ct. Figure 15). A slider setting of 100% causes no change. A value above 100% increases the range, while a value between 0 and 100% decreases it.

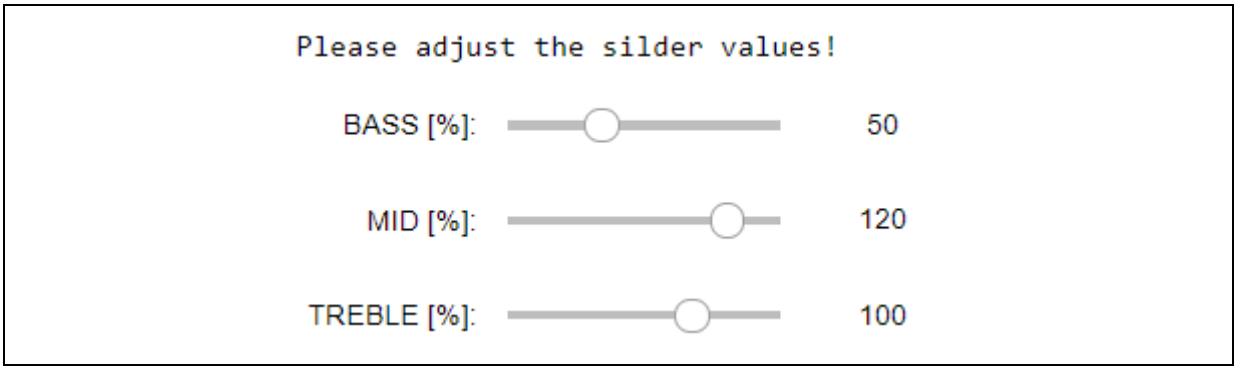



Figure 15: Audio adjustment using sliders

If the FFT is applied to the low-pass, band-pass and high-pass filtered values, the filtering effect becomes clear (ct. Figure 16). For example, the sample values of the low-pass filter only contain frequencies up to about 3 kHz.

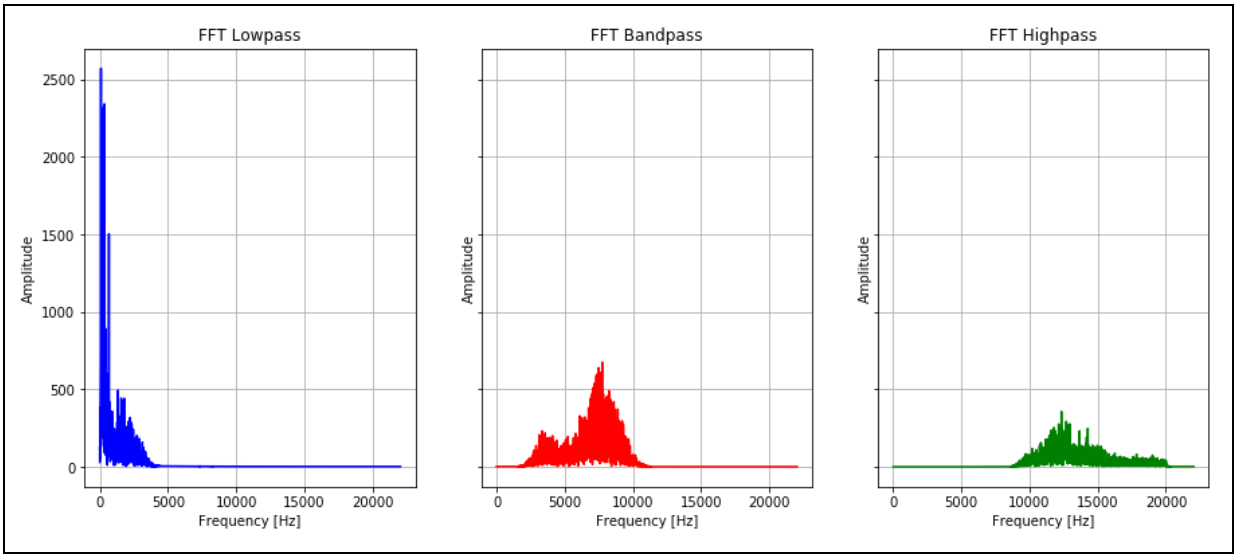



Figure 16: FFT filtered samples

If the output values of the two solutions are compared, no difference is noticeable (Tryout: raw song). Furthermore, it is noticeable that the amplitude values within the bass range have halved, the values in the midrange have increased somewhat and the values in the treble range have remained the same. This confirms the effect of the amplification values (Tryout: mixed song).

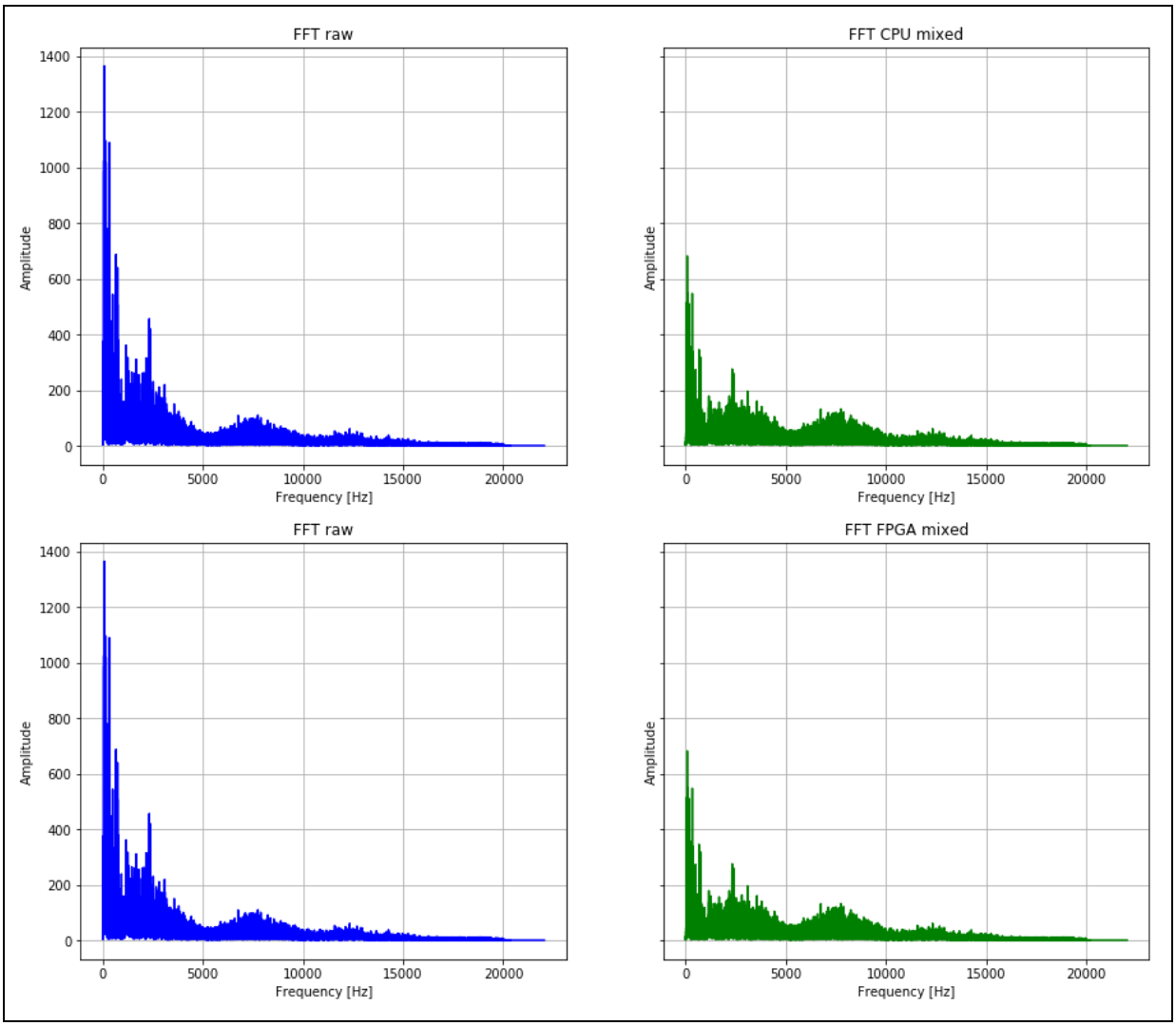



Figure 17: Output comparison of two solutions

# 5 Comparison PS versus PL

## 5.1 Runtime Comparison

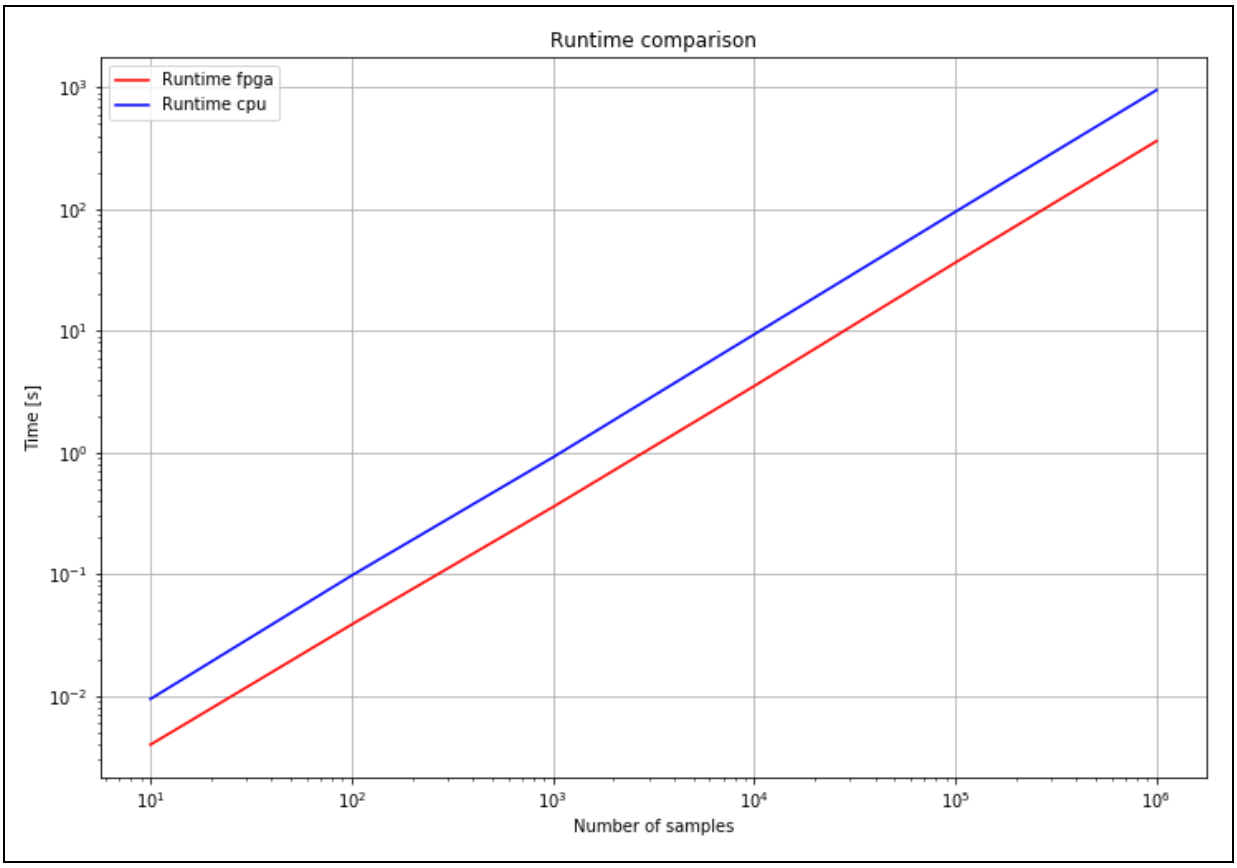



Figure 18: Runtime comparison

The above graph is plotted on a logarithmic scale. It is evident that the runtime and the number of processed samples are in a linear relationship. This can be justified by the fact that the runtime for a sample value is approximately the same. It is also visible that the solution with the use of the FPGA achieves significantly faster runtimes. For the processing of 10000 sample values the PS solution needs about 10 seconds, while the PL solution needs only about 3 seconds.

## 5.2 Speedup

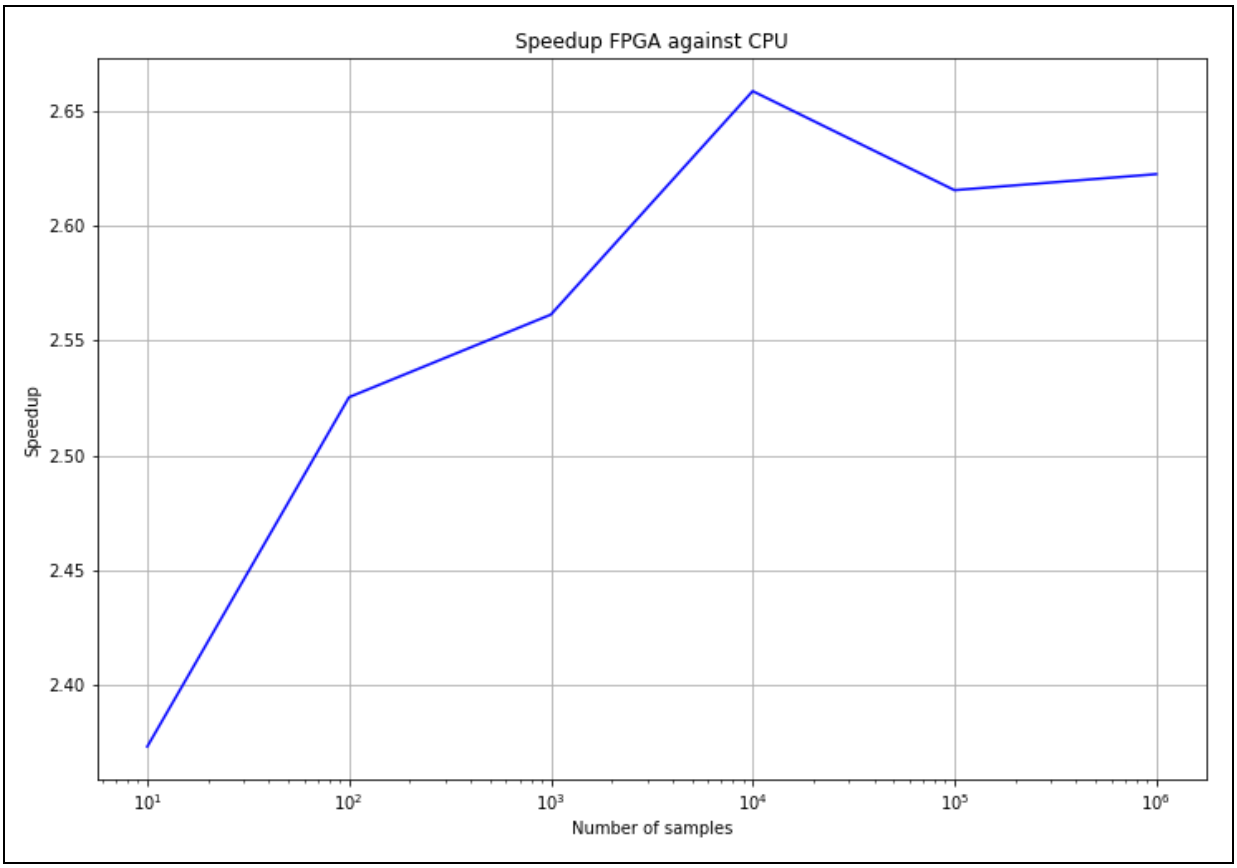



Figure 19: Speedup PS and PL

The Speedup is calculated by dividing the CPU runtime by FPGA runtime, for the same number of samples. It is visible that the FPGA speedup is also increasing as the number of samples increases. Noticeable is the increased speedup at 10000 sample values, which cannot be justified at first glance.

# 6 Conclusion

The FPGA is very useful if you must buffer elements and do independent operations on each buffer element (as with FIR filters).

For a limited number of samples, we saw a significant speedup in the execution of the algorithm. This might not be case every time, but there are various real-world implementations for using FPGA acceleration.

As an outlook, another AMIX IP blocks could be created in the block diagram, for example to process multiple audio channels simultaneously.