University of Pisa

IIR Filter – Report

Master Degree in
Embedded Computing Systems

Project for the Course of
Digital Systems Design

A.Y. 2016/2017

Baris Gabriele — 506321

Contents

1 Introduction

1.1 Requirements

You are asked to design an IIR filter for audio applications, following the difference equation:

$$y[n] = y[n-1] - \frac{1}{4}x[n] + \frac{1}{4}x[n-4] \tag{1}$$

This filter allows to reduce the input signal components at half of the sampling frequency.

Refere to a possible application using a 16-bit WAV file.

1.2 Digital filters

In signal processing, a digital filter is a system that performs mathematical operations on a sampled, discrete-time signal to reduce or enhance certain aspects of that signal. This is in contrast to the other major type of electronic filter, the analog filter, which is an electronic circuit operating on continuous-time analog signals.

A digital filter system usually consists of an Analog-to-Digital Converter to sample the input signal, followed by a microprocessor and some peripheral components such as memory to store data and filter coefficients etc. Finally a Digital-to-Analog Converter to complete the output stage. Program Instructions (software) running on the microprocessor implement the digital filter by performing the necessary mathematical operations on the numbers received from the ADC. In some high performance applications, an FPGA (as in this project) or ASIC is used instead of a general purpose microprocessor, or a specialized DSP with specific paralleled architecture for expediting operations such as filtering.

Digital filters may be more expensive than an equivalent analog filter due to their increased complexity, but they make practical many designs that are impractical or impossible as analog filters. Digital filters are not subject to the component non-linearities that greatly complicate the design of analog filters. Analog filters consist of imperfect electronic components, whose values are specified to a limit tolerance and which may also change with temperature and drift with time. As the order of an analog filter increases, and thus its component count, the effect of variable component errors is greatly magnified. In digital filters, the coefficient values are stored in computer memory, making them far more stable and predictable.

Because the coefficients of digital filters are definite, they can be used to achieve much more complex and selective designs – specifically with digital filters, one can achieve a lower passband ripple, faster transition, and higher stopband attenuation than is practical with analog filters. Even if the design could be achieved using analog filters, the engineering cost of designing an equivalent digital filter would likely be much lower. Furthermore, one can readily modify the coefficients of a digital filter to make an adaptive filter or a user-controllable parametric filter. While these techniques are possible in an analog filter, they are again considerably more difficult.

When used in the context of real-time analog systems, digital filters sometimes have problematic latency (the difference in time between the input and the response) due to the associated Analog-to-Digital and Digital-to-Analog conversions and anti-aliasing filters, or due to other delays in their implementation.

Digital filters are commonplace and an essential element of everyday electronics such as radios and cellphones.

Characterization

A digital filter is characterized by its transfer function, or equivalently, its difference equation. Mathematical analysis of the transfer function can describe how it will respond to any input. As such, designing a filter consists of developing specifications appropriate to the problem (for example, a second-order low pass filter with a specific cut-off frequency), and then producing a transfer function which meets the specifications.

The transfer function for a linear, time-invariant, digital filter can be ex-

pressed as a transfer function in the *Z-domain*; if it is causal, then it has the form:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_N z^{-N}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_M z^{-M}}$$

where the order of the filter is the greater of N or M.

This is the form for a recursive filter, which typically leads to an IIR (or Infinite Impulse Response) behaviour, but if the denominator is made equal to unity (i.e. no feedback), then this becomes a FIR (or Finite Impulse Response) filter.

The impulse response, often denoted as $h[k]$ or h_k, is a measurement of how a filter will respond to the Kronecker delta function. For example, given a difference equation, one would set $x_0 = 1$ and $x_k = 0$ for $k \neq 0$ and evaluate.

The impulse response is a characterization of the filter's behaviour. Digital filters are typically considered in two categories: Infinite Impulse Response (IIR) and Finite Impulse Response (FIR). In the case of linear time-invariant FIR filters, the impulse response is exactly equal to the sequence of filter coefficients:

$$y_n = \sum_{k=0}^{N} h_k x_{n-k}$$

IIR filters, on the other hand, are recursive, with the output depending on both current and previous inputs as well as previous outputs. The general form of an IIR filter is thus:

$$\sum_{m=0}^{M} a_m y_{n-m} = \sum_{k=0}^{N} b_k x_{n-k}$$

Plotting the impulse response will reveal how a filter will respond to a sudden, momentary disturbance.

Realization

After a filter is designed, it must be realized by developing a signal flow diagram that describes the filter in terms of operations on sample sequences.

A given transfer function may be realized in many ways. In the same way, all realizations may be seen as "factorizations" of the same transfer function, but different realizations will have different numerical properties. Specifically, some realizations are more efficient in terms of the number of operations or storage elements required for their implementation, and others provide advantages such as improved numerical stability and reduced round-off error. Some structures are better for fixed-point arithmetic and others may be better for floating-point arithmetic.

Direct Form I A straightforward approach for IIR filter realization is Direct Form I, where the difference equation is evaluated directly. This form is practical

for small filters, but may be inefficient and impractical (numerically unstable) for complex designs. In general, this form requires $2N$ delay elements (for both input and output signals) for a filter of order N.

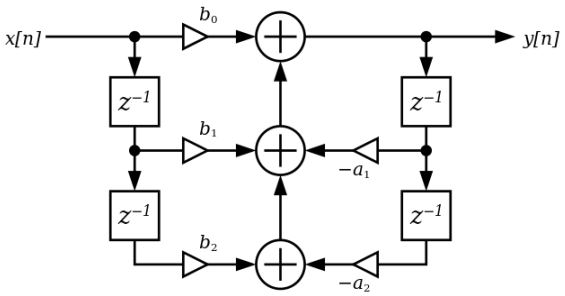


Figure 1: Filter realization in Direct Form I

Direct Form II The alternate Direct Form II only needs N delay units, where N is the order of the filter – potentially half as much as Direct Form I. This structure is obtained by reversing the order of the numerator and denominator sections of Direct Form I, since they are in fact two linear systems, and the commutativity property applies. Then, one will notice that there are two columns of delays that tap off the center net, and these can be combined since they are redundant, yielding the implementation as shown below.

The disadvantage is that direct form II increases the possibility of arithmetic overflow for filters of high Q or resonance. It has been shown that as Q increases, the round-off noise of both direct form topologies increases without bounds. This is because, conceptually, the signal is first passed through an all-pole filter (which normally boosts gain at the resonant frequencies) before the result of that is saturated, then passed through an all-zero filter (which often attenuates much of what the all-pole half amplifies).

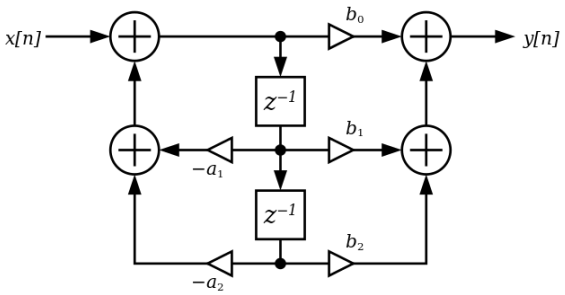


Figure 2: Filter realization in Direct Form II

Cascaded Second-Order Sections A common strategy is to realize a higher-order (greater than 2) digital filter as a cascaded series of Second-Order "biquadratric" (or "biquad") Sections. The advantage of this strategy is that the coefficient range is limited. Cascading Direct Form II sections results in N delay elements for filters of order N. Cascading Direct Form I sections results in $N + 2$ delay elements, since the delay elements of the input of any section (except the first section) are redundant with the delay elements of the output of the preceding section.

1.3 The WAV audio format

The Waveform Audio File Format (WAVE, or more commonly known as WAV due to its filename extension) is a Microsoft and IBM audio file format standard for storing an audio bitstream on PCs.

Though a WAV file can contain compressed audio, the most common WAV audio format is uncompressed audio in the Linear Pulse-Code Modulation (LPCM) format. LPCM is also the standard audio coding format for audio CDs, which store two-channel LPCM audio sampled 44,100 times per second with 16 bits per sample. Since LPCM is uncompressed and retains all of the samples of an audio track, professional users or audio experts may use the WAV format with LPCM audio for maximum audio quality. WAV files can also be edited and manipulated with relative ease using software.

In addition to sampled data, the WAV file has also an header containing some useful information such has the sampling frequency, the number of channels, and so on. It can also be enhanced with metadata using Extensible Metadata Platform (XMP) data or ID3 tags.

The WAV format is limited to files that are less than 4 GiB, because of its use of a 32-bit unsigned integer to record the file size header.

1.4 Linear Pulse-Code Modulation

Pulse-code modulation (PCM) is a method used to digitally represent sampled analog signals. It is the standard form of digital audio in computers, compact discs, digital telephony and other digital audio applications. In a PCM stream, the amplitude of the analog signal is sampled regularly at uniform intervals, and each sample is quantized to the nearest value within a range of digital steps.

Linear pulse-code modulation (LPCM) is a specific type of PCM where the quantization levels are linearly uniform. This is in contrast to PCM encodings where quantization levels vary as a function of amplitude. Though PCM is a more general term, it is often used to describe data encoded as LPCM.

A PCM stream has two basic properties that determine the stream's fidelity to the original analog signal: the sampling rate, which is the number of times per second that samples are taken; and the bit depth, which determines the number of possible digital values that can be used to represent each sample.

Modulation

In Figure 3, a sine wave (red curve) is sampled and quantized for PCM. The sine wave is sampled at regular intervals, shown as vertical lines. For each sample, one of the available values (on the y-axis) is chosen by some algorithm. This produces a fully discrete representation of the input signal (blue points) that can be easily encoded as digital data for storage or manipulation.

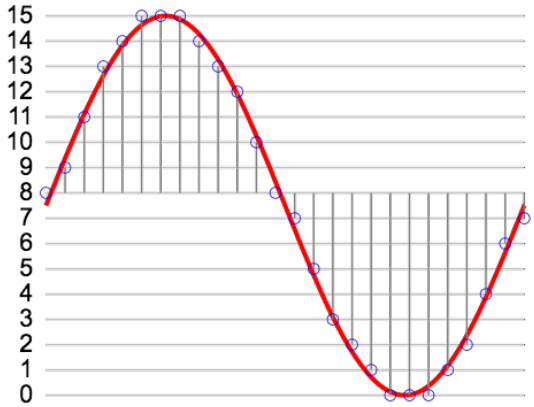


Figure 3: Sampling and quantization of a signal for 4-bit LPCM

For the previous example, we can verify that the quantized values at the sampling moments are 8, 9, 11, 13, 14, 15, 15, 15, 14, etc. Encoding these values as binary numbers would result in the following set of nibbles: 1000, 1001, 1011, 1101, 1110, 1111, 1111, 1111, 1110, etc. These digital values could then be further processed or analyzed by a Digital Signal Processor (DSP).

Several PCM streams could also be multiplexed into a larger aggregate data stream, generally for transmission of multiple streams over a single physical link. One technique is called Time-Division Multiplexing (TDM) and is widely used, notably in the modern public telephone system.

The PCM process is commonly implemented on a single integrated circuit generally referred to as an Analog-to-Digital Converter (ADC).

Demodulation

To recover the original signal from the sampled data, a "demodulator" can apply the procedure of modulation in reverse. After each sampling period, the demodulator reads the next value and shifts the output signal to the new value. As a result of these transitions, the signal has a significant amount of high-frequency energy caused by aliasing. To remove these undesirable frequencies and leave the original signal, the demodulator passes the signal through analog filters that suppress energy outside the expected frequency range (greater than the Nyquist frequency $\frac{f_s}{2}$).

The sampling theorem shows PCM devices can operate without introducing distortions within their designed frequency bands if they provide a sampling frequency twice that of the input signal. For example, in telephony, the usable voice frequency band ranges from approximately 300 Hz to 3400 Hz. Therefore, per the Nyquist–Shannon sampling theorem, the sampling frequency (8 kHz) must be at least twice the voice frequency (4 kHz) for effective reconstruction of the voice signal.

The electronics involved in producing an accurate analog signal from the discrete data are similar to those used for generating the digital signal. These devices are Digital-to-Analog Converters (DACs). They produce a voltage or current (depending on type) that represents the value presented on their digital inputs. This output would then generally be filtered and amplified for effective use.

1.5 Project structure

The project folder has the following structure:

```
IIR
├── matlab ..... MATLAB design files and folders
├── report ..... LaTeX report files and folders
├── db
│   ├── src ..... VHDL source files
│   └── tb ..... VHDL testbench files
├── active_hdl ..... Active-HDL project folder
└── vivado_synth ..... Vivado project folder
```

Also, the software versions used for this project are the following:

- MATLAB R2017a
- Active-HDL Student Edition
- Vivado HLx Edition 2017.1

2 Design

2.1 Filter

Starting from Equation 1 the discrete time transfer function has been derived as

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\frac{1}{4}z^{-4} - \frac{1}{4}}{1 - z^{-1}} \tag{2}$$

Then the filter has been designed using the software MATLAB and the file `Design.m`.

It has been decided to use the Direct Form II architecture since the filter is not so complex and it allows to use half of the registers required by Direct Form I.

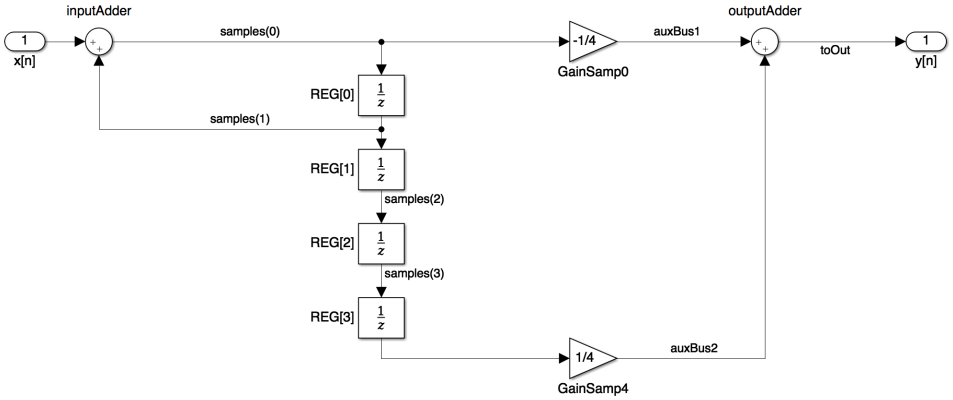


Figure 4: Schematic representation of the IIR Filter architecture

The transfer function in Equation 2 has been used to implement a fixed point discrete time filter in MATLAB.

It has been decided to specify the precision for all the components inside the filter in order to have complete control over the synthesis. Both the input and the output are over 16 bits, as said by the requirements; instead, the internal connections are over 18 bits in order to deal with the bit-growth.

Then the filter has been checked using the `fvtool()` function, which has provided the Magnitude Response and the Impulse Response rispectively depicted in Figure 5 and Figure 6.

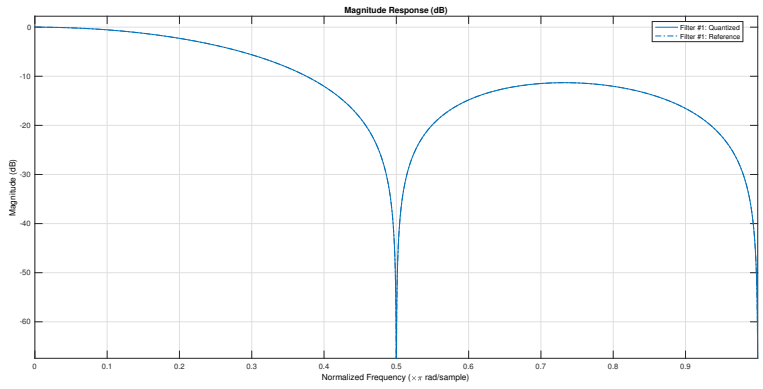


Figure 5: Magnitude Response obtained by the `fvtool()` MATLAB function

```
% Starting from the difference equation of the filter,
% it's possible to compute
% the transfer function as
% tf = (1/4 z^-4 -1/4)/(1 - z^-1)
% so the coefficients of the numerator and denominator are
num_coeff = [-1/4 0 0 0 1/4];
den_coeff = [1 -1];
% this filter can be used using the filter() function

% Designing the filter in fixed point aritmetic
iir = dfilt.df2;
set(iir, 'arithmetic', 'fixed', ...
    'OutputMode', 'SpecifyPrecision', ...
    'Numerator', num_coeff, ...
    'Denominator', den_coeff, ...
    'InputWordLength',16, ...
    'InputFracLength', 0, ...
    'OutputWordLength', 16, ...
    'OutputFracLength', 2, ...
    'StateWordLength', 16, ...
    'StateFracLength', 0, ...
    'ProductMode', 'SpecifyPrecision', ...
    'ProductWordLength', 18, ...
    'NumProdFracLength', 2, ...
    'DenProdFracLength', 2, ...
    'AccumMode', 'SpecifyPrecision', ...
    'AccumWordLength', 18, ...
    'NumAccumFracLength', 2, ...
    'DenAccumFracLength', 2, ...
    'CastBeforeSum', false);
```

Listing 1: Fixed point arithmetic filter in MATLAB

2.2 Test data

The second part of the `Design.m` script is used to filter two WAV mono files and build up the data that later will be used for the testbench.

Also, in order to test the VHDL implementation of the filter, the first 1000 samples of both the original and the filtered audio signals are stored as strings into a text file inside the `test` folder. For more details see Subsection 3.2.

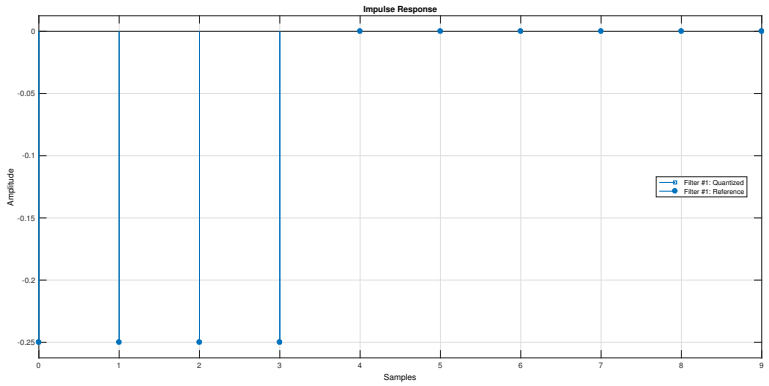


Figure 6: Impulse Response obtained by the `fvtool()` MATLAB function

```
%% Test with audio signals

% Read the audio samples from the WAV file as int16 array
[synth_original, Fs_synth] = audioread('wav/Synthesizer.wav', 'native');
[street_original, Fs_street] = audioread('wav/Street.wav', 'native');

% Convert the int16 audio samples into 16 bit signed fixed values
synth_fix = fi(synth_original, true, 16, 0);
street_fix = fi(street_original, true, 16, 0);

% Filter the audio signal
synth_filtered_fix = filter(iir, synth_fix);
street_filtered_fix = filter(iir, street_fix);

% Save the filtered audio in a WAV file
audiowrite('wav/SynthesizerFiltered.wav', int16(synth_filtered_fix),...
        Fs_synth);
audiowrite('wav/StreetFiltered.wav', int16(street_filtered_fix),...
        Fs_street);


%% Save data for testbenches in txt files

buildTxtFiles('synth', synth_fix, synth_filtered_fix);
buildTxtFiles('street', synth_fix, synth_filtered_fix);
```

Listing 2: Script used to read the WAV file filter it and save the result into the corresponding WAV file

3 Implementation

3.1 Filter

The VHDL implementation of the filter is inside the file `IIR.vhd`. It has been built up using 4 registers, a ripple carry adder and a ripple carry adder-subtractor.

The definition of the entity is shown in Listing 3. For more details look at the file `IIR.vhd`.

```
entity IIR is
        generic (Nbit : positive := 8);
        port (
                clk : in std_ulogic; -- clock
                rst_l : in std_ulogic; -- reset (active low)
                x : in std_ulogic_vector(Nbit-1 downto 0); -- input
                y : out std_ulogic_vector(Nbit-1 downto 0) -- output
        );
end IIR;
```

Listing 3: IIR filter entity definition

It has been implemented using a generic dimension for more flexibility, even if in this project it was requested to work in 16 bits values.

Given N the size (in bits), the input adder is a ripple carry adder over N bits and its carry out is neglected (left `open`).

The multiplication by the coefficients is hard coded using signals because it's just (neglecting the sign) a division by 4, which can be efficiently computed by an arithmetic right shift of 2 bits. Another strategy would have been to use registers for the coefficients, a dedicated line to read them from the outside and store them, and many multipliers and adders. This would have provided a much more flexible structure for the filter (the coefficients could have been changed at runtime) but for this project it was not requested and it would have been a useless complication.

After the shift, both the signals are over $N + 2$ bits and they go into an adder-subtractor: this allows to save one adder (and so reduce complexity). In fact, theoretically speaking, two different adders would be needed: one for `-1/4*samples(0)`, which is computed as `1/4*samples(0)` and then 2 complemented (and an adder is needed since $cpl2(x) = not(x) + 1$) and another one for the output sum.

Instead, using just a single adder-subtractor, it's possible to save complexity and perform the same operation. In fact

```
toOut = not(auxBus1) + auxBus2 + 1 =
        = auxBus2 - auxBus1 =
  = 1/4*samples(4) - 1/4samples(0)
```

The output of the sum is still on $N + 2$ bits: at this point, the N least significant bits are taken to obtain the actual output.

3.2 Testbench

Testbenches aim to check if the filter works as expected: it this case they are used in order to verify if the VHDL implementation conforms to the MATLAB design. The verification is performed by taking arrays of inputs and expected outputs from MATLAB and then by using assertions in the VHDL simulator in order to highlight possibile discrepancies: if no error message is shown, then the designed filter and the implementd one have the same behavior.

The test has been done providing inputs and expected output as arrays of samples and feeding the filter with a new sample on each clock period.

Impulse Response The first testbench aims to check if the impulse response is as expected. To do that, a discrete time impulse is obtained as an array of 5 samples

$$delta = [1\ 0\ 0\ 0\ 0]$$

A snippet of the verification code used for the testbenches is shown in Listing 4. Since all the testbenches have this structure, no other code is presented in this report. For a more details look at the code inside the `db/tb/` folder.

```
architecture TB_Arch of IIR_TB_Delta is

        [...]

begin
        [...]
        -- stimuli
        driver_p: process

        variable delta_in : WAV_IN := ("0000000000000001",
        "0000000000000000", "0000000000000000",
        "0000000000000000", "0000000000000000");

        variable expected_resp : WAV_IN := ("1111111111111111",
        "1111111111111111", "1111111111111111",
        "1111111111111111", "0000000000000000");
begin

        -- inital reset of the filter
        rst_l <= '0';
        wait until clk'event and clk='1';
        rst_l <= '1';

        -- loop over all the samples
        for i in 0 to SAMPLES-1 loop

                sample <= delta_in(i);
                expected <= expected_resp(i);


                wait until clk'event and clk='1';

                -- If the actual output and the expected output
                -- mismatch, raise an asserion
                assert (output = expected)
                report "Mismatch for index i = " & integer'image(i)
                severity error;

        end loop;

        enable <= '0';
        end process;

end TB_Arch;
```

Listing 4: Snippet of the testbench verification through assertions

As shown in Figure 7, the two responses are exactly the same.

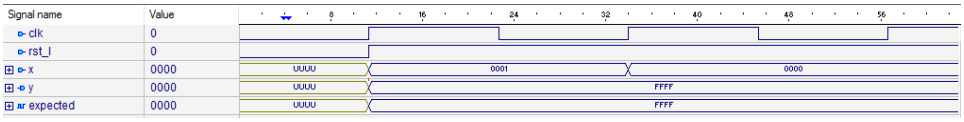
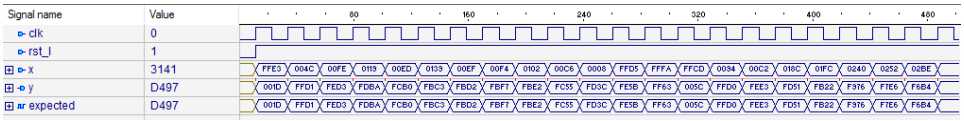


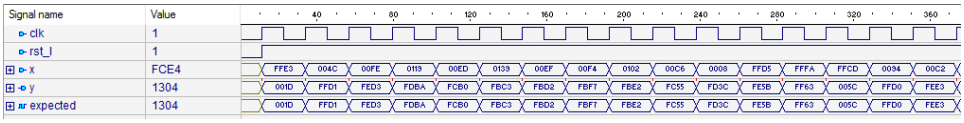
Figure 7: Active-HDL testbench for impulse response

WAV samples In order to have a reliable testing phase, several input files of several natures should be tested, in order to see the behavior of the filter with respect to the audible spectrum (ranging from 20 Hz to 20 kHz). For simplicity it has been decided to test just two WAV files. Also, since they have thousands of samples, only the first 1000 samples have been tested.

The two files are `Street.wav` and `Synthesizer.wav` and for each of them a different VHDL file has been provided. The output of the simulator is depicted in Figure 8.



(a) Synthesizer



(b) Street

Figure 8: Active-HDL testbenches for WAV files

4 Synthesis

Before proceeding with the synthesis, a wrapper for the filter is needed, since it has been designed using a `generic` dimension for more flexibility: because of that, the `IIRWrapper.vhd` does nothing more than setting the dimension of the filter to be a 16-bit IIR filter.

Then, the syntesis has been performed using the Vivado software by *Xilinx* and addressing the `xc7z010clg400-3`, which is the FPGA on the *ZyBo board*.

4.1 RTL Analysis

The first step was to obtain the RTL Netlist through the RTL Analysis. Provided a filter object with 18 inputs and 16 outputs (34 I/O ports) and containing 4 Registers, a RippleCarryAdder and a RippleCarryAdderSubtractor (Figure 10).

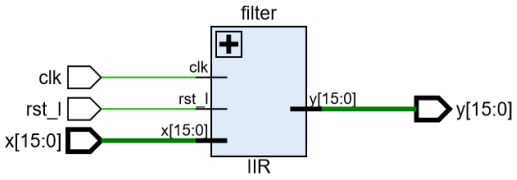


Figure 9: Filter Wrapper

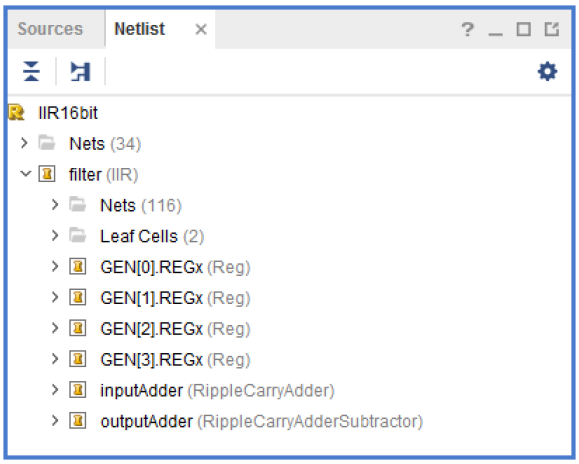


Figure 10: RTL Netlist

4.2 Synthesis

Since it was not asked to generate the bitstream for the board, the synthesis has been performed without the IO planning, but imposing just the clock constraint.

The filter must work at 44100 Hz, so the clock period has been chosen

$$t_{ck} = \frac{1}{44100} \simeq 22676\,ns$$

which provides a clock frequency of

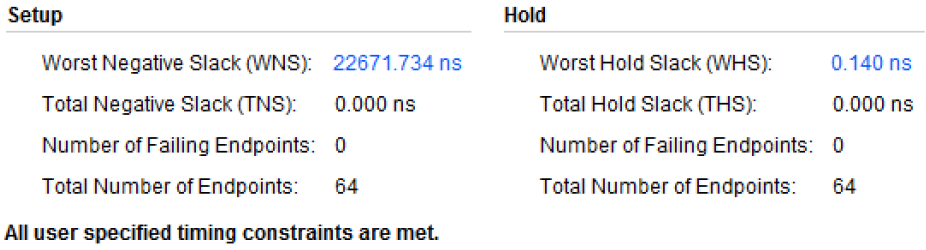
$$f_{ck} = \frac{1}{22676 \times 10^{-9}} \simeq 44099,48\,Hz$$

that causes a relative error of about 0.001%.

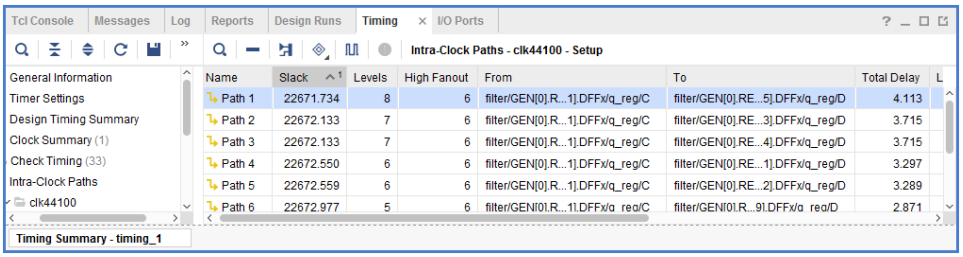
The process terminates correctly giving no errors or warnings. The main information is depicted in Figure 11.

Running the syntesis with this constraint only, we obtain a clock with a period of 22675,998 ns. At this point, the *Worst Negative Slack* (WNS) is 22671,734 ns. Given this information, the maximum theoretical frequency which the circuit could properly work at is about

$$f_{max} = \frac{1}{(22675,998 - 22671,734) \times 10^{-9}} \simeq 234\,MHz$$



(a) Timing Report



(b) Worst Negative Slack

Figure 11: Timing Information after Synthesis

The critical path is given by Path 1 and goes from GEN[0].REGx/GEN[1].DFFx to GEN[0].REGx/GEN[15].DFFx and it's due to the carry chain inside the input RippleCarryAdder.

4.3 Implementation

The implementation is a fundamental step because, after the *Place and Route*, the tool is able to compute the actual delays due to the interconnection network.

The process terminates correctly giving no errors or warnings. The main information is depicted in Figure 12.

After the implementation, the *Worst Negative Slack* (WNS) is 22671,021 ns and the maximum theoretical frequency which the circuit could properly work at is about

$$f_{max} = \frac{1}{(22675,998 - 22671,021) \times 10^{-9}} \simeq 200\,MHz$$

The critical path is given by Path 1 and goes from GEN[0].REGx/GEN[2].DFFx to GEN[0].REGx/GEN[14].DFFx, which in the path on the input RippleCarryAdder.

A view of the FPGA after the implementation is shown in Figure 13: as you can see from the light blue regions, only very few gates are used.

Setup

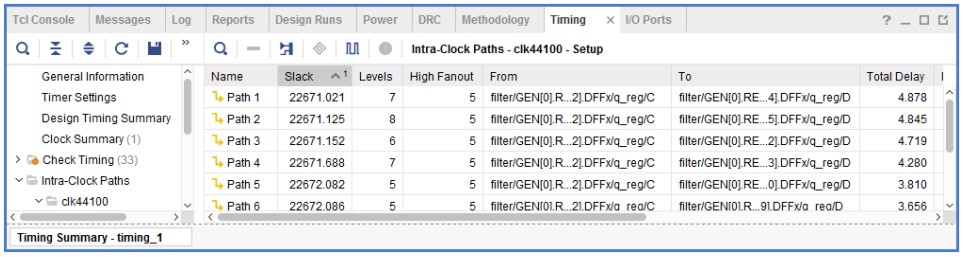
Worst Negative Slack (WNS):	22671.021 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	64

Hold

Worst Hold Slack (WHS):	0.161 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	64

All user specified timing constraints are met.

(a) Timing Report



Name	Slack	Levels	High Fanout	From	To	Total Delay
Path 1	22671.021	7	5	filter/GEN[0].R...2].DFFx/q_reg/C	filter/GEN[0].RE...4].DFFx/q_reg/D	4.878
Path 2	22671.125	8	5	filter/GEN[0].R...2].DFFx/q_reg/C	filter/GEN[0].RE...5].DFFx/q_reg/D	4.845
Path 3	22671.152	6	5	filter/GEN[0].R...2].DFFx/q_reg/C	filter/GEN[0].RE...2].DFFx/q_reg/D	4.719
Path 4	22671.688	7	5	filter/GEN[0].R...2].DFFx/q_reg/C	filter/GEN[0].RE...3].DFFx/q_reg/D	4.280
Path 5	22672.082	5	5	filter/GEN[0].R...2].DFFx/q_reg/C	filter/GEN[0].RE...0].DFFx/q_reg/D	3.810
Path 6	22672.086	5	5	filter/GEN[0].R...2].DFFx/q_reg/C	filter/GEN[0].R...9].DFFx/q_reg/D	3.656

(b) Worst Negative Slack

Figure 12: Timing Information after Implementation

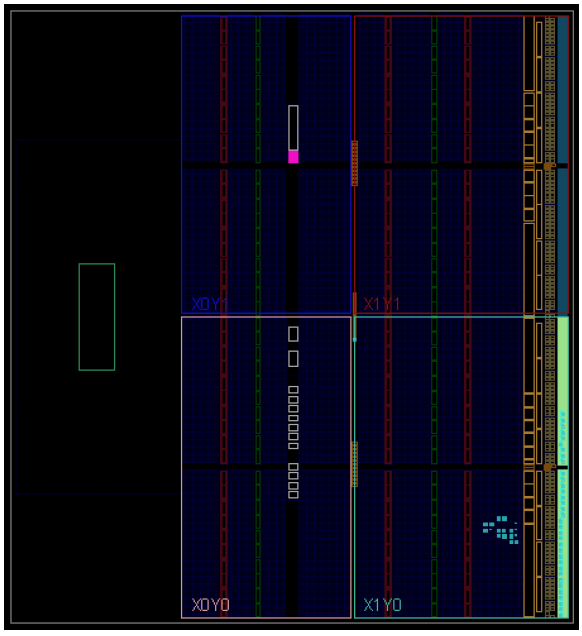


Figure 13: FPGA content after implementation

5 Conclusions

Let's first say that, even if both the synthesis and the implementation provided no errors, the DRC Violations section contains some warnings. They are due to:

- We are not using the ARM Cortex microcontroller available on the *ZyBo board*
- We are not properly setting the input and output pins, since as we have already said, it's not requested for this project

In a real project you can easily ignoring the first one if you are just using the FPGA, while you would get rid of the second one by properly configuring all the IO ports.

There are two interesting reports to look at: the Utilization Report and the Power Report.

5.1 Utilization Report

The utilization in terms of available resources is summarized in Table 1.

Resource	Utilization	Available	Utilization %
LUT	45	17600	0.26
FF	64	35200	0.18
IO	34	100	34.00

Table 1: Resource utilized by the syntesis

5.2 Power Report

In order to have a credible estimation about the power consumption of the circuit, the implementation has been done again using a 3.3V logic. The summary is shown in Figure 14.

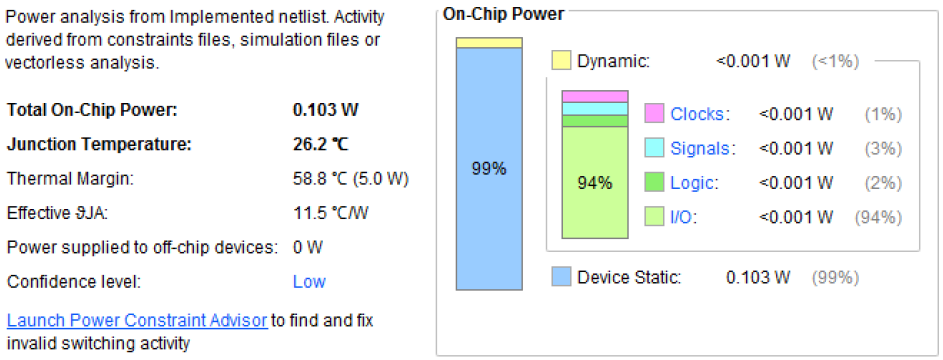


Figure 14: Power Report Summary

References

[1] Course Slides, Fanucci L., Meoni G., Pilato L., University of Pisa, 2017

[2] Digital Filter, *Wikipedia: The Free Encyclopedia*, `https://en.wikipedia.org/wiki/Digital_filter`

[3] WAV Audio Format, *Wikipedia: The Free Encyclopedia*, `https://en.wikipedia.org/wiki/WAV`

[4] PCM Encodign, *Wikipedia: The Free Encyclopedia*, `https://en.wikipedia.org/wiki/Pulse-code_modulation`

[5] Adder-Subtractor, *Wikipedia: The Free Encyclopedia*, `https://en.wikipedia.org/wiki/Adder-subtractor`