Table of Contents

[Educational Objective 2](#_Toc476837851)

[Technical Objective 2](#_Toc476837852)

[Deliverables 2](#_Toc476837853)

[Part 1 – Create Quartus Project With Timing Constraints 2](#_Toc476837854)

[Part 2 – Measure the sample rate of the CODEC. 2](#_Toc476837855)

[Part 3 – Experiment With Different Audio Waveshapes 3](#_Toc476837856)

[Part 4 – Write VHDL Implementation of Digital Filter 3](#_Toc476837857)

[Filter Specification 3](#_Toc476837858)

[How to Implement Filter 5](#_Toc476837859)

[Part 5 – Verify Your Digital Filter VHDL 6](#_Toc476837860)

[Part 6 – Implement the Digital Filter in Hardware 6](#_Toc476837861)

# Educational Objective

In this lab, you will learn how to…

1. Use fixed point **DSP** resources of the FPGA.
2. Write VHDL code for a **digital filter**.
3. Use **ModelSim** to verify HDL code.
4. Design and code a test-bench, and perform functional **verification** of your code.
5. Gain some familiarity with using Verilog, which is a commonly used language similar to VHDL.
6. Add basic **timing constraints** to your Quartus project.
7. Use the **dual port ram** buffer to write a new waveform to the audio waveRAM created in lab 4.
8. Perform a hardware **validation** of your design.

# Technical Objective

There are six main technical objectives in this assignment

1. Experiment with changing the waveform of the audio signal.
2. Measure the CODEC sample rate using digital oscilloscope.
3. Implement a low pass digital filter in VHDL.
4. Verify your filter design using a test-bench in VHDL.
5. Add basic timing constraints in your Quartus project.
6. Validate your filter design in hardware, by listening to and comparing the sound of a sequence of tones before and after filtering.

# Deliverables

* VHDL Code for Digital Filter
* VHDL Test Bench
* All appropriate sign-offs.
* Demo of working code with Digital Filter

# Part 1 – Create Quartus Project With Timing Constraints

In this section you will create a Quartus project for lab 5, which is essentially a clone of the project for lab 4. However, before modifying this, you will add some basic timing constraints. Please read the document ug\_tq\_tutorial, which will explain timing constraints, and how to implement them using TimeQuest, from Quartus. You will need to tell the tools that your clock rate is 50MHz. We will talk more about timing closure and pipelining later in the semester, after the mid-term.

Verify that your code still functions as it did for the lab 4 project.

# Part 2 – Measure the sample rate of the CODEC.

In order to measure the sample rate, you will route the AUD\_DACLRCK signal to an FPGA pin, connected to one of the 40 pin GPIO headers on the DE1. Using the digital oscilloscope you will measure the frequency of AUD\_DACLRCK.

In the reference material for this lab, you will find the data sheet for the audio codec, as well as documentation for the DE1 board. You will need the DE1 documentation to locate a pin on the one of the 40 pin GPIO headers. You should examine the CODEC data sheet to determine what the DACLRCK signal does.

You are basically routing an input pin (from the CODEC) directly to an output pin (connected to one of the 40 pin headers). In order to do so, you should use an intermediate signal, for example:

aud\_daclrck\_sig <= AUD\_DACLRCK;

GPIO\_1(0) <= aud\_daclrck\_sig;

# Part 3 – Experiment With Different Audio Waveshapes

For this part of the lab, you will write a different wave shape to the waveRAM, and listen to the different sound created by the new wave-shape. You can generate the waveform data through any means you choose. You will then need to write c-code to send 256 audio samples to the waveRAM, which you created in the previous lab. This can be done algorithmically, for example generating a ramp (triangular) wave function would be fairly easy. Or, for an arbitrary waveform, you will need 256 lines of c-code, to download the entire thing. You can also generate random numbers to be sent to the waveRAM, and this should produce roughly white noise at your speakers.

Each sample in the waveRAM is 32 bits wide, 16 bits for each of the two channels. The 16 bit channel data is in signed two’s complement format​.

# Part 4 – Write VHDL Implementation of Digital Filter

## Filter Specification

The low pass filter was designed using Matlab. The filter design is documented in Figure 1 through Figure 5, including filter coefficients, block diagram, and the required port map for the digital filter VHDL entity. Part of your task is to figure out how to implement this filter using fixed point arithmetic in the FPGA.

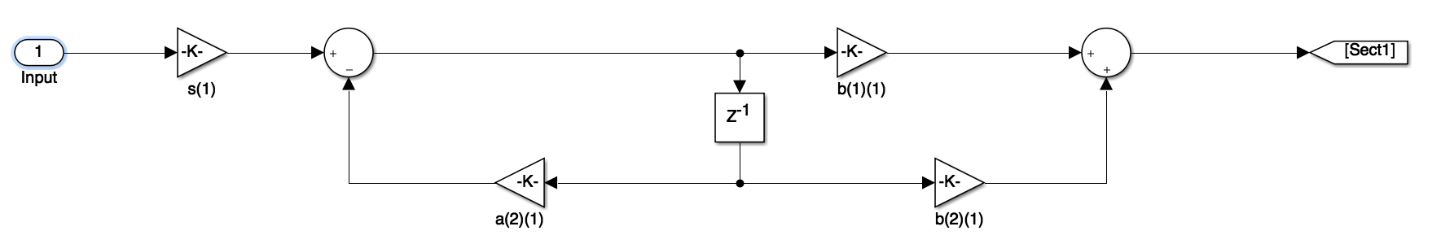


Figure 1: First Stage

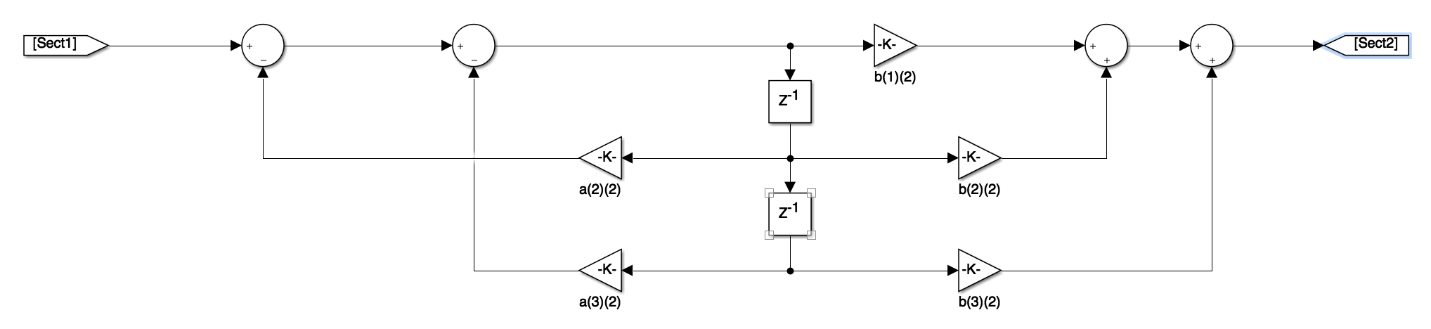


Figure 2: Second Stage

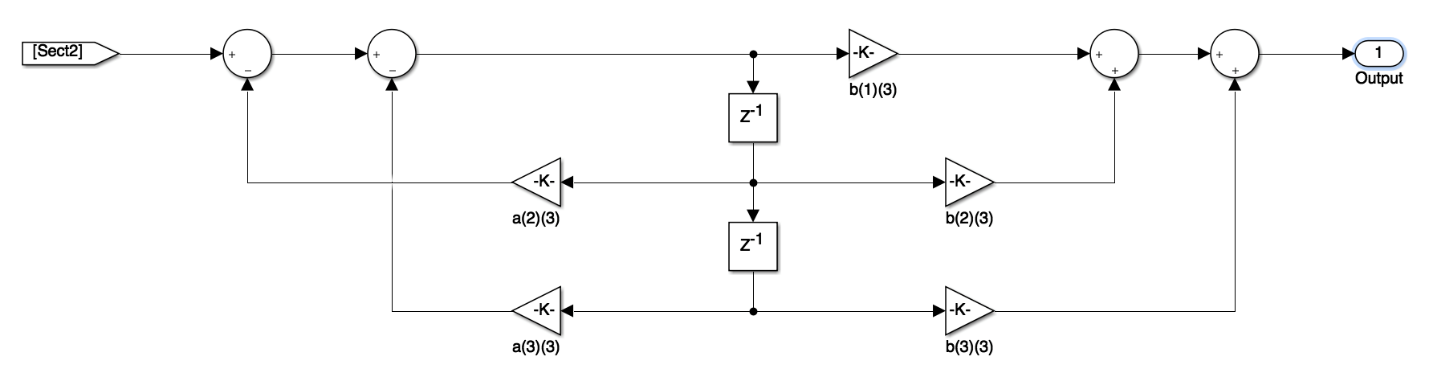


Figure 3: Third Stage

|  |  |  |
| --- | --- | --- |
| First Stage | b(1)(1) | 0.0033507 |
|  | b(2)(1) | 0.0033507 |
|  | b(3)(1) | 0 |
|  | a(2)(1) | -0.91 |
|  | a(3)(1) | 0 |
| Second Stage | b(1)(2) | 0.0026446 |
|  | b(2)(2) | 0.0052893 |
|  | b(3)(2) | 0.0026446 |
|  | a(2)(2) | -1.9349 |
|  | a(3)(2) | 0.94353 |
| Third Stage | b(1)(3) | 0.25008 |
|  | b(2)(3) | 0.50015 |
|  | b(3)(3) | 0.25008 |
|  | a(2)(3) | -1.8504 |
|  | a(3)(3) | 0.85861 |

Figure 4: Filter Coefficients

entity audio\_filter is

port (

i\_clk\_50 : in std\_logic;

i\_reset : in std\_logic;

i\_audioSample : in std\_logic\_vector(31 downto 0);

i\_dataReq : in std\_logic;

o\_audioSampleFiltered : out std\_logic\_vector(31 downto 0)

);

end entity;

Figure 5: VHDL Entity Declaration

* **i\_audioSample** is the audio sample, with 16 bits of left channel and 16 bits of right channel. Each channel is signed 16 bit number, representing inputs from -1 to +1. This is the data that comes from your waveRAM.
* **i\_dataReq** is a positive going pulse that comes from the DAC on every sample period. On the rising edge of this pulse you should update the filter output. Put another way, the filter calculation is done once for every pulse on the i\_dataReq signal.
* **o\_audioSampleFiltered** is the output of your digital filter, also 16 bits of left and 16 bits of right channel, and each one is 16 signed representing numbers between -1 and +1. For the purposes of this lab, we are only using one channel, so the two 16 bit samples are duplicated. You can therefore grab just 1/2 of the i\_audioSample (upper or lower 16 bits), as the input to your filter. Similarly, you can duplicate the output of your filter to fill the upper and lower 16 bit samples in o\_audioSampleFiltered.

## How to Implement Filter

Remember that the calculation of the filter output happens once per audio sample (NOT on every 50MHz Clock cycle) Following is an explanation of all the blocks in the filter block diagrams, and how you can implement them.

* The **circles** are adders. They all have one output, which is the sum of all the inputs. If an input is indicated with a negative sign, then it is subtracted instead of added. Be careful, because we are working with signed numbers, you must have the appropriate libraries at the top of your VHDL code.

use IEEE.std\_logic\_1164.all;

use ieee.std\_logic\_signed.all;

use ieee.std\_logic\_arith.all;

* The **triangles** are multipliers. We will be working with 36 bit signed numbers, so we need to account for…
  + The result of the multiplication is 72 bits.
  + The result of the multiplier must also be a signed number.

For our Quartus project, we will create a multiplier IP, (similar to the way we created the dual port RAM in the last lab). We will then instantiate multiple instances of this component, one for each multiplier.

For our ModelSim simulation, you will be given a module that performs this signed multiplication.

* The **z-1** blocks are delays. These are like d-ff’s, that clock the input data to the output once per audio sample. So for example, if our audio sample rate is 44kHz, then the audio sample period is 1/44kHz = 23us. The **i\_audioSample** signal is a one clock wide pulse (i.e. one 50MHz clock), which occurs once per audio sample. You will use this to implement the z-1 delays.

if (rising\_edge(clk\_50MHz)) then

if (i\_audioSample = ‘1’) then

delay\_out <= delay\_in;

end if;

end if;

# Part 5 – Verify Your Digital Filter VHDL

In this part of the lab, you will write a test bench to verify your design from part 3. You will execute this test bench using ModelSim. I have included two example test-benches, along with some information on doing file I/O, in a VHDL test bench, (see the lab 5 folder on MyCourses). These are only examples demonstrating the techniques, they are NOT intended to be something easily modified to create your test-bench.

In order verify your design, you will read a waveform from a file, process it through your filter, and output the filtered waveform to a file. The file containing the input waveform is included in the lab 5 folder on MyCourses, and is called one\_cycle\_200\_8k.

In order to verify functionality, you will plot your output waveform, and verify that it shows the appropriate response based on the filter transfer function.

In order to guide you in writing this test bench, you are given a test bench written in Verilog, and your task is…

* Figure out how to implement this test bench in VHDL. In the process of translating from Verilog to VHDL you will learn how the test bench works, and also gain some familiarity with Verilog, which is an equally common language used for digital design and verification.
* Run the test bench on your filter module, and use ModelSim to debug your code.
* If there are bugs in your code, (syntax err), use ModelSim to debug the design.

# Part 6 – Implement the Digital Filter in Hardware

In this part of the lab, you will add the digital filter to your design, and validate its performance on the actual hardware. In order to do this validation, you will use switch **SW0** to select either the original tone, or the filtered tone to be sent to the speakers or headphones. As you step through the frequencies, using **KEY1** you should be able to hear that the higher frequency tones are attenuated relative to the lower frequency tones.