Contents

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

Fixed Point Math

In a fixed point representation, we use binary numbers with an implied decimal point. In this example we will use signed 36 bit numbers, with 17 bits after the implied decimal. To convert a floating point number to a signed 36 bit number, with 17 bits after the implied decimal, you simply multiply by 2^{17}, round the result to nearest integer, and convert the integer to binary or hex. For example, using Excel…

```
=DEC2HEX(ROUND(A1*2^17,0))
```

The number may be positive or negative. If it is negative, be sure to include all the 1's in the MSbits.

For example, one of the coefficients of our digital filter will be -0.91 which can be translated to a VHDL contant as…

```
constant a21_const : signed(35 downto 0) := x"FFFFE2E14";
```

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 and Timequest found on MyCourses, 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.

Before running the Timequest Analyzer you should disable Signal Tap from the project. To disable Signal Tap click on Assignments > Settings and then Signal Tap II Logic Analyzer. In the pop-up box, uncheck the box next to "Enable Signal Tap II Logic Analyzer"

In TimeQuest, you will look at the following reports, and record the measurements on the appropriate entries in your sign-off sheet for this lab. As we progress through this project and future labs, you will look at these numbers again, to verify that your new designs are able to run at the required 50MHz clock rate. In the Tasks Pain of TimeQuest, run the following:

- **Reports -> Slack -> Report Setup Summary**, Record the Slack, which should be a positive number (i.e. black text)
- **Reports-> Datasheet -> Report Fmax Summary**, Record the maximum clock rate your design can run at.

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 figuring out how to implement this filter using fixed point arithmetic in the FPGA.

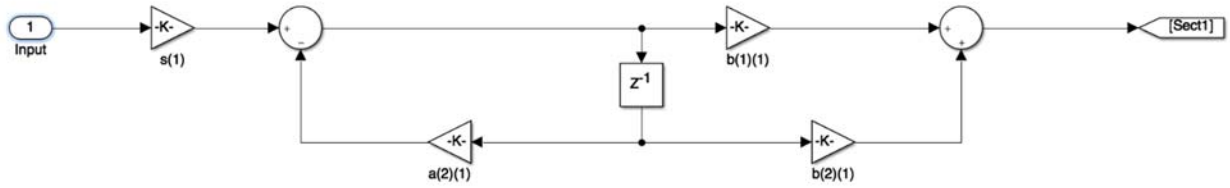

Figure 1: First Stage

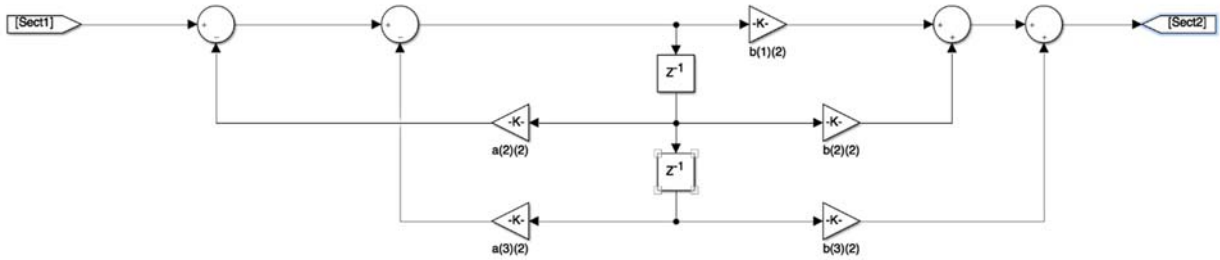

Figure 2: Second Stage

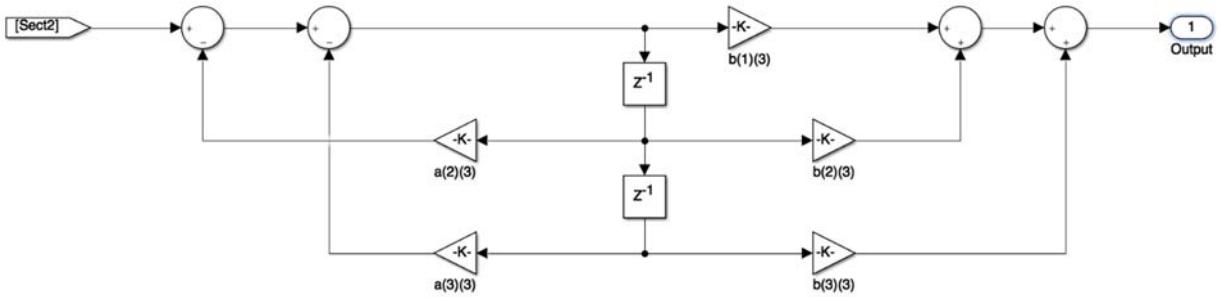

Figure 3: Third Stage

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 signed(31 downto 0);
    i_dataReq : in std_logic;
    o_audioSampleFiltered : out signed(31 downto 0)
  );
end entity;
```

Figure 5: VHDL Entity Declaration

The next section, “How to Implement Filter”, provides more detail. Below is just a brief description of the required ports.

- **i_audioSample** is the audio sample, with 16 bits of left channel and 16 bits of right channel. Each channel is a 16 bit signed fixed point number with 15 bits after the implied decimal point. This is the data that comes from your waveRAM. 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. You also need to deal with the conversion from 16 bit signed fixed point with 15 bits after the implied decimal, to 36 bit, signed fixed point with 17 bits after the implied decimal.
- **i_dataReq** is a positive going, one 50 MHz clock wide pulse, that comes from the DAC on every sample period. The filter calculation is done once for every pulse on the i_dataReq signal.
- **o_audioSampleFiltered** is the output of your digital filter. This is 16 bits of left and 16 bits of right channel. Each channel is a 16 bit signed fixed point number with 15 bits after the implied

decimal point. You can duplicate the output of your filter, to fill the upper and lower 16 bit samples in o_audioSampleFiltered.

How to Implement Filter

The VHDL code below can be used in your VHDL filter module. The comments explain what each block does.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_textio.all;
use std.textio.all;

-- Signal declarations
signal filterInOneChannel : signed(15 downto 0);
signal filterInResized : signed(35 downto 0);
signal filterSection_1in : signed(35 downto 0);
signal filterOutput : signed(35 downto 0);

-- Grab just one channel from input
filterInOneChannel <= i_audioSample(15 downto 0);

-- Simply resize the 16 bit input to 36 bits. There is an implied
-- divide by 4 involved in this, since we are going from 15 bits to
-- 17 bits after the implied decimal point. This will be canceled by
-- the multiply by 4 on the output.
filterInResized <= resize(filterInOneChannel, filterInResized'length);

-- Implement the divide by 16 which is multiplier s(1)
filterSection_1in <= shift_right(filterInResized, 4);

-- Grab the lowest 16 bits of your filter output and place them
-- into the output port. There is an implied multiply by 4 here
-- due to going from 15 bits to 17 bits after the decimal. This cancels
-- the previous divide by 4.
o_audioSampleFiltered <= filterOutput(15 downto 0) & filterOutput(15 downto 0);
```

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 (see above) at the top of your VHDL code.
- The **triangles** are multipliers. We will be working with 36 bit, signed, fixed point numbers, so we need to account for…
    - 17 bits of fraction
    - 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). Instructions for generating the multiplier IP We will then instantiate **multiple instances of this component, one for each multiplier**.

The procedure to create the multiplier IP is as follows:

1. Go to the **IP Catalog**, at the right side of the screen in Quartus.
2. Go to Installed IP->Library->Basic Functions->Arithmetic->LPM_MULT.
3. Enter the desired name, this will be the name of files created and of the entity you will instantiate for the multiplier.
4. Choose VHDL for the language.
5. It will take a few seconds for the wizard to open.
6. Follow the screen shots on MyCourses under Multiplier Configuration for required settings.
7. You will need to manually enter 36 bits for width, as it is not in the pull-down.
8. On the last screen, make sure you check **Instantiation Template File**. You will then be able to copy and paste the multiplier instantiation template from this file into your code.

The output of the multiplier will be 72 bits, with 34 bits after the decimal. Therefore we must drop 17 of the bits after the decimal (lower bits), as well as truncate some of the upper bits to convert the result back to 36 bits. An example of code to do this multiplication is shown below.

```
filter_mult_a21 : filter_mult
  port map (
    dataa => x1_d1,
    datab => a21_const,
    result => multOuta21_full
  );
multOuta21 <= multOuta21_full(52 downto 17);
```

- Remember that the calculation of the filter output happens once per audio sample (NOT on every 50MHz Clock cycle) . The $\mathbf{z^{-1}}$ blocks are delays. They are like d-ff's, that are clocked 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. Example code to implement the $\mathbf{z^{-1}}$ delays is shown below.

```
if (rising_edge(clk_50MHz)) then
  if (i_reset = '1') then
    delayOutput <= (others => 0);
  elsif (i_dataReq = '1') then
    delayOutput <= delayInput;
  end if;
end if;
```

In order to improve the maximum clock rate, (we will look more at this later), you will add three additional z^{-1} delays not shown in the figures. Two of these are between the filter sections, and one is on the filter output.

Part 5 – Verification of Your Digital Filter VHDL

In this part of the lab, you will write a test bench to verify your design from part 4. You will execute this test bench using ModelSim. In order verify your design, your testbench will read a waveform from a file, process it through your filter, and output the filtered waveform to a file. In order to verify functionality, you will plot the output using Excel. If your filter is functioning properly, the magnitude of the sine wave output should be 0.05 peak value. Remember that you will need to divide the integer output of your filter by 2^17 in Excel, in order to convert it to a floating point number.

You are provided with the following to help guide you through this process…

- The file raminfr_be_tb.vhd demonstrates most of the VHDL code structures you will need in your testbench.
- The file test_filter_tb.v is a Verilog version of the exact testbench you need to write. Verilog and VHDL are the two most commonly used languages for digital hardware design and simulation. Common sense and a little research should allow you to figure out what this code is doing.
- The file containing the input waveform is included in the lab 5 folder on MyCourses, and is called one_cycle_200_8k.
- The code below gives you a template you can use for part of your testbench. This template includes code for doing the file I/O in VHDL, which is not as straight-forward as it is in Verilog.

```
stimulus : process is
  file read_file : text open read_mode is ./src/verification_src/one_cycle_integer.csv";
  file results_file : text open write_mode is ./src/verification_src/output_waveform.csv";
  variable lineIn : line;
  variable lineOut : line;
  variable readValue : integer;
  variable writeValue : integer;
begin
  wait for 100 ns;
  reset <= '0';
  -- Read data from file into an array
  for i in 0 to 39 loop
    readline(read_file, lineIn);
    read(lineIn, readValue);
    audioSampleArray(i) <= to_signed(readValue, 16);
    wait for 50 ns;
  end loop;
  file_close(read_file);

  -- Apply the test data and put the result into an output file
  for i in 1 to 100 loop
    for j in 0 to 39 loop

      -- Your code here...

      -- Write filter output to file
      writeValue := to_integer(audioSampleFiltered);
      write(lineOut, writeValue);
      writeline(results_file, lineOut);

      -- Your code here...

    end loop;
  end loop;

  file_close(results_file);
```

```
  -- end simulation
  sim_done <= true;
  wait for 100 ns;

  -- last wait statement needs to be here to prevent the process
  -- sequence from re starting at the beginning
  wait;

end process stimulus;
```

Part 6 – Validation of 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.

The instantiation of the filter in your top level code should look like the following. Please note the conversion from `signed` to `std_logic_vector` and vice-versa, you will need this.

```
-- Instantiate the audio filter
wave_data_signed <= signed(wave_data);
wave_data_filtered <= std_logic_vector(wave_data_filtered_signed);
audio_filter_inst : audio_filter
  port map (
    i_clk_50              => CLOCK2_50,
    i_reset               => reset,
    i_audioSample         => wave_data_signed,
    i_dataReq             => dataReq,
    o_audioSampleFiltered => wave_data_filtered_signed
  );
```

You will also need to add to the top level VHDL code so that the port “i_audioSample” which is mapped to the component codec_dac_interface comes from the dual port ram when SW0 is 0 or the output of the digital filter when SW0 is 1.

As you step through the frequencies, using **KEY1**, you should be able to hear higher frequency tones are attenuated, relative to the lower frequency tones. In other words, moving switch **SW0** for the lower frequency tones should not have much effect, but for the higher tones you should clearly hear the attenuation of the filter (i.e. the filtered output won’t be as loud).