AUDIO VISUALIZER

Dinu Diana-Gabriela

AUGUST 8, 2018

FINAL PROJECT FOR THE 2018 FPGA DESIGN SUMMER SCHOOL

Abstract

This report presents the design and implementation in Verilog of an Audio Visualizer using the DE1 Altera FPGA board.

The board comes with a 24-bit CD-quality audio CODEC with line-in, line-out, and microphone-in jack and a VGA connector which makes it the perfect environment for an Audio Visualizer to be implemented. The FPGA can synthetize a big number of devices simultaneously, therefore creating a multi-purpose device.

In this project, the code is structured in three main components which correspond to the audio interface, digital signal processing and the VGA controller.
In order to display the spectrum for the digital signal processing, the Fast Fourier Transform is used. Other mathematic functions are used in order to calculate the amplitude of each Fast Fourier Transform bin.

This project makes a playback device with adjustable volume from the FPGA board. It can also overlap the two audio signals from the line-in and the mic-in ports, making this a good karaoke machine for the user that wishes to have fun. Besides this, it should also be able to display on a monitor, through the VGA cable, the spectrum of the sound.

Table of Contents

Table of figures

1. Introduction

The FPGA boards allows programmers to manipulate signals however they want and use them in multiple forms. In this project, the continuous audio signals will be processed in real time through the FPGA board.

The purpose of this project is to display, on a monitor, the frequency spectrum of the input audio signal in real time and to play the music through the line-out port into headphones or speakers. This was accomplished using the Fast Fourier Transform to process the digital signal from the on-board analogue to digital converter.

The analog music signal will be inserted through the line-in port from a music playing device such as a phone or a mp3 player. Using the on-board digital to analogue converter that the Wolfson WM8731 audio CODEC has, it will be able to process this signal once it is sampled.

The audio CODEC offers a large variety of sampling frequencies which is very useful for this project. The sampling frequency and other aspects regarding the configuration of the audio CODEC will be discussed in the corresponding sections below.

The spectrum is represented by 64 vertical bars. These bars correspond to half of the Fast Fourier Transform bins.

Since this only requires the display of the positive frequency spectrum, just the upper half of these bins was used. Because the input signal only has a real component, the spectrum computed by the Fast Fourier Transform will be symmetrical. The positive frequencies are the ones of interest, and therefore there is no need to display the negative part.

Each bin displayed corresponds to a frequency interval. The Fast Fourier Transform was used with the necessary signals such that the spectrum displayed will cover most of the audible range of frequencies.

The audio CODEC can process stereo audio input and output, meaning there will be data for both the right and the left channel. This makes it possible to display two frequency spectrums, one for each channel, and a combined one, that results from the calculation of the mean of both frequencies in the corresponding bin. The project was supposed to have both types of spectrums, but due to the time limit and some problems that are discussed later in this report, only one spectrum is displayed on the monitor. The spectrum displayed is a mean between the two channels.

2. Modules

This project contains three main modules, each of them serving a different purpose: the first one configures the CODEC through an I2C interface and also provides the user interface for tasks such as volume control; the second one takes the digital data from the analogue to digital converter and applies the Fast Fourier Transform to each set of data; the third module carries out the calculations required to find the amplitude of each bin and it also displays each bar with the corresponding amplitude.

The schematic below shows how the blocks are connected.

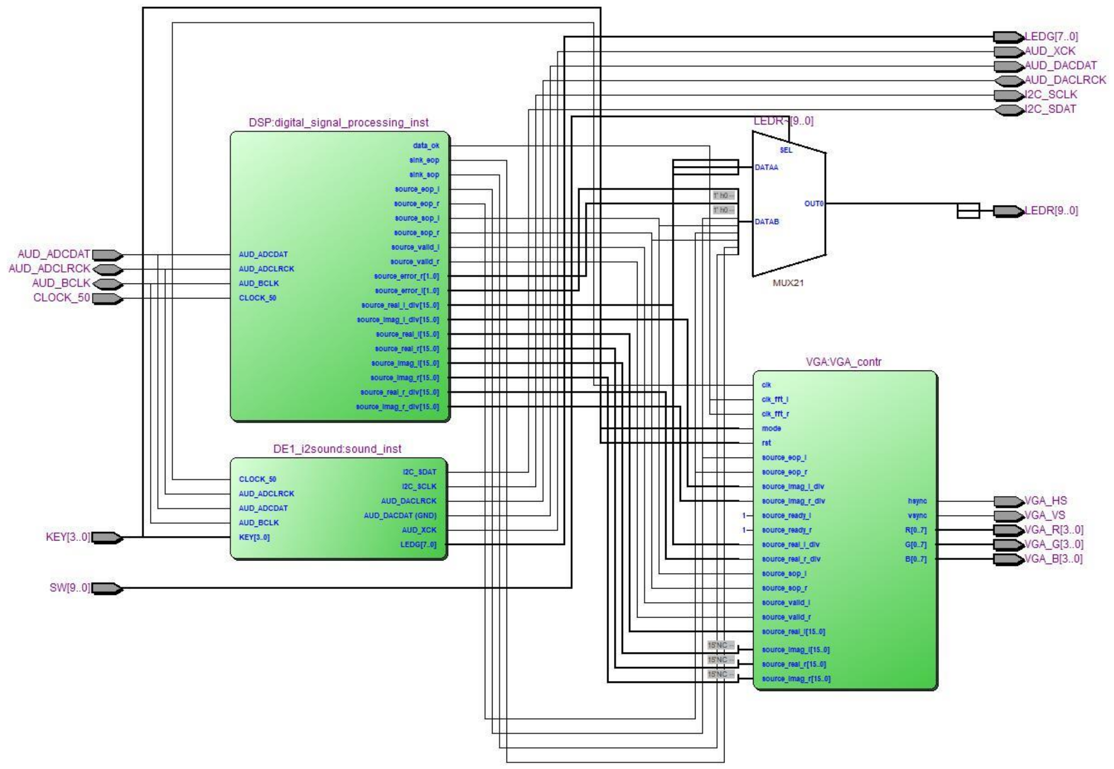


Figure 1 Audio Visualizer - schematic

These three main modules correspond with three stages: the audio stage, the digital processing stage and the display and calculations stage. All these stages need to be completed in order to have a working project.

Each stage is described below mentioning the objectives of each module and their submodules.

2.1. THE AUDIO STAGE

This stage corresponds to three modules that work together to configure the CODEC. These modules make it possible for the music to be heard in the speakers and most importantly, to convert the analog data into a digital signal.

They represent a modified version of one of the example codes that come with the board. They were adapted to work without PLLs and with the necessary parameters that match this projects' objectives.

The three modules are: DE1_i2sound.v, I2C_AV_Config.v and I2C_Controller.v.

The I2C_Controller module contains the necessary code to use the I2C protocol. It is instanced in the I2C_AV_Config module.

The I2C_AV_Config module contains all the necessary addresses and the configuration data that are needed to set up the audio CODEC through the I2C Interface. In

these modules, the parameters and the modes of operation are set for the CODEC. These specifications include:
-sampling rate of 48kHz
-audio interface in DSP mode
-the microphone and the line-in are both heard in the speakers
-master clock frequency of 12MHz
-the Digital Processing Interface, that allows access to the digital audio data, is enabled
-audio data bit length of 16 bits

This module is instanced in the DE1_i2sound module.

The DE1_i2sound module provides easy access to all the input and output ports needed to work with the audio CODEC. It also includes a clock divider that provides the necessary clock frequency to the I2C_AV_Config module, sets the unused ports in the proper state and provides a way for the user to modify and view the volume of the output.

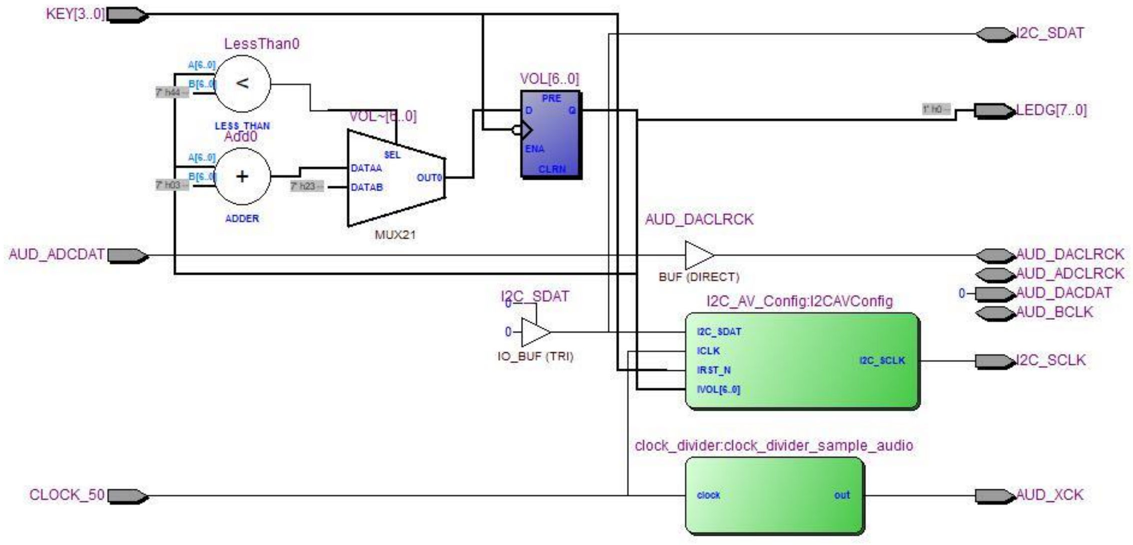


Figure 2 DE1_i2sound - schematic

2.2. THE DIGITAL PROCESSING STAGE

This part consists of four main modules: DSP.v, fft.v, fft_divider.v and SIPO.v. These modules work together to compute the Fast Fourier Transform from the samples received from the audio CODEC.

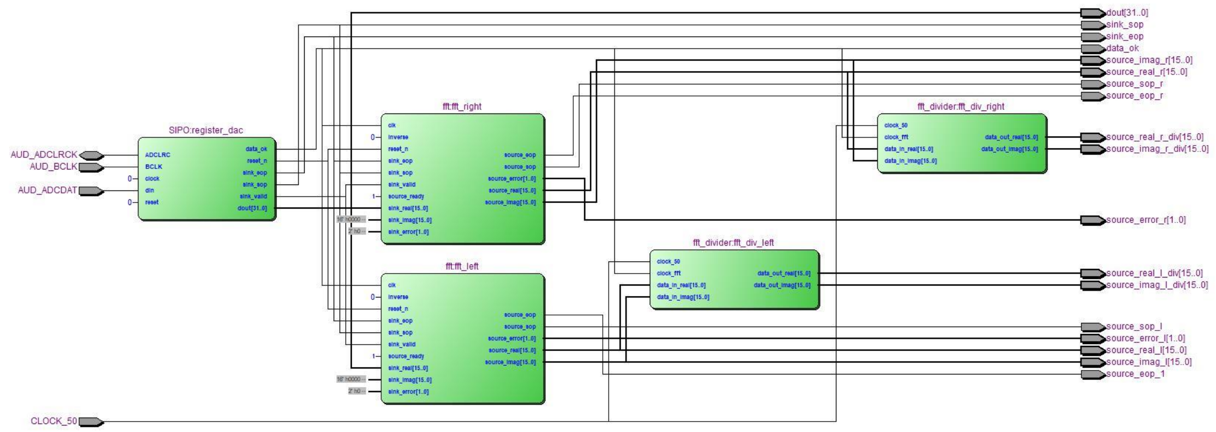


Figure 3 DSP - schematic

The top module in this stage is DSP.v. This module contains an instance of SIPO which receives the samples for both the right and the left channels and sends them to the corresponding instance of Fast Fourier Transform. The DSP module also contains two instances of the Fast Fourier Transform for each channel named fft_left and fft_right. Each instance will have a corresponding fft_divider instance. The fft_divider takes a frame of 128 samples from the output of the fft and displays it at a frequency that allows us to verify that it is in fact working properly, or in the case of displaying on the monitor, at a frequency that can be seen by the human eye.

The SIPO module is, as the name suggests, a serial input parallel output register that also creates the necessary signals for the Fast Fourier Transform module.

Data comes to these modules through a 1bit data bus from the audio CODEC. As it was mentioned before, the data is in 16 bit format and in DSP mode, which means that data is transmitted as shown in the picture bellow.

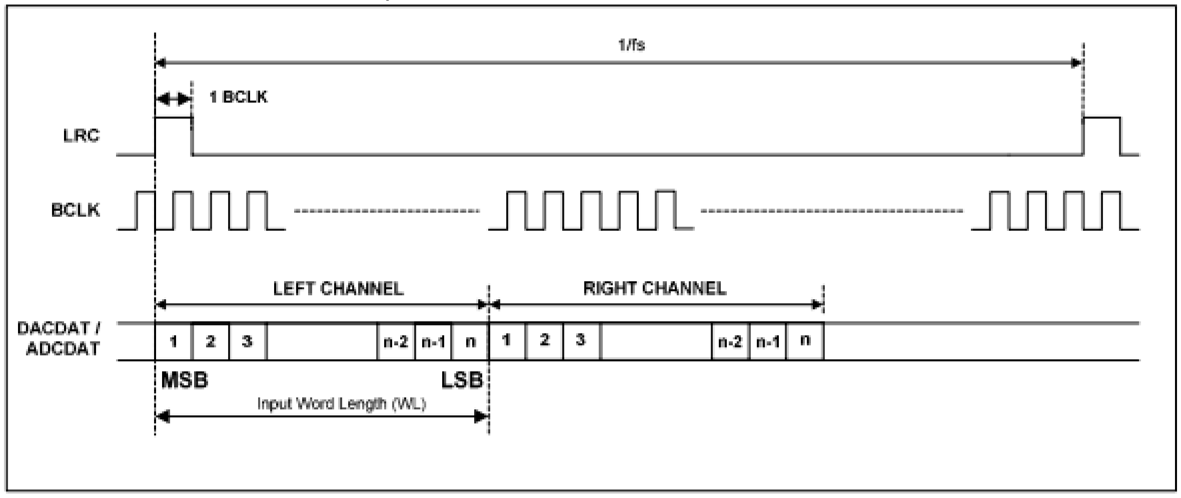
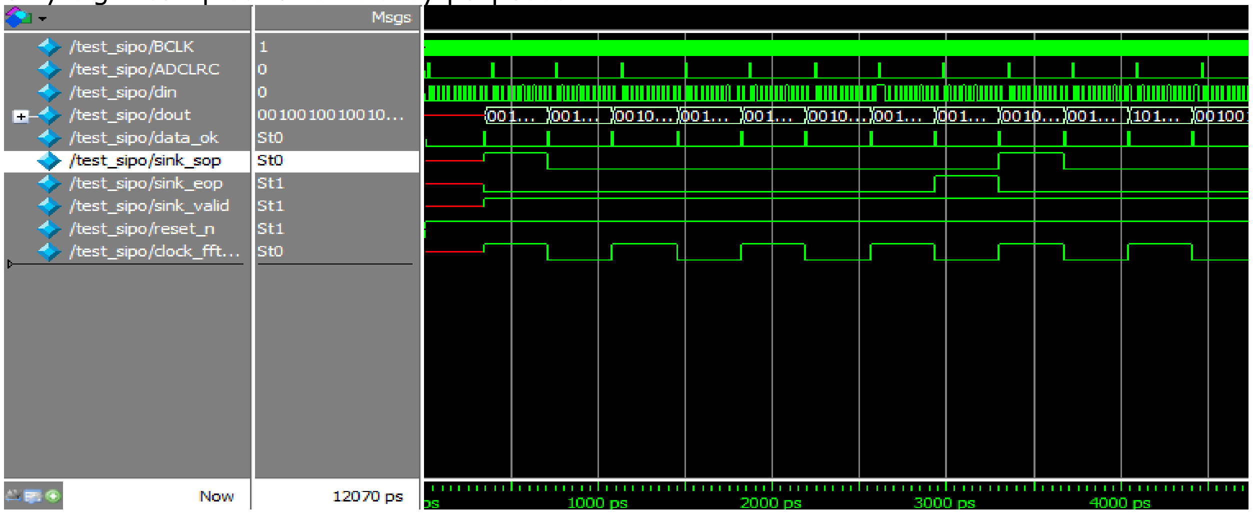


Figure 4 DSP mode

This data fills a 32 bit register. When the register is filled with new data, these 32 bits are transferred in another variable which represents the output for these samples.

In order to use these samples for the Fast Fourier Transform they require to be grouped in frames. This is made with a counter that counts the number of samples. This counter controls two signals which are asserted during the first and respectively the last sample of each frame.

The picture below shows a testbench for this module; the frame is made to contain only eight samples for efficiency purposes.



```
                          Msgs
/test_sipo/BCLK           1
/test_sipo/ADCLRC         0
/test_sipo/din            0
/test_sipo/dout           00100100100100...
/test_sipo/data_ok        St0
/test_sipo/sink_sop       St0
/test_sipo/sink_eop       St1
/test_sipo/sink_valid     St1
/test_sipo/reset_n        St1
/test_sipo/clock_fft...   St0

Now                       12070 ps
```

Figure 5 SIPO - waveform

The fft module is a parameterized megafunction provided by Altera. Using the Wizard in the Quartus II software, this module was parameterized. A 128 point Fast Fourier Transform was created, with an input word size of 16 bits, with streaming I/O Data Flow type. This type of I/O Data Flow allows continuous processing of input data.

The frequency resolution of the bins is in direct correlation with the sampling frequency. In this case, the frequency resolution is 375Hz. The maximum frequency it detects is around 24kHz.

2.3. THE DISPLAY AND CALCULATIONS STAGE

The primary function of this stage is to manage the VGA controller. This project offers two available resolutions: 640 X 480 pixels with 60Hz timing and 800 X 600 pixels with 72Hz timing. The user can select which resolution is used. The second function of this stage is to execute the calculus needed to find the amplitude for each bin.

Firstly, the VGA controller will be detailed.

This stage has as top module VGA.v. It has four main submodules: choose_sync.v, syncro.v, h_sync.v and v_sync.v. These modules are parameterized, so they can be instanced multiple times in order to give the two resolutions.

v_sync and h_sync provide the necessary signals for the vertical and horizontal synchronization with respect to the resolution, which represents a parameter. These two modules are instanced in syncro, a module that uses the synchronization signals to create a signal that notifies if the current pixel displayed at any time is in the active portion of the monitor.

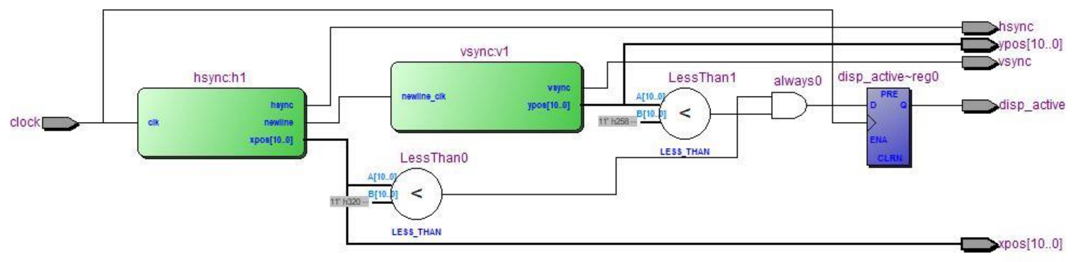


Figure 6 syncro - schematic

In choose_sync the syncro module is instanced twice with the corresponding parameter for the two resolutions. This module is where the switch between the two resolutions happens.

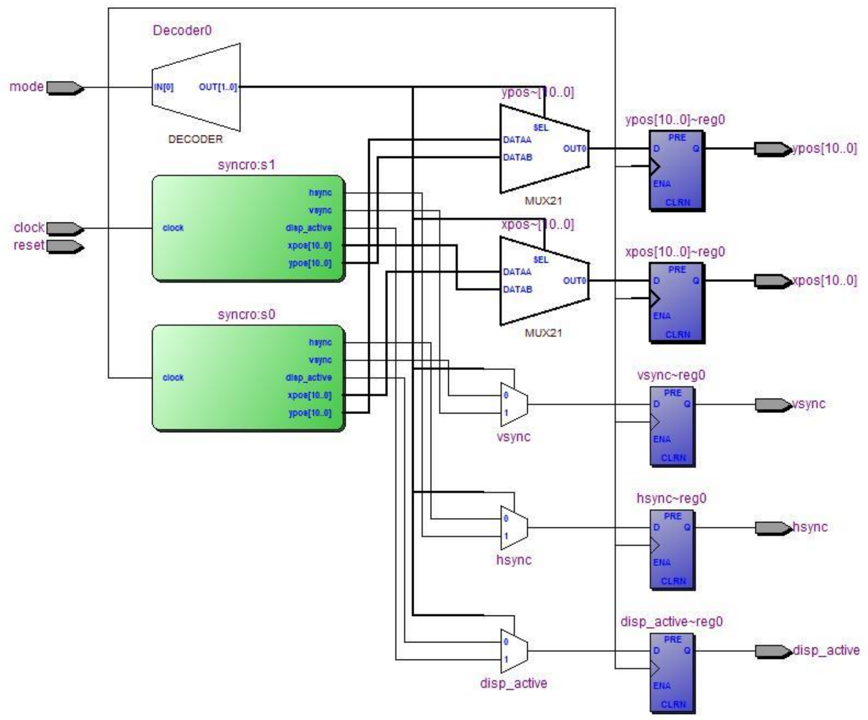


Figure 7 choose_sync - schematic

In the module VGA choose_sync is instanced and there is a clock divider for the clock frequencies needed for the submodules.

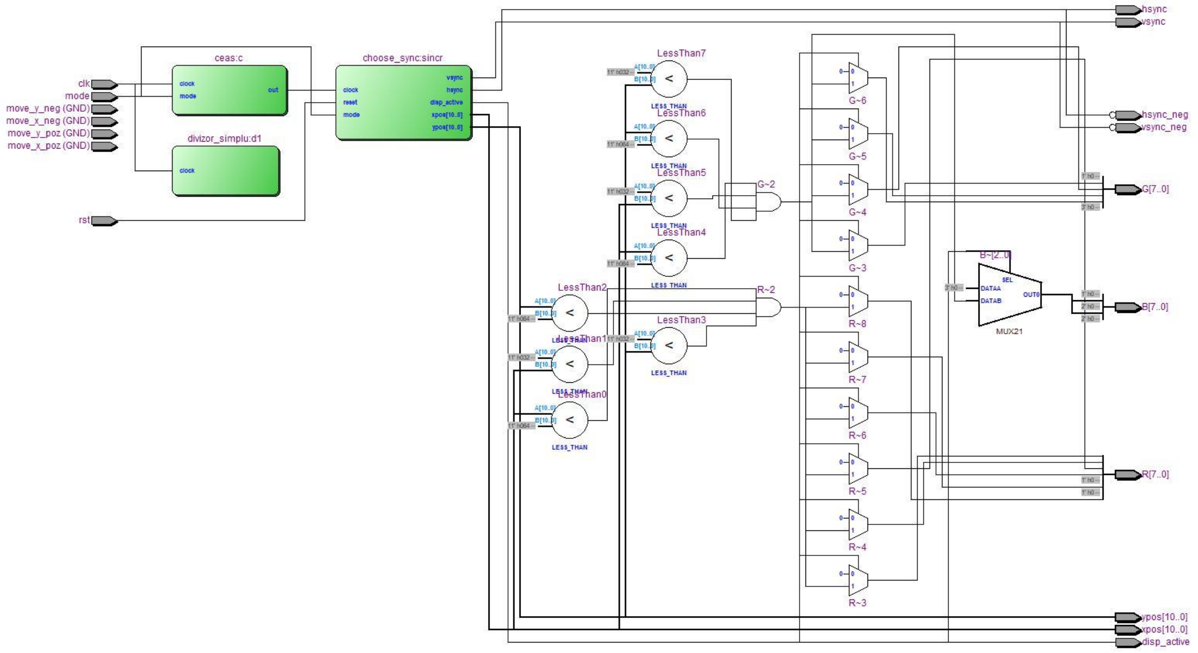


Figure 8 VGA Controller - before FFT calculus - schematic

The second aspect addressed in this stage is the amplitude of each bin. In order to do this, both the imaginary and the real outputs of the Fast Fourier Transform need to be used. For each bin the following mathematical formula was applied, which represents the mean of the two channels:

$$A = \sqrt{\left[\frac{\left(real\,l_{output_{left_{ch}}} + real\,l_{output_{right_{ch}}}\right)}{2}\right]^2 + \left[\frac{\left(imaginary_{output_{left_{ch}}} + imaginary_{output_{right_{ch}}}\right)}{2}\right]^2}$$

In order to do this in Verilog, a few megafunctions were needed. The project contains a function to calculate the square root and the logarithm. The logarithm is used to display the amplitude of the bins on a logarithmic scale, like it is often done in order to be able to easily observe the changes in amplitude. For this, the following formula was used:

$$Amplitude = 10 \times \log_{10} A$$

It is necessary to access all the samples at once to be able to make all these. To do this a virtual RAM was implemented inside the VGA module. Below is a picture that shows the testbench for this piece of code.

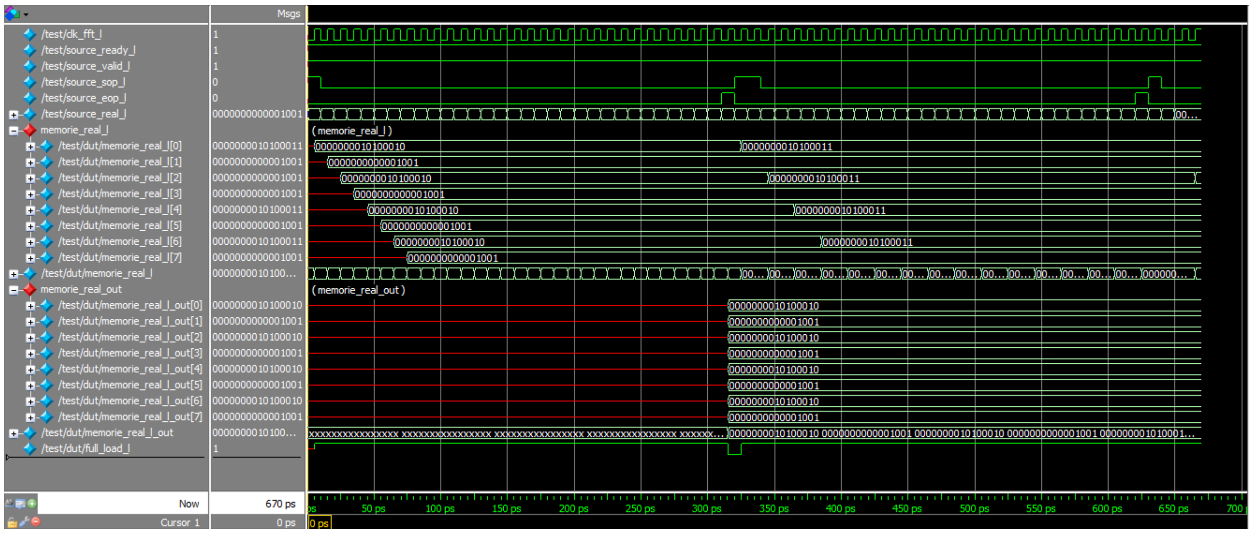

Msgs
/test/clk_fft_l
/test/source_ready_l
/test/source_valid_l
/test/source_sop_l
/test/source_eop_l
/test/source_real_l
memorie_real_l
/test/dut/memorie_real_l[0]
/test/dut/memorie_real_l[1]
/test/dut/memorie_real_l[2]
/test/dut/memorie_real_l[3]
/test/dut/memorie_real_l[4]
/test/dut/memorie_real_l[5]
/test/dut/memorie_real_l[6]
/test/dut/memorie_real_l[7]
/test/dut/memorie_real_l
memorie_real_out
/test/dut/memorie_real_l_out[0]
/test/dut/memorie_real_l_out[1]
/test/dut/memorie_real_l_out[2]
/test/dut/memorie_real_l_out[3]
/test/dut/memorie_real_l_out[4]
/test/dut/memorie_real_l_out[5]
/test/dut/memorie_real_l_out[6]
/test/dut/memorie_real_l_out[7]
/test/dut/memorie_real_l_out
/test/dut/full_load_l
Now 670 ps
Cursor 1 0 ps

Figure 9 FFT calculus - RAM - waveform

3. Top Module

In this section, it is presented the top module of the project. The code below illustrates all the signals and their description. Besides these details, in the comments present in the code the ports that require user input can be observed. It is also explained how the user is supposed to give these inputs.

```
module Audio_Visualizer(
////////////////////////        Clock Input         ////////////////////////
input                           CLOCK_50,                   //      50 MHz
////////////////////////        Push Button         ////////////////////////
input           [3:0]           KEY,                        //      Pushbutton[3:0]
                                                            //      KEY[0] increases volume; It must be
                                                            //pushed once to allow data to be heard on
                                                            //the speakers
                                                            //      KEY[1] changes the resolution
                                                            //      KEY[3] – reset for VGA Controller
////////////////////////        DPDT Switch         ////////////////////////
input           [9:0]           SW,                         //      Toggle Switch[9:0]
                                                            //      SW[1] changes the data displayed on the
                                                            //red LEDs– if SW1 = 1 it shows troubleshooting
                                                            //data, else displays FFT samples from the left
                                                            //channel
////////////////////////////    LED                 ////////////////////////////
output          [7:0]           LEDG,                       //      LED Green[7:0] – displays volume
output          [9:0]           LEDR,                       //      LED Red[9:0] – displays data from FFT – if
                                                            //SW1 = 1 it shows troubleshooting data, else
                                                            //displays FFT samples from the left channel

////////////////////////        I2C                 ////////////////////////////////
inout                           I2C_SDAT,                   //      I2C Data
output                          I2C_SCLK,                   //      I2C Clock
////////////////////////        VGA                 ////////////////////////////
output                          VGA_HS,                     //      VGA H_SYNC
output                          VGA_VS,                     //      VGA V_SYNC
output          [3:0]           VGA_R,                      //      VGA Red[3:0]
output          [3:0]           VGA_G,                      //      VGA Green[3:0]
output          [3:0]           VGA_B,                      //      VGA Blue[3:0]
```

```
////////////////////          Audio CODEC              ////////////////////////////
inout                         AUD_ADCLRCK,                   //        Audio CODEC ADC LR Clock
input                         AUD_ADCDAT,                    //        Audio CODEC ADC Data
inout                         AUD_DACLRCK,                   //        Audio CODEC DAC LR Clock
output                        AUD_DACDAT,                    //        Audio CODEC DAC Data
inout                         AUD_BCLK,                      //        Audio CODEC Bit-Stream Clock
output                        AUD_XCK                        //        Audio CODEC Chip Clock
          );

wire [7:0] R, G, B;

DE1_i2sound sound_inst(
////////////////////          Clock Input              ////////////////////
CLOCK_50,                                                    //        50 MHz
////////////////////          Push Button              ////////////////////
KEY,                                                         //        Pushbutton[3:0]
                                                             //        KEY[0] increases volume
////////////////////////      LED                      ////////////////////////
LEDG,                                                        //        LED Green[7:0]
                                                             //        Used for displaying the volume
////////////////////          I2C                      ////////////////////////////
I2C_SDAT,                                                    //        I2C Data
I2C_SCLK,                                                    //        I2C Clock
////////////////              Audio CODEC              ////////////////////////
AUD_ADCLRCK,                                                 //        Audio CODEC ADC LR Clock
AUD_ADCDAT,                                                  //        Audio CODEC ADC Data
AUD_DACLRCK,                                                 //        Audio CODEC DAC LR Clock
AUD_DACDAT,                                                  //        Audio CODEC DAC Data
AUD_BCLK,                                                    //        Audio CODEC Bit-Stream Clock
AUD_XCK                                                      //        Audio CODEC Chip Clock
);

DSP digital_signal_processing_inst(
////////////////////    Clock Input   ////////////////////
.CLOCK_50(CLOCK_50),                                         //        50MHz
////////////////     Audio CODEC    ////////////////////////
.AUD_ADCLRCK(AUD_ADCLRCK),                                   //        Audio CODEC ADC LR Clock
.AUD_ADCDAT(AUD_ADCDAT),                                     //        Audio CODEC ADC Data
.AUD_BCLK(AUD_BCLK),                                         //        Audio CODEC Bit-Stream Clock
////////////////Fast Fourier Transform Data - Output ////////////
.source_eop_l(source_eop_l),                                 //indicates end of outgoing FFT frame – left channel
.source_sop_l(source_sop_l),                                 //indicated start of outgoing FFT frame – left channel
.source_eop_r(source_eop_r),                                 //indicates end of outgoing FFT frame – right channel
.source_sop_r(source_sop_r),                                 //indicates start of outgoing FFT frame–right channel
.source_error_r(source_error_r),                             //indicates an error has occurred either in an
                                                             //upstream module or within the FFT module – right
                                                             //channel
.source_error_l(source_error_l),                             // indicates an error has occurred either in an
                                                             //upstream module or within the FFT module – left
                                                             //channel
.dout(dout),                                                 //input for FFT – 32 bit bus – ADC data samples –
                                                             //both channels
.sink_eop(sink_eop),                                         //indicates the end of the incoming FFT frame
.sink_sop(sink_sop),                                         //indicates the start of the incoming FFT frame
.source_real_l_div(source_real_l_div),                       //real output data – left channel – divided=slow
.data_ok(data_ok),                                           //indicates a new sample enters the fft – used as
                                                             //clock signal
.source_imag_l_div(source_imag_l_div),                       //imaginary data output – left channel - divided
.source_real_l(source_real_l),                               //real output data – left channel
.source_real_r(source_real_r),                               //real output data – right channel
.source_imag_l(source_imag_l),                               //imaginary output data – left channel
.source_imag_r(source_imag_r),                               //imaginary output data – right channel
.source_real_r_div(source_real_r_div),                       //real output data – right channel – divided = slow
.source_imag_r_div(source_imag_r_div),              //imaginary output data – right channel - divided = slow
```

```
.source_valid_r(source_valid_r),                //asserted by FFT when there is valid data to output
.source_valid_l(source_valid_l)                 //asserted by FFT when there is valid data to output
);

VGA VGA_contr(
///////////////////          Clock Input              ///////////////////
.clk(CLOCK_50),                      //clock from FPGA – 50MHz
///////////////////          Input from user          ///////////////////
.mode(mode),                         //selects the resolution we want the image to be displayed on
                                              //(800 X 600 or 640 X 480)
.rst(rst),
///////////////////          For other modules        ///////////////////
.xpos(xpos),                         //horizontal position
.ypos(ypos),                         //vertical position
///////////////////          VGA Display              ///////////////////
.disp_active(disp_active),           //synchronization signal that indicates if the displayed pixel is in the
                                              //active area of the display or in the back or front porch
.R(R), .G(G), .B(B),                 //4 bit output for the red, green and blue pixel which dictates the color
                                              //of the displayed object
.hsync(VGA_HS),                      //horizontal sync – it activates after the active and the front porch areas
                                     //of pixels were displayed horizontally and stays active until it reaches
                                              //the back porch area
.vsync(VGA_VS),                      //vertical sync – it activates after the active and the front porch areas
                                     //of pixels were displayed vertically and stays active until it reaches the
                                     //back porch area
//////////Fast Fourier Transform Data - Input ////////////
.clk_fft_l(data_ok),                            //left channel FFT clock
.clk_fft_r(data_ok),                            //right channel FFT clock
.source_real_l(source_real_l),                  //real output data – left channel
.source_imag_l(source_imag_l),                  //imaginary output data – left channel
.source_ready_l(source_ready_l),                //asserted if it is able to accept data – left channel
.source_valid_l(source_valid_l),                //asserted by FFT when there is valid data to output
.source_sop_l(source_sop_l),                    //start of incoming frame – left channel
.source_eop_l(source_eop_l),                    //end of incoming frame – left channel
.source_real_r(source_real_r),                  // real output data – right channel
.source_imag_r(source_imag_r),                  // imaginary output data – right channel
.source_ready_r(source_ready_r),                //asserted if it is able to accept data – right channel
.source_valid_r(source_valid_r),                //asserted by FFT when there is valid data to output
.source_sop_r(source_sop_r),                    //start of incoming frame – right channel
.source_eop_r(source_eop_r),                    //end of incoming frame – right channel
.source_real_l_div(source_real_l_div),          //real output data – left channel - divided
.source_real_r_div(source_real_r_div),          //real output data – right channel - divided
.source_imag_l_div(source_imag_l_div),          //imaginary output data – right channel - divided
.source_imag_r_div(source_imag_r_div)           //imaginary output data – right channel - divided
);

assign VGA_R = R[7:4];
assign VGA_G = G[7:4];
assign VGA_B = B[7:4];
assign rst = KEY[3];
assign mode = KEY[1];

assign LEDR[0] = (SW[1] == 1) ? sink_sop : source_real_l_div[0];
assign LEDR[1] = (SW[1] == 1) ? sink_eop : source_real_l_div[1];
assign LEDR[2] = (SW[1] == 1) ? source_sop_r : source_real_l_div[2];
assign LEDR[3] = (SW[1] == 1) ? source_eop_r : source_real_l_div[3];
assign LEDR[4] = (SW[1] == 1) ? source_sop_l : source_real_l_div[4];
assign LEDR[5] = (SW[1] == 1) ? source_eop_l : source_real_l_div[5];
assign LEDR[7:6] = (SW[1] == 1) ? source_error_r : source_real_l_div[7:6];
assign LEDR[9:8] = (SW[1] == 1) ? source_error_l : source_real_l_div[9:8];

wire [15:0] source_real_l_div;
wire [15:0] source_real_l;

endmodule
```

4. Testing and troubleshooting

The testing and troubleshooting of this project were challenging because some modules could not be simulated using ModelSim. In this case it was necessary to find visual ways of debugging.

The red LEDs are a great example of troubleshooting the Fast Fourier Transform. The LEDs were useful in a number of ways: they were used to indicate if the modules had errors, if the audio CODEC was enabled, if there was any output from the Fast Fourier Transform with and without a clock divider (in order to be able to see the LEDs change). This method proved to be an efficient one as it did help to identify and solve a lot of problems.

Another debugging method was simulating modules. This was not always easy to do, as most of the time, the number of inputs needed to view the correct behavior of modules was simply too big. This is the reason why the waveforms presented in this report differ in this aspect.

5. Observations and difficulties

This project is, unfortunately, not finished. In its' current state, the audio CODEC, the Fast Fourier Transform modules and the VGA Controller work as they should, but the part that correlates the output data from the Fast Fourier Transform and the height of the bins is not working.

This was due to not having enough time to finish and some other difficulties mentioned below. In the current state, a possible problem the module might have occurs somewhere between the Digital Signal Processing phase and the calculus part. In other words, the Fast Fourier Transform modules work (as it can be seen on the red LEDs) but no matter what was done, the height of the displayed bars could not move with respect to the amplitude calculated. Another source of error might be the audio CODEC itself. It is possible that the ADC converter does not work as it should: from what can be seen, the amplitude of the bins is high even when there is no music playing through the line-in or mic-in ports. The noise can't justify amplitudes so big that it fills the samples all the way up to the 16th bit for most of the time.

A lot of difficulties were encountered in the making of this project. Most of the problems are related to the absence of a full license, meaning many functions of this FPGA were not accessible. One of the barriers is the fact that the simulation of the Fast Fourier Transform module was not possible. This slowed down the coding process a lot because it was very difficult to understand where the mistakes are found.

Another problem caused by the lack of a license is the fact that PLLs could not be used. This made the implementing of many resolutions for the VGA Controller impossible. It also caused problems with the audio CODEC and its' clock signals. The biggest problem appeared when a clock signal for the Fast Fourier Transform module had to be created. The clock currently in use is an improvisation that, by chance, worked.

The third problem encountered is the fact that, as a beginner, I didn’t find the datasheet for the audio CODEC a very informative one. This specific datasheet changes the notation for some command bits and has notations that are not explained or used by any other manufacturer.

6. Conclusions

In conclusion, this is a hard project to implement, especially when the programmer is unable to use all the necessary functions of the FPGA board. This project required a lot of knowledge about mathematics and coding and also information about digital signal processing, the Fast Fourier Transform, the I2C Protocol and debugging, which contributed to the difficulty level.

If given more time and documentation, this project can be made functional.
Although the project is not finished, it is a good exercise in learning how to use a FPGA board, and more importantly, how to correlate it with signal processing and its many applications.

7. References

1.https://www.intel.co.jp/content/www/jp/ja/programmable/documentation/hco1419012539637.html#dmi1413899813110
2.http://www.cs.columbia.edu/~sedwards/classes/2008/4840/Wolfson-WM8731-audio-CODEC.pdf