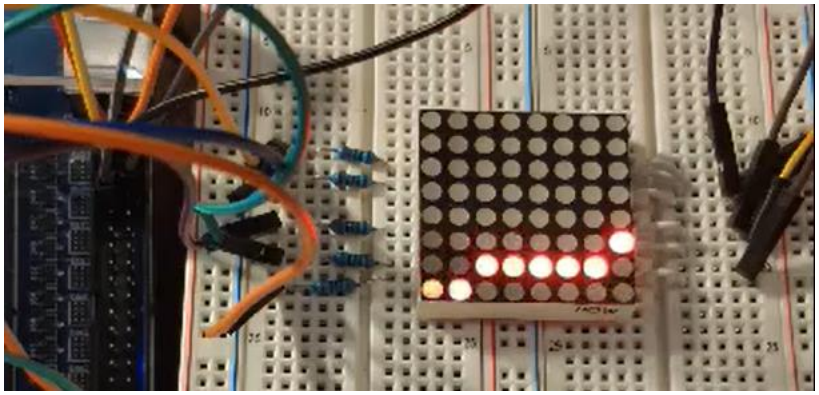
Introduction



Video Demonstration : https://drive.google.com/open?id=1LbecW3AJ5WolBymgIdRnwdaf04mCPvlT

(Song : Higher Love by Whitney Houston & Kygo)

The goal of this project is to build an audio visualizer for an 8x8 LED matrix on FPGA using HDL. The audio visualizer receives analogue audio data and outputs signals for an 8x8 LED matrix. When connected to a LED matrix, the output signals will generate a wave that corresponds to the volume of the input audio. For simplification, three rows on the LED matrix are used.

High Level Design

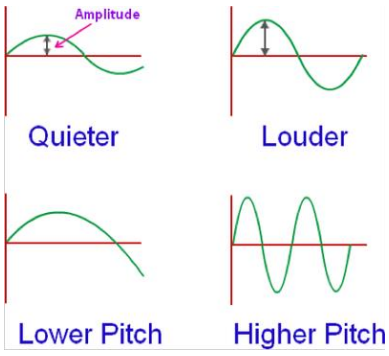


Figure 1. Sound wave

Sound waves come with two components: amplitude and frequency. The amplitude corresponds to the volume of sound, whereas frequency corresponds to its pitch. In this project, the amplitude of input sound, volume, is visualized on a 8x8 LED

matrix. The two main modules in this project are on-board WM8731 audio codec and an audio visualizer.

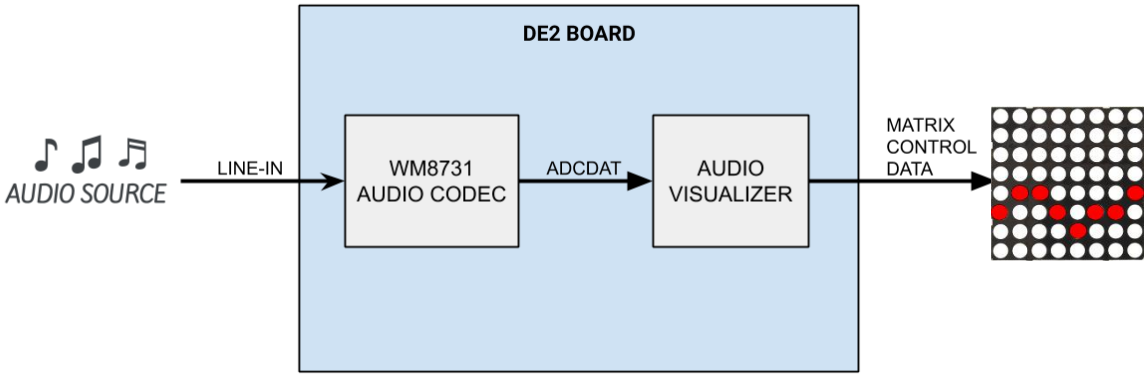


Figure 2. High level design

The audio visualizer is built on Intel's DE-2 board using VHDL and Verilog. Through the LINE-IN/MIC port on board, it receives audio data from an analogue audio source, represented as voltage. However, the audio data needs to be converted to digital signals first before it could be used. The conversion is done by on-board WM8731 audio codec that features ADCs(Analogue-to-Digital-Converter) and DACs(Digital-to-Analogue-Converter). WM873 also attenuates the audio data and filters out unwanted signals.

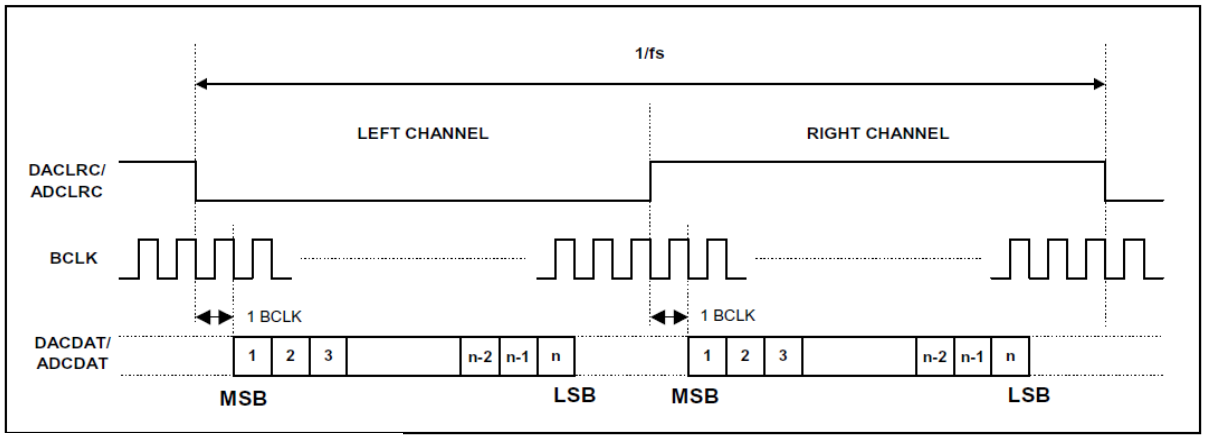


Figure 3. WM8731 I^2S mode

WM9731 module accommodates 4 audio interface modes, which are : left justified / right justified / I^2S/ and DSP mode. Each mode defines how audio data is passed to left and right channel. In this project I^2S mode is used, where MSB of audio data is available on the 2^{nd} rising edge of BCLK(Bit-Clk) after LRC(Left-Right-Clk) transision. Figure 3 describes signals in I^2S mode. For the sampling rate, default of 48kHz is used.

Initially, the analogue audio data travels through WM8731 in I^2S mode and gets converted to digital signals. The digitalized audio data then proceeds to the audio visualizer, which generates corresponding GPIO signals for the LED matrix.

Program/Hardware design

1. WM8731 AUDIO CODEC

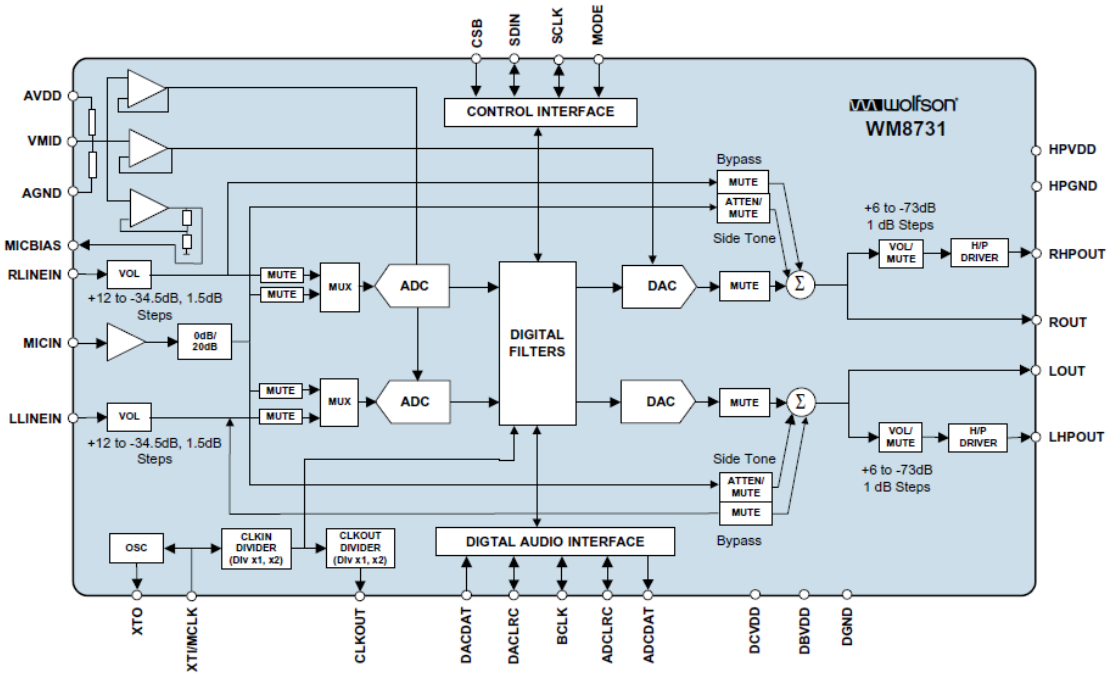


Figure 4. WM8731 schematic

WM8731 is an on-board audio codec that features ADCs and DACs. Audio data from an audio source is first passed through WM8731 for an analogue-to-digital-conversion and audio attenuation. WM8731 also simplifies the ranges of resulting ADCDAT(Analogue-to-Digital-Converter-Data). The attenuation produces more linear and smooth graphs on the LED matrix. On WM8731, any configuration, including sound attenuation, is done by writing values in designated registers. The register address for sound attenuation is "0000101." Figure 5 describes the designated register.

REGISTER ADDRESS	BIT	LABEL	DEFAULT	DESCRIPTION
0000101 Digital Audio Path Control	0	ADCHPD	0	ADC High Pass Filter Enable 1 = Disable High Pass Filter 0 = Enable High Pass Filter
	2:1	DEEMP[1:0]	00	De-emphasis Control 11 = 48kHz 10 = 44.1kHz 01 = 32kHz 00 = Disable
	3	DACMU	1	DAC Soft Mute Control 1 = Enable soft mute 0 = Disable soft mute
	4	HPOR	0	Store dc offset when High Pass Filter disabled 1 = store offset 0 = clear offset

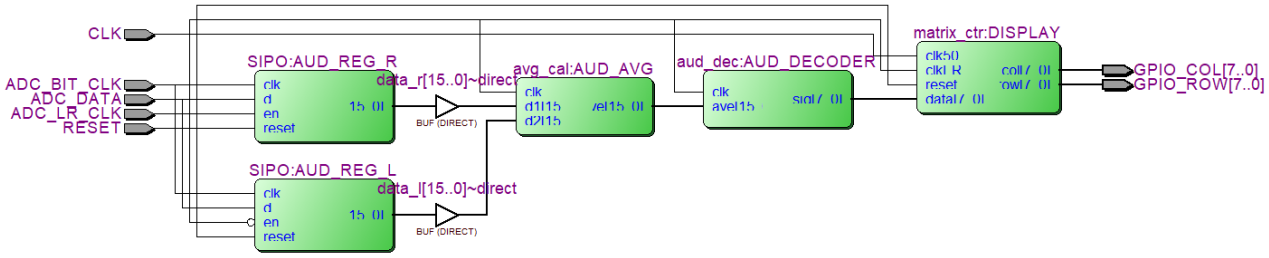
Figure 5. WM8731 register

In order to write to the designated register, Intel's example code was modified implemented to the following :

```
always @(posedge END) begin
   ROM[0]= 16'h0c00;       //power down
   ROM[1]= 16'h0ec2;       //master
   ROM[2]= 16'h0838;       //sound select
   ROM[3]= 16'h1000;       //mclk
   ROM[4]= 16'h0017;       //left channel
   ROM[5]= 16'h0217;       //right channel
   ROM[6]= 16'h0A0b;       //filters
   ROM[7]= {8'h06,1'b0,vol[6:0]}; //sound vol
   ROM[8]= {8'h04,1'b0,vol[6:0]};
```

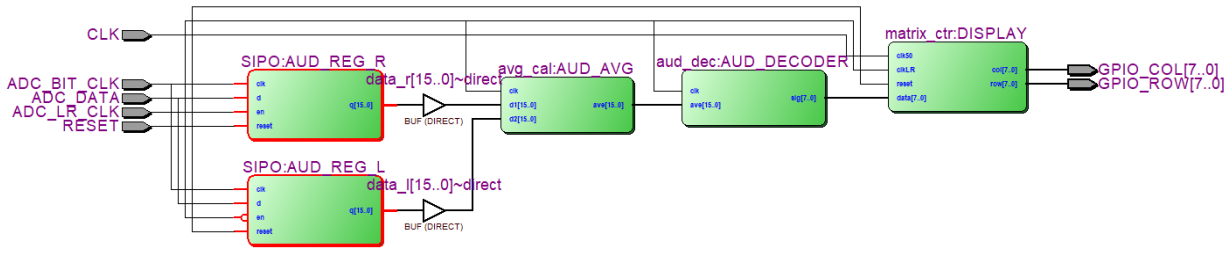
ADC high pass filter was enabled and 32kHz de-emphasis filter was applied.

2. AUDIO VISUALIZER



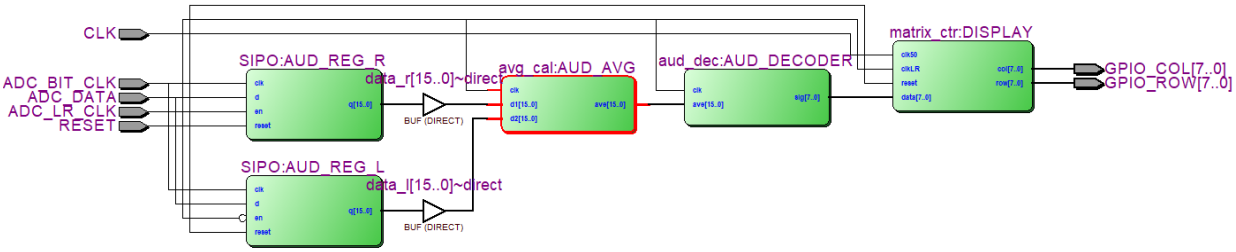
Audio visualizer decodes audio data and generate control signals to light up the 8x8 LED matrix accordingly. Input to the audio visualizer include ADC_DATA, ADC_LR_CLK, ADC_BIT_CLK, RESET, and CLK. ADC_DATA is serial audio data for left/right channel, which is sent alternatively on ADC_LR_CLK. Each data bit is received on the falling edge of ADC_BIT_CLK. For accuracy, the average of left and right audio data is calculated and used in this project. The output to the module includes GPIO_COL[7:0] and GPIO_ROW[7:0]. These outputs control which rows and column light up on the LED matrix. Audio visualizer comprises of smaller modules that perform specific operation. Below are descriptions of the smaller modules in the audio visualizer.

(1) LEFT/RIGHT AUDIO REGISTERS



Left/right audio registers are Shift-In-Parallel-Out registers that convert serial input into 16-bit parallel output. Each left/right register is enabled on ADC_LR_CLK accordingly and receives 32-bit ADC_DATA in serial. The first 16-bits contains the actual audio data and are sent out as output.

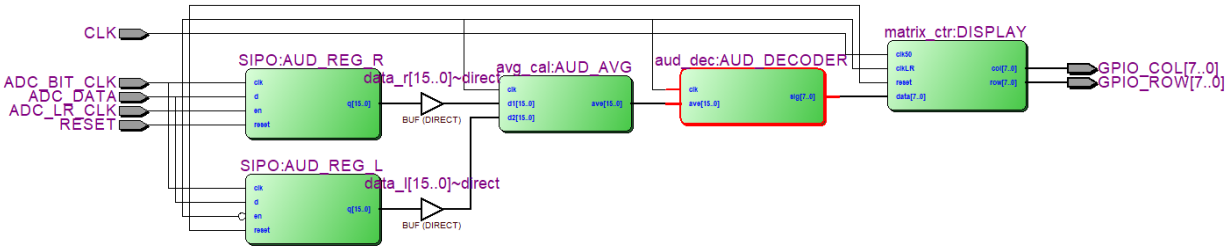
(2) AVERAGE CALCULATOR

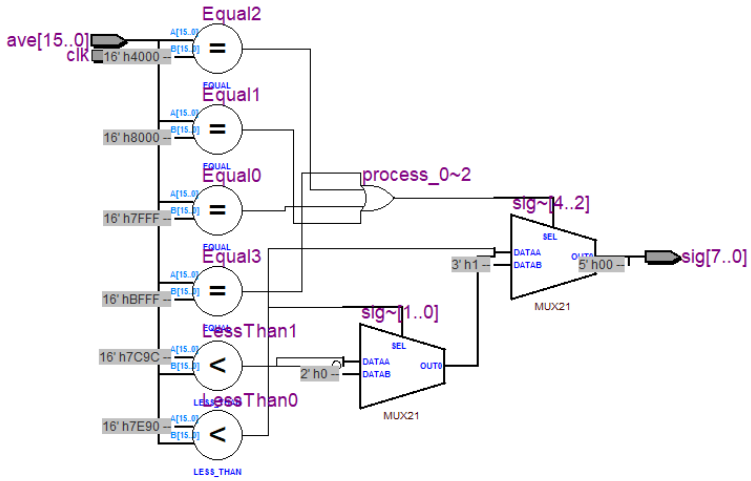


The average calculator calculates the average of left and right channel audio data.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity avg_cal is
    port(clk : in std_logic;
         d1 : in std_logic_vector(15 downto 0);
         d2 : in std_logic_vector(15 downto 0);
         ave : out std_logic_vector(15 downto 0)
         );
end avg_cal;

architecture rtl of avg_cal is
signal sum: std_logic_vector(16 downto 0) := (others => '0');
signal d1_temp : std_logic_vector(15 downto 0);
signal d2_temp : std_logic_vector(15 downto 0);

begin
    process (clk) begin
       if (rising_edge(clk)) then
          d2_temp <= d2;
       elsif (falling_edge(clk)) then
          d1_temp <= d1;
       end if;
    end process;

    sum <= ("0" & d1_temp) + d2_temp;
    ave <= sum(16 downto 1) + sum(0);
end rtl;
```

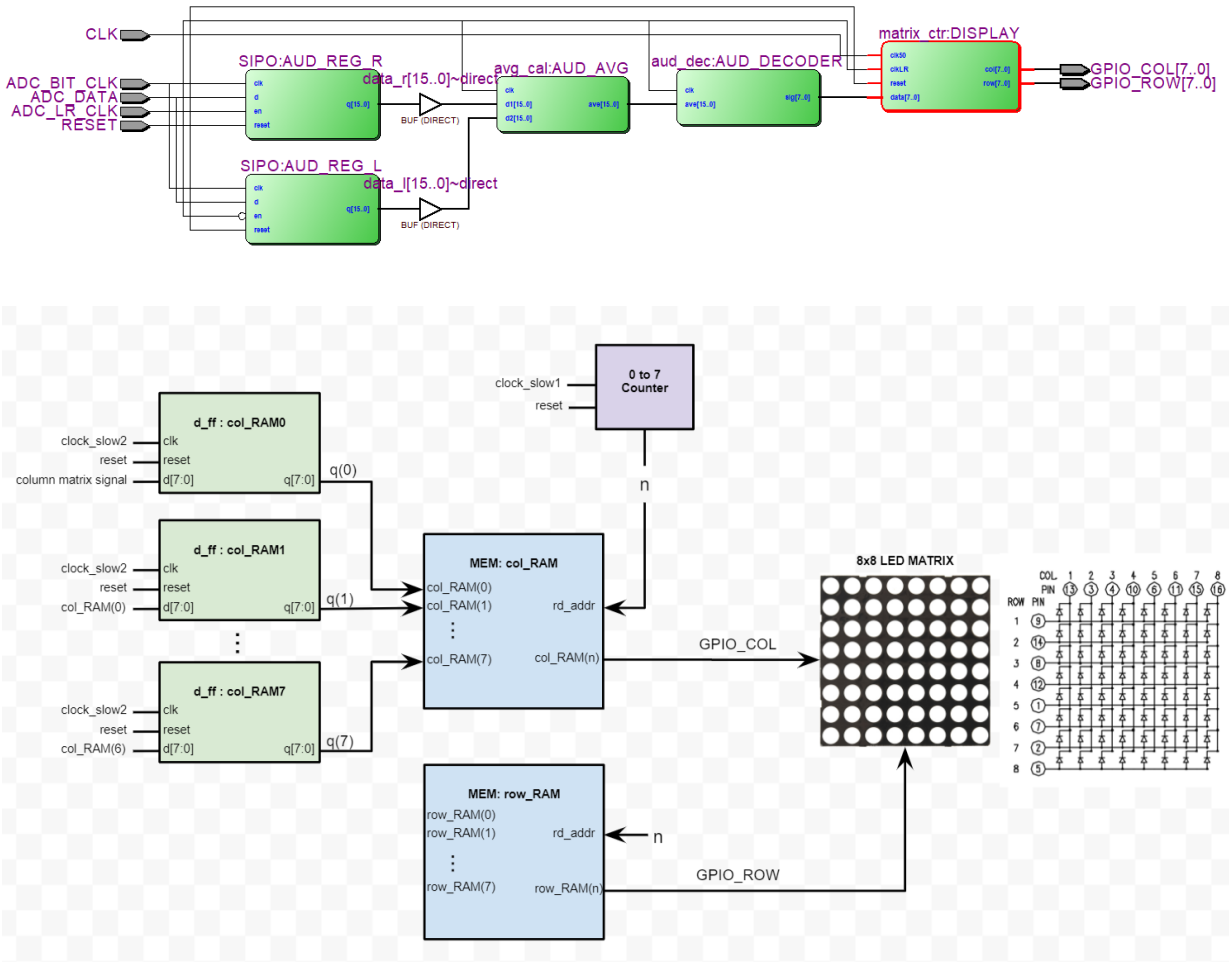
(3) AUDIO DECODER



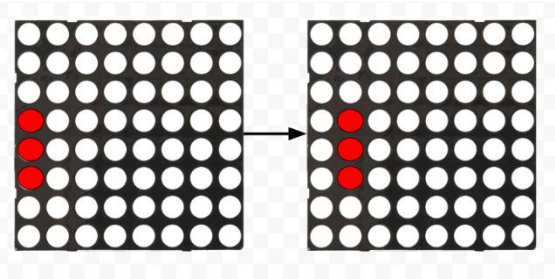


The audio decoder decodes the 16-bit average audio data and output 7-bit signals for the first column on LED matrix.

(4) MATRIX CONTROL



Matrix control is the most important module in Audio visualizer. The matrix control controls the graphic on the LED matrix. At each rising clock, the matrix control takes in LED signals for first column from the audio decoder and shifts previous column data column by column. The goal of this operation is to create the shifting movement across the LED matrix, demonstrated in below picture.



On the other hand, the LED matrix allows only one row to be lit at a time. Thus, each row has to be lit up consecutively at the right rate to make the whole matrix to appear to be lit on. Also, the speed at which the graph shifts across the LED matrix should be set. For these reasons, there are two clocks used in Matrix control, which are slow_clock1(17KHz) and slow_clock2(15Hz). The components that operate on these two different clocks are d_flip_flops and RAMs. D_flip_flops shifts graphic across the LED matrix and RAMs store control signals for lighting up the LEDs.

First, the LED signals from audio decoder is stored in col_RAM(0) at each slow_clock2 cycle. These LED signal data is shifted by the eight d_flop_flops. The row_RAM is only used to alternatively light up each row, thus already loaded with preset data. These LED signals are then read consecutively as outputs at clock2 cycle, using counter that counts from 0 to 7.

Result

(1) Simulation

Before hardware implementation, the operations of WM8781 and the matrix control module were verified. For the matrix control module, ModelSim simulation was done using a testbench.

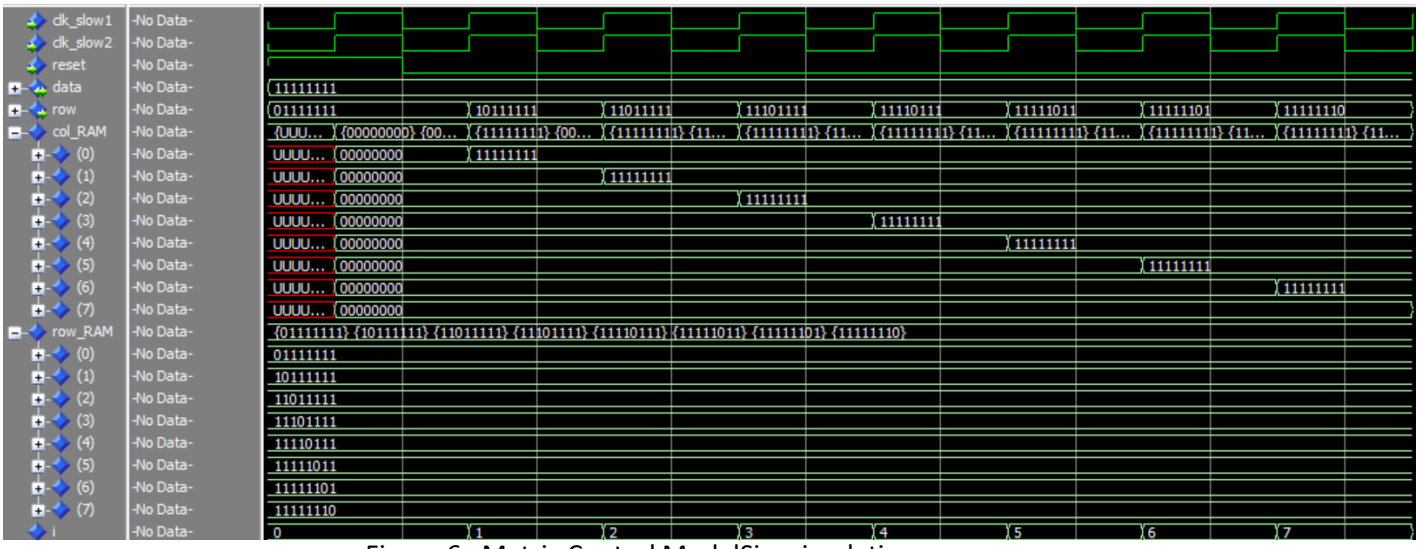


Figure 6. Matrix Control ModelSim simulation

Figure 6 includes the ModelSim simulation result on the matrix control. First, the module was reset, then LED data of "11111111" was fed. Following the reset, it can be verified that col_RAM was updated and shifted by byte at the rising edge of clk_slow2. Also, each RAM was read byte by byte at the rising clock edges.

For the WM8781 audio codec, SignalTap Logic Analyzer on Quartus was used instead of simulation to verify I^2S mode is working properly.

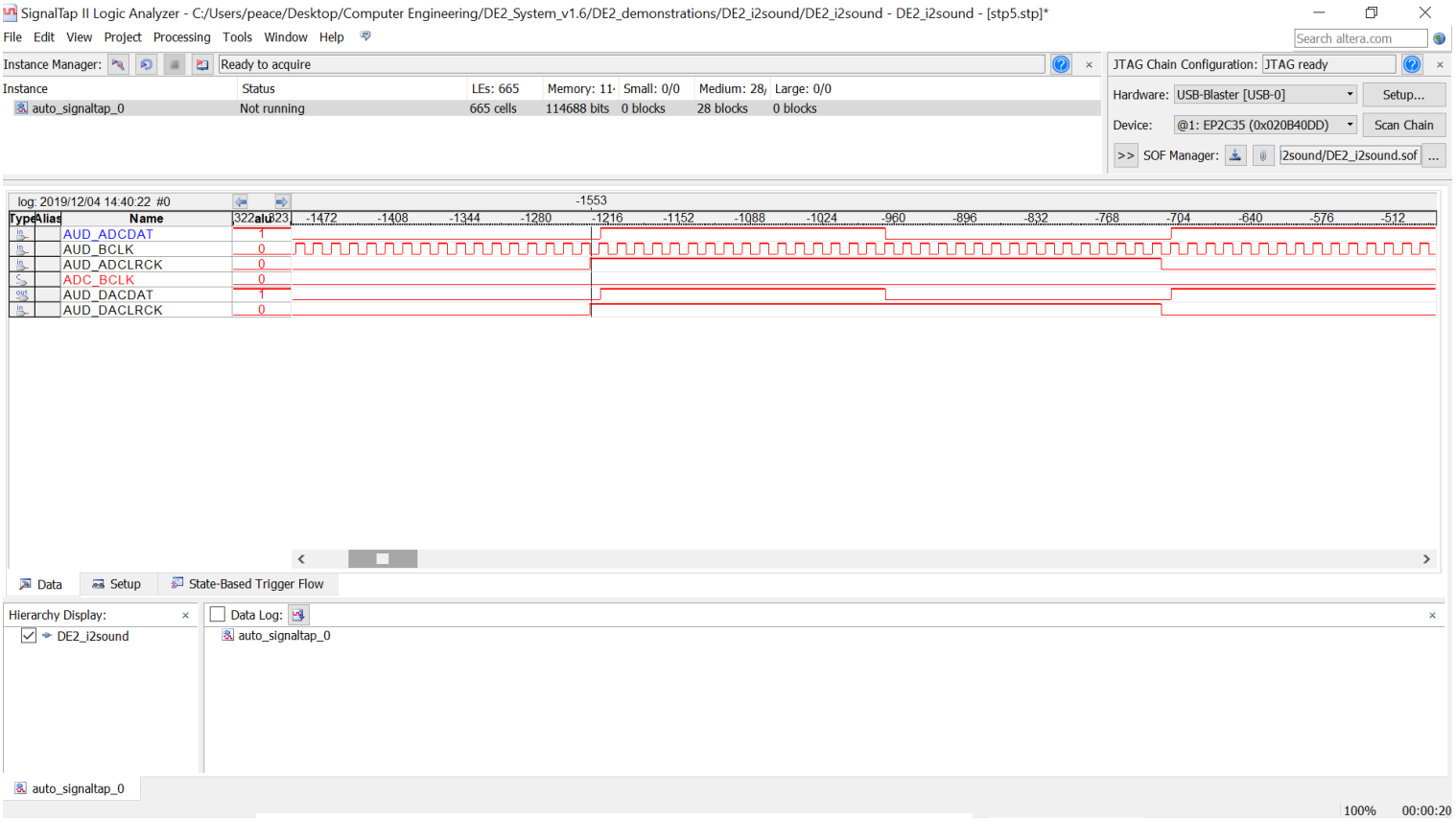


Figure 7. SignalTap Logic Analyzer for WM8781

Figure 7 includes the result of Signal Tap Logic Analyzer. It can be verified that 16-bit audio data is available for each left/right channel following AUD_ADCLRCK.

(2) Hardware implementation

DE2 board was programmed and GPIO pins were connected to a 8x8 LED module. Video :

https://drive.google.com/open?id=1LbecW3AJ5WolBymgIdRnwdaf04mCPvlT

(Song : Higher Love by Whitney Houston & Kygo)

The video demonstrates hardware implementation of the audio visualizer. However, the audio volume had to be set to a certain level (about 50% on Galaxy S8+ phone.) The volume had to be within the range defined in the code.

Conclusion / Future Improvements

The goal of the project was to build an audio visualizer that creates a smooth graph of input audio volume. The audio visualizer was build on DE2 board using VHDL. Overall, the project was successful, and I was able to build an audio visualizer that creates 3 x 8 waveforms on a LED matrix. However, there were limitations and flaws that could be improved in future projects.

First, the module was overly simplified. Instead of using 8 rows of LEDs, I used 3 rows to simplify the process. This decreased the input volume range that could be visualized. As a result, the module only worked over a certain volume level (about 50% volume on Galaxy S8+ phone.) In the future, using more rows of LEDs and setting a wider range for input volume would help.

Second, using volume as input resulted in an inconsistent waveform; volume alone did not provide a responsive waveform that represented the music well. Instead, using a frequency would have resulted in better visualization. However, I could not figure out how to calculate frequency using WM8731 codec. In the future, a different approach would help.

Overall, I learned a lot about microcontrollers through this project. I built everything from the scratch, except WM1875 audio codec, using VHDL and got myself familiar with VHDL. Also, I got to explore how to interface with peripherals, such as a logic analyzer and audio interface and how audio codec works. While doing so, I learned how to interface serial

communication on a microcontroller. Although the project has few improvements to be made, it was a great project that I learned a lot about hardware languages and microcontrollers.

References

https://github.com/biscuit0x/CPE302-Final-Project/tree/master/Datasheets,%20References