

ELEX 7660: Digital System Design

Project: Interactive Graphic Equalizer

April 13, 2018

Brandon Lee

Hubert Pham

Andy Zhen

Table of Contents

Introduction and Background	3
Objective.....	4
Implementation.....	4
LED Module.....	4
Audio Codec via I2C	9
OpenCores I2C IP Core	11
Conclusion	13
Suggestions for Future Work.....	13
Appendix.....	14
LED Module.....	14
LED Pin Assignments	18
I2C Module.....	24

Table of Figures

Figure 1 - Interactive Graphic Equalizer block diagram	3
Figure 2 - IR beam matrix in front of a display.....	3
Figure 3 - An IR emitter/receiver pair.....	3
Figure 4 - Picture of the Actual LED Matrix.....	5
Figure 5 - Wiring Diagram Between FPGA and LED Matrix.....	6
Figure 6 -Circuit Diagram for LED Matrix	7
Figure 7 - Pin functionality as described in the datasheet	9
Figure 8 - Diagram and functionality of the SDA and SCL lines.....	10

Introduction and Background

In sound recording and reproduction, equalization is the process of balancing the strength of different frequency components within an audio signal. The tools used range from discrete equipment called equalizers to software at the consumer-level. An important feature of equalizers is the ability to create a visual representation of a range of frequencies and their relative energy levels in an audio signal, and then be able to adjust those levels as the user sees fit.

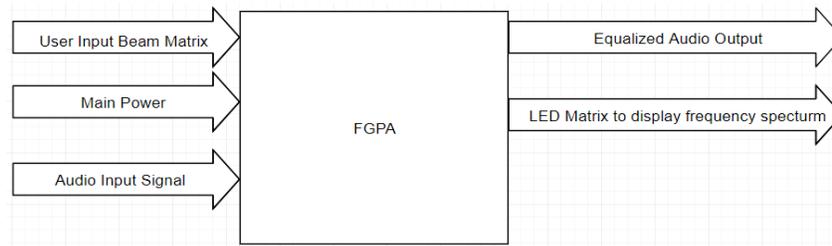


Figure 1 - Interactive Graphic Equalizer block diagram

This project attempts to use a field-programmable gate array (FPGA) as a digital graphic equalizer. It analyzes an input audio signal and displays the frequency spectrum levels in real-time. Users would be able to attenuate the levels (represented on the vertical axis of the display) of individual frequencies (laid-out horizontally) without traditional knobs or sliders, but instead by hovering their finger at the desired height on the display for any of the frequencies displayed.

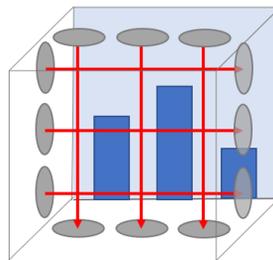


Figure 2 - IR beam matrix in front of a display.

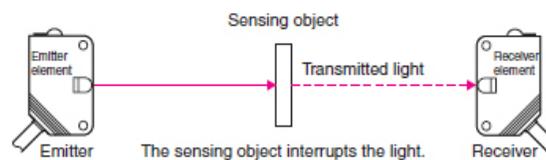


Figure 3 - An IR emitter/receiver pair

The display was to be an upright LED matrix. Another matrix formed by IR emitter/receiver pairs would be placed in front of the display. The dimensions would be limited by budget and technical constraints. The IR matrix would form an “invisible” plane in front of the display that can locate any obstruction (e.g. a finger) in 2D using a row/column break scanning algorithm similar to a switch matrix. Figure 2 shows a matrix formed by 6 pairs of IR break-beam sensors. The audio input and output signals would be line-level and would utilize 3.5mm jacks.

Objective

Our main goal was to create an interactive Graphical Equalizer interface. The goal was to play music from a computer/phone and input it into our FPGA board. Through software, logic would be implemented and the output would light up a LED Matrix. This LED display would then show the levels of each of the 5 frequency bands which can be controlled by the user. To achieve this, we tried to use a matrix made up of infrared sensors. The sensors detect the position of the user’s finger and determine which band the user wants to select and the gain at which to set the band.

As a backup plan, we thought that if the audio inputting interface didn’t work, we would use c code to inject a sequence of tones instead. This way, we would still be able to use the equalizer interface to control the level output of the frequencies. Additionally, the backup plan of the LED Panel was to use the computer monitor instead, through SPI interfacing. On a side note, LED strips would have also been an option when trying to display levels.

The motivation behind the project was mainly our interest in music and sound manipulation. This school term we used a spectral analyzer in the analog and digital communications course. We also learned about digital signal processing and filters in another course. We thought that by choosing this project, we would be able to apply what we learnt throughout the semester in order to create a physical project.

Implementation

LED Module

The purpose of this module is to determine which LEDs on the matrix turns on. The Matrix is divided into a 5 columns and 8 rows. The 5 columns in our LED matrix represents 5 different frequency bands. We chose the following frequencies for each of the bands:

fband1 = 60Hz

fband2 = 200Hz

fband3 = 1000Hz

fband4 = 2 kHz

fband5 = 10kHz

Each of these frequency bands has their purpose when it comes to music. 60Hz gives the song a “thump in a kick drum”. As for 200Hz, this gives richness or “bloom”. Moving on to 1000 Hz, this frequency band gives body and tone to many instruments. And at 2 khz, it controls the sound of guitars and vocals. Last of all, at 10 Khz, this adds overall clarity to the songs.

As for the rows, there are 8 of them, each representing an audio output level (0 through 8). The way this LED matrix works is as follows. For example, given the fband1 (left most column), which controls the audio output level at about 60Hz. The initial level will be determined via music that is playing from the audio codec, and will be varied depending on the user input (to determine what level the user wants it do be). Given that it initial output level is 4, then the bottom 4 LEDs of the left most column will turn on, representing a level of 4 for frequency band 1 (60Hz). The user will then input a value of 8 via IR Beam Matrix, which will turn on all 8 of them in column 1.

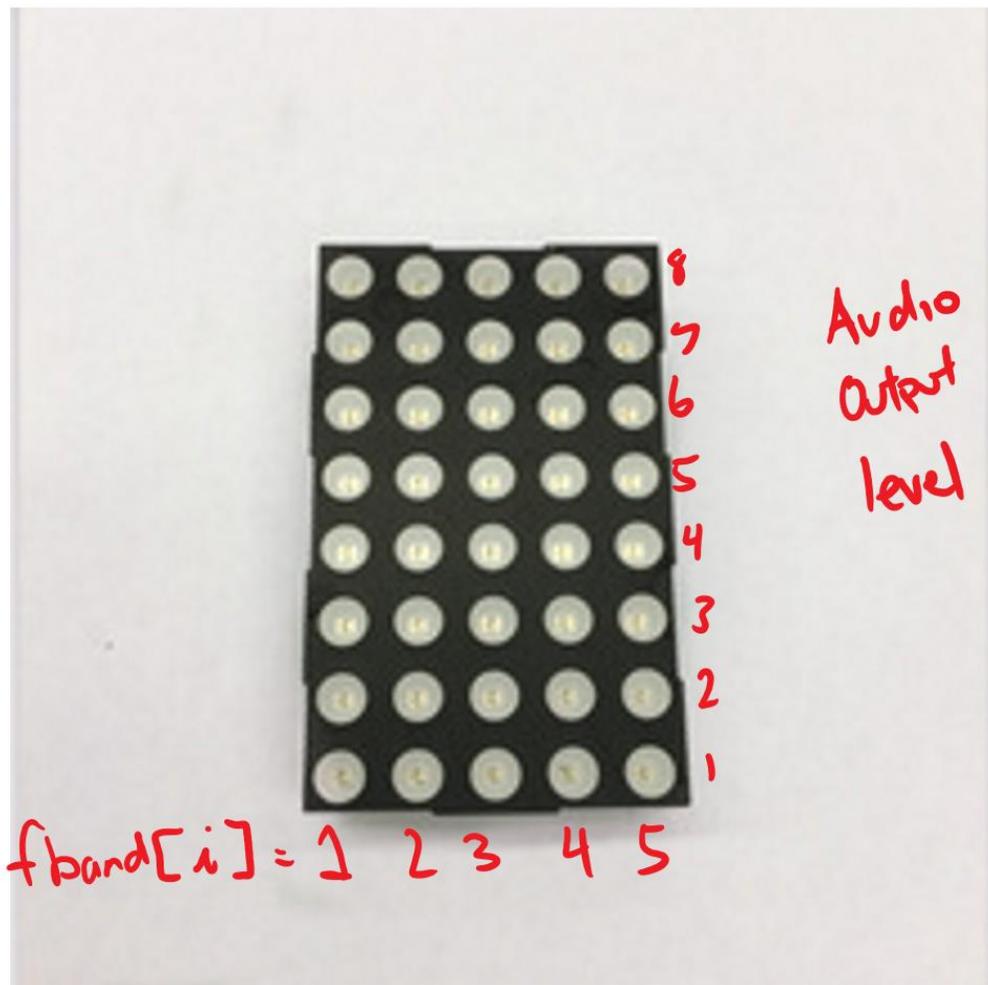


Figure 4 - Picture of the Actual LED Matrix

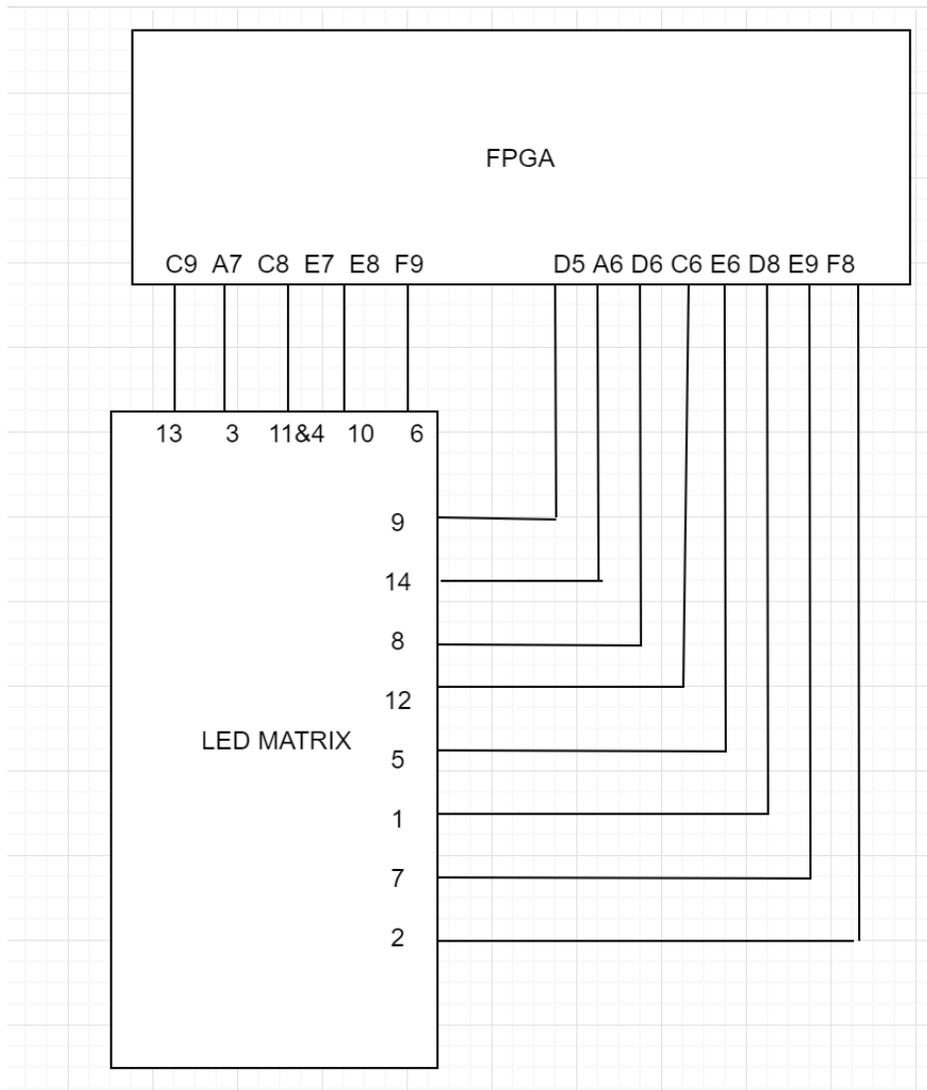


Figure 5 - Wiring Diagram Between FPGA and LED Matrix

The above diagram is the pinouts for how the LED Matrix is connected. This pinouts for the FPGA is configured using the project.qsf file. Using the command “set_location_assignment PIN_C7 -to col[4]” we are able to set the Pin C9 of the FPGA to control our 4th columns logic level. By making it a 0, it turns on the column, allowing the corresponding rows to light up corresponding LEDs.

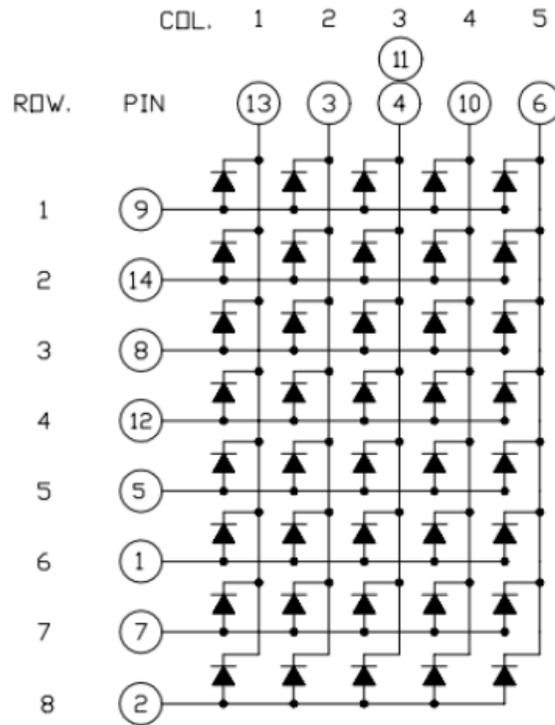


Figure 6 -Circuit Diagram for LED Matrix

The LED Matrix consists of many diodes that are connected together. To turn on certain LEDs, it's just simple circuit analysis. Let's say you want to turn on the top left LED. For it to turn on, Pin 13 has to be connected to ground and Pin 9 will be connected to =3.3V. This way current can run through the diode, thus turning on the LED.

One problem occurs when using the LEDs to display levels at different frequency bands. Say $f_{band1} = 7$, and $f_{band2} = 1$. This means that Pins 13 and 3 will be connected to ground. For column 1 (f_{band1}), turning on 7 LEDs means pins 2,7,1,5,12,8,14 are all on. Now if Pin 3 (column 2) is connected to ground, then we can see that more than 5 LEDs will be turned on. In fact, the bottom 7 LEDs will turn on due to the fact that f_{band1} needs to output a level of 7.

The solution to this is to cycle through each column, one at a time, fast enough that the human eye cannot see it cycling. This means that FPGA will select column 1, output the correct audio level output, then cycle to the next column. This way only 1 column is being powered at a time, thus preventing the problem of incorrectly displaying audio output levels.

IR Matrix

The break-beam matrix was formed using pairs IR emitters and receivers. We used VSMB10940 IR emitting diodes and TSOP38238 IR receivers.

Our initial component choice for detecting the emitted IR light was the VEMD10940F photodiode since it had a matching form factor with the IR diodes and was intended to be used with them. Realizing that filtering and amplification for each photodiode would be required to achieve a suitable signal for interfacing with the FPGA's GPIO pins, we selected the TSOP38238 which is intended for remote control applications.

Each receiver includes a photodiode as well as a pre-amplifier and modulating circuit within an IR-filtering housing. The result is that it is sensitive only to infrared light that is pulsed at 38kHz and provides an active low signal that can be interfaced directly to the FPGA pins. To achieve a 38kHz pulse, we powered all the emitters in parallel from a single GPIO pin outputting a 38kHz square wave by use of a clock divider.

Lastly, to eliminate the possibility of adjacent emitters receiving IR light from any emitter except from the emitter directly across from it, heat shrink tubing was placed around each diode. The resulting IR array was able to track the coordinates of an obstruction accurately and without interference.

Audio Codec via I2C

We chose to implement the audio input/output using the MIKROE-506 Audio Codec Board. The board contains the WM8731 audio codec and provides circuitry and breakout pins for simplified physical interfacing. The board is used to convert an analog audio signal to digital and vice versa by means of an ADC and DAC within the chip. The audio codec is good for applications that require audio playback, and we thought that it would work well with our FPGA implementation because we would be able to input and output music through this device. This module uses 24-bit sigma delta ADCs and DACs with oversampling digital interpolation filters, at a 8kHz to 96kHz sampling frequency. The main interface of the device is controlled using a 2-wire I2C or 3-wire SPI, which is the one of the main modules that we needed to code in order to convert the input and output sound signals.

The function of pins:

SCK	- Digital Audio Bit Clock, Pull Down
MISO	- ADC Digital Audio Data Output
MOSI	- DAC Digital Audio Data Input
ADCL	- ADC Sample Rate Left/Right Clock, Pull Down
DACL	- DAC Sample Rate Left/Right Clock, Pull Down
SDA	- 3-Wire MPU Data Input / 2-Wire MPU Data Input
SCL	- 3-Wire MPU Clock Input / 2-Wire MPU Clock Input

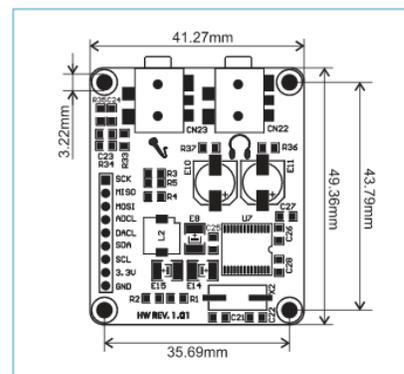


Figure 7 - Pin functionality as described in the datasheet

To get the sound portion of our project working through this codec, we first needed to configure the codec using the I2C protocol. The audio codec functions as an I2C slave device and the FPGA is the master. It is configurable by writing specific values to the codec's various registers. The configurable properties and complete register map is available in the codec's datasheet available on the MikroElektronika website.

On the I2C bus there are two wires called SCL and SDA. SCL is the line that carries the clock that is used to synchronize all data transfers over the I2C bus. The SDA wire is used to transfer the data signal. In the idle state, both of these signals are kept high. To start the configuration and the data transfer, the data line SDA needs to be low (0) while the SCL is high (1). To stop the data transfer (when data transmission is complete) the clock SCL needs to be held high while the SDA is low.

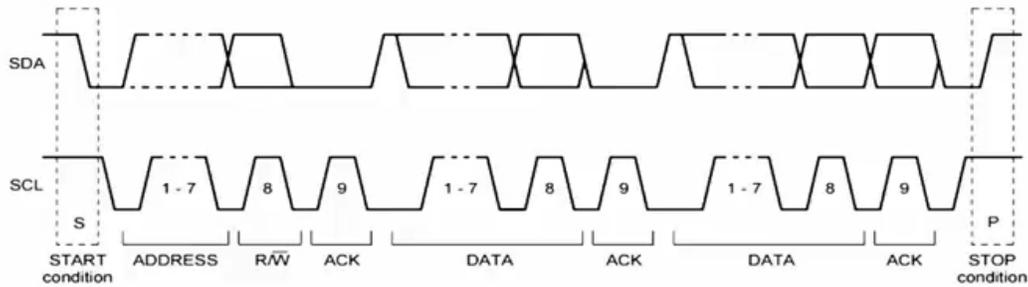


Figure 8 - Diagram and functionality of the SDA and SCL lines

The data line has a pull-up resistor: the line is pulled high if the data state is not being actively controlled, and is pulled low if the I2C device receives an address at the next clock cycle. Data is transferred in sequences of 8 bits, and for every 8 bits transferred, the device receiving the data (ie. the FPGA) sends back an acknowledge (ACK) bit. Once the ACK bit is received, the receiver is ready to accept another byte.

The typical send and receive order of operations for the I2C is as follows. First, the FPGA sends the address of the device it wants to talk to. After a 7-bit address followed by a read/write bit and an acknowledge bit is sent, the data bytes are ready to be sent. The first data byte is then sent, followed by an ACK bit, and then the second data byte is sent, again followed by an ACK. After three ACK bits are sent, the stop condition is generated indicating that the data transmission is complete. In order for the I2C and the data transmission to work properly, it is important to know that the data line should only change while the clock SCL is low, and the acknowledge (ACK) condition is only checked when SCL is high.

The first I2C module that we tried to implement was one that was created by AntonZero. The original code was implemented on a Terasic DE10-Standard development board. We exported the code, which was in a Verilog format, and converted it into a System Verilog file in order to see if we can get the audio codec to function. The code can be found in the Appendix.

We later found a different way of implementing the codec, which was the idea of using I2C opencores. The I2C opencores contain functions which we can include in platform designer. C coding and then use those functions to pass the address and data bits through the I2C.

OpenCores I2C IP Core

The first several lines of the Nios II Console output when executing the main C file is shown below. The OpenCores I2C controller is initialized by specifying the reference clock frequency and desired I2C SCLK frequency. Next, as per the I2C protocol, the address of the slave device (the WM8731) is written to the SDA bus.

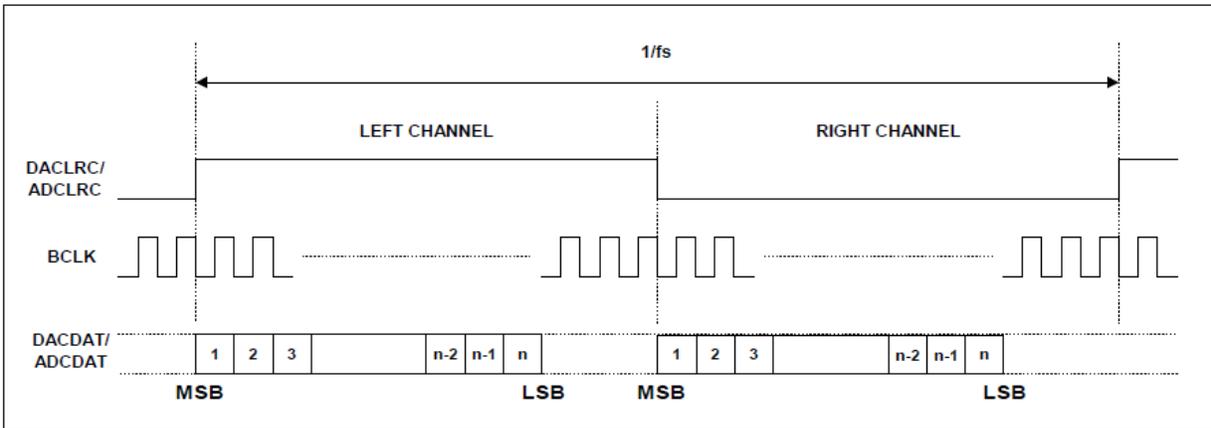
Following the acknowledge, the WM8731 expects the 7-bit register address followed by 9-bits that each pertain to a configurable setting in that register. Because the I2C protocol uses 8-bit frames, the most significant bit of the 9-bit configuration value (if applicable to the particular register) is tagged on at the end of the 7-bit register address as the least significant bit of that frame. The remaining least significant 8-bits are sent in the following frame. The WM8731_CMD function implements this by shifting the 7-bit address left and masking in the most significant configuration bit.

```
I2C master initialized.  
reset register:  
i2c_start slave write...nack = 0  
i2c_write reg addr 1eh...nack = 0  
i2c_write data...nack = 0  
power down control register:  
i2c_start slave write...nack = 0  
i2c_write reg addr ch...nack = 0  
i2c_write data...nack = 0
```

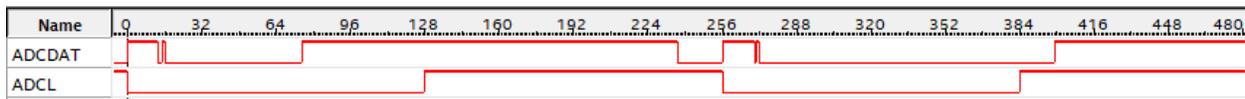
To confirm that I2C communication was functioning, an incorrect slave address was passed to the `i2c_start` function. As expected, this resulted in the not-acknowledged signal returning a '1'. We were then able to set various settings of the codec such as enabling bypass of the line-in and/or microphone signals to the line-out jack of the board, DAC and ADC clocking schemes, sampling frequency (in which we specified the division factor between the bitclock (SCK) and left/right clock (DAKL and ADCL)).

Audio Sampling

The WM8731 was configured to be in slave mode and left-justified mode. In slave mode, the clock signals are inputs to the audio codec, determining the clocking rate of data. The left-right clocks for audio input and output (DACLRC and ADCLRC), and the bit-clock (BCLK) are all generated by the master, in this case the FPGA.



The ADCDAT signal being output by the audio codec and into our FPGA can be seen in the SignalTap Logic Analyzer waveforms below.



Conclusion

While we were not able to fully complete what we intended, we were still able to implement some of the main goals of the project. The physical implementation, such as a working spectrum analyzer that is connected to the FPGA, was not achieved. We found that we were not able to complete everything as planned mostly due to time constraints. It was also difficult to find references that aided in the design and implementation of our specific goals and components.

We learned the value in effective troubleshooting. For example, becoming familiar with SignalTap Logic Analyzer and being able to use it throughout development can reduce many problems as the project progresses. Writing testbenches in SystemVerilog can also verify the operation of individual modules or subsystems before implementing them within a larger system, at which point troubleshooting would seem a much more immense task.

Working on this project has also exposed us to the considerations and methods involved in software and hardware interfacing. We met many obstacles, but this specific experience that involved implementing configuration protocols, generating clocks with the use of phase-locked loops and clock dividers, and referring to datasheets to meet device requirements has reinforced our grasp on the general concepts of hardware/software interfacing and embedded system design.

Suggestions for Future Work

In order to get a fully working spectrum analyzer, you would need to understand many subject areas, such as sampling, quantization, ADC and DAC, DSP, and Fourier Transform, to name a few. Heavy research in these subjects are highly recommended. It is also recommended that in-depth planning of parts and complexity is done prior to implementation so that all components can be easily implemented together.

Appendix

LED Module

```
// LED MODULE

// Controls which LED turns on based on the level of the output on each frequency band

/* module */
module project (
    input CLOCK_50, //fband1,fband2,fband3,fband4,fband5,
    output [3:0]col, [3:0]row);

    int fband1 = 8; // made up input values.
    int fband2 = 6;
    int fband3 = 4;
    int fband4 = 3;
    int fband5 = 1;

    int count = 0;

    //logic for determining which band to light and at what level. Cycles through the 5 bands,
    lighting up each level corresponding to fband(#).

    always_comb begin

        case (count) // count is the number of freq bands (1 through 5).
            0: begin //first freq band
                col = 8'b1111_1110; // turns on col corresponding to the band
                if(fband1 == 0)
                    row = 8'b0000_0000 // turns on no led
                if (fband1 == 1)
                    row = 8'b0000_0001; //turns on bottom led
                if(fband1 == 2)
                    row = 8'b0000_0011; // turns on bottom 2
                if(fband1 == 3)
```

```

        row = 8'b0000_0111; //turns on bottom 3

    if(fband1 == 4)
        row = 8'b0000_1111;
    if(fband1 == 5)
        row = 8'b0001_1111;
    if(fband1 == 6)
        row = 8'b0011_1111;
    if(fband1 == 7)
        row = 8'b0111_1111;
    if(fband1 == 8)
        row = 8'b1111_1111;
    end

1: begin //2nd freq band
    col = 8'b1111_1101;
    if(fband2 == 0)
        row = 8'b0000_0000;
    if (fband2 == 1)
        row = 8'b0000_0001;
    if(fband2 == 2)
        row = 8'b0000_0011;
    if(fband2 == 3)
        row = 8'b0000_0111;
    if(fband2 == 4)
        row = 8'b0000_1111;
    if(fband2 == 5)
        row = 8'b0001_1111;
    if(fband2 == 6)
        row = 8'b0011_1111;
    if(fband2 == 7)
        row = 8'b0111_1111;
    if(fband2 == 8)
        row = 8'b1111_1111;

```

```
end

2: begin // 3rd frequency band
    col = 8'b1111_1011;
    if(fband3 == 0)
        row = 8'b0000_0000;
    if (fband3 == 1)
        row = 8'b0000_0001;
    if(fband3 == 2)
        row = 8'b0000_0011;
    if(fband3 == 3)
        row = 8'b0000_0111;
    if(fband3 == 4)
        row = 8'b0000_1111;
    if(fband3 == 5)
        row = 8'b0001_1111;
    if(fband3 == 6)
        row = 8'b0011_1111;
    if(fband3 == 7)
        row = 8'b0111_1111;
    if(fband3 == 8)
        row = 8'b1111_1111;
    end

3: begin //4th freq band
    col = 8'b1111_0111;
    if(fband4 == 0)
        row = 8'b0000_0000;
    if (fband4 == 1)
        row = 8'b0000_0001;
    if(fband4 == 2)
        row = 8'b0000_0011;
    if(fband4 == 3)
```

```
        row = 8'b0000_0111;

    if(fband4 == 4)
        row = 8'b0000_1111;
    if(fband4 == 5)
        row = 8'b0001_1111;
    if(fband4 == 6)
        row = 8'b0011_1111;
    if(fband4 == 7)
        row = 8'b0111_1111;
    if(fband4 == 8)
        row = 8'b1111_1111;

    end

4: begin //5th freq band
    col = 8'b1110_1111;
    if(fband5 == 0)
        row = 8'b0000_0000;
    if (fband5 == 1)
        row = 8'b0000_0001;
    if(fband5 == 2)
        row = 8'b0000_0011;
    if(fband5 == 3)
        row = 8'b0000_0111;
    if(fband5 == 4)
        row = 8'b0000_1111;
    if(fband5 == 5)
        row = 8'b0001_1111;
    if(fband5 == 6)
        row = 8'b0011_1111;
    if(fband5 == 7)
        row = 8'b0111_1111;
    if(fband5 == 8)
        row = 8'b1111_1111;
```

```

        end

        default: col = 8'b1111_1111;

    endcase
end

always @ (posedge CLOCK_50) begin

    if (count > 4) // resets count for the col.

        count <= '0;

    else count <= count + 1; //counts up to enter next freq band.

end

endmodule

```

LED Pin Assignments

```

set_location_assignment PIN_R8 -to CLOCK_50
set_location_assignment PIN_A15 -to LED[0]
set_location_assignment PIN_A13 -to LED[1]
set_location_assignment PIN_B13 -to LED[2]
set_location_assignment PIN_A11 -to LED[3]
set_location_assignment PIN_D1 -to LED[4]
set_location_assignment PIN_F3 -to LED[5]
set_location_assignment PIN_B1 -to LED[6]
set_location_assignment PIN_L3 -to LED[7]
set_location_assignment PIN_J15 -to KEY[0]
set_location_assignment PIN_E1 -to KEY[1]
set_location_assignment PIN_M1 -to SW[0]
set_location_assignment PIN_T8 -to SW[1]
set_location_assignment PIN_B9 -to SW[2]

```

```
set_location_assignment PIN_M15 -to SW[3]
set_location_assignment PIN_P2 -to DRAM_ADDR[0]
set_location_assignment PIN_N5 -to DRAM_ADDR[1]
set_location_assignment PIN_N6 -to DRAM_ADDR[2]
set_location_assignment PIN_M8 -to DRAM_ADDR[3]
set_location_assignment PIN_P8 -to DRAM_ADDR[4]
set_location_assignment PIN_T7 -to DRAM_ADDR[5]
set_location_assignment PIN_N8 -to DRAM_ADDR[6]
set_location_assignment PIN_T6 -to DRAM_ADDR[7]
set_location_assignment PIN_R1 -to DRAM_ADDR[8]
set_location_assignment PIN_P1 -to DRAM_ADDR[9]
set_location_assignment PIN_N2 -to DRAM_ADDR[10]
set_location_assignment PIN_N1 -to DRAM_ADDR[11]
set_location_assignment PIN_L4 -to DRAM_ADDR[12]
set_location_assignment PIN_M7 -to DRAM_BA[0]
set_location_assignment PIN_M6 -to DRAM_BA[1]
set_location_assignment PIN_L7 -to DRAM_CKE
set_location_assignment PIN_R4 -to DRAM_CLK
set_location_assignment PIN_P6 -to DRAM_CS_N
set_location_assignment PIN_G2 -to DRAM_DQ[0]
set_location_assignment PIN_G1 -to DRAM_DQ[1]
set_location_assignment PIN_L8 -to DRAM_DQ[2]
set_location_assignment PIN_K5 -to DRAM_DQ[3]
set_location_assignment PIN_K2 -to DRAM_DQ[4]
set_location_assignment PIN_J2 -to DRAM_DQ[5]
set_location_assignment PIN_J1 -to DRAM_DQ[6]
set_location_assignment PIN_R7 -to DRAM_DQ[7]
set_location_assignment PIN_T4 -to DRAM_DQ[8]
set_location_assignment PIN_T2 -to DRAM_DQ[9]
set_location_assignment PIN_T3 -to DRAM_DQ[10]
set_location_assignment PIN_R3 -to DRAM_DQ[11]
set_location_assignment PIN_R5 -to DRAM_DQ[12]
set_location_assignment PIN_P3 -to DRAM_DQ[13]
```

```
set_location_assignment PIN_N3 -to DRAM_DQ[14]
set_location_assignment PIN_K1 -to DRAM_DQ[15]
set_location_assignment PIN_R6 -to DRAM_DQM[0]
set_location_assignment PIN_T5 -to DRAM_DQM[1]
set_location_assignment PIN_L1 -to DRAM_CAS_N
set_location_assignment PIN_L2 -to DRAM_RAS_N
set_location_assignment PIN_C2 -to DRAM_WE_N
set_location_assignment PIN_F2 -to I2C_SCLK
set_location_assignment PIN_F1 -to I2C_SDAT
set_location_assignment PIN_G5 -to G_SENSOR_CS_N
set_location_assignment PIN_M2 -to G_SENSOR_INT
set_location_assignment PIN_A14 -to GPIO_2[0]
set_location_assignment PIN_B16 -to GPIO_2[1]
set_location_assignment PIN_C14 -to GPIO_2[2]
set_location_assignment PIN_C16 -to GPIO_2[3]
set_location_assignment PIN_C15 -to GPIO_2[4]
set_location_assignment PIN_D16 -to GPIO_2[5]
set_location_assignment PIN_D15 -to GPIO_2[6]
set_location_assignment PIN_D14 -to GPIO_2[7]
set_location_assignment PIN_F15 -to GPIO_2[8]
set_location_assignment PIN_F16 -to GPIO_2[9]
set_location_assignment PIN_F14 -to GPIO_2[10]
set_location_assignment PIN_G16 -to GPIO_2[11]
set_location_assignment PIN_G15 -to GPIO_2[12]
set_location_assignment PIN_E15 -to GPIO_2_IN[0]
set_location_assignment PIN_E16 -to GPIO_2_IN[1]
set_location_assignment PIN_M16 -to GPIO_2_IN[2]
set_location_assignment PIN_A8 -to GPIO_0_IN[0]
set_location_assignment PIN_D3 -to GPIO_0[0]
set_location_assignment PIN_B8 -to GPIO_0_IN[1]
set_location_assignment PIN_C3 -to GPIO_0[1]
set_location_assignment PIN_A2 -to GPIO_0[2]
set_location_assignment PIN_A3 -to GPIO_0[3]
```

```
set_location_assignment PIN_B3 -to GPIO_0[4]
set_location_assignment PIN_B4 -to GPIO_0[5]
set_location_assignment PIN_A4 -to GPIO_0[6]
set_location_assignment PIN_B5 -to GPIO_0[7]
set_location_assignment PIN_A5 -to GPIO_0[8]
set_location_assignment PIN_D5 -to GPIO_0[9]
set_location_assignment PIN_B6 -to GPIO_0[10]
set_location_assignment PIN_A6 -to GPIO_0[11]
set_location_assignment PIN_B7 -to GPIO_0[12]
set_location_assignment PIN_D6 -to GPIO_0[13]
set_location_assignment PIN_A7 -to GPIO_0[14]
set_location_assignment PIN_C6 -to GPIO_0[15]
set_location_assignment PIN_C8 -to GPIO_0[16]
set_location_assignment PIN_E6 -to GPIO_0[17]
set_location_assignment PIN_E7 -to GPIO_0[18]
set_location_assignment PIN_D8 -to GPIO_0[19]
set_location_assignment PIN_E8 -to GPIO_0[20]
set_location_assignment PIN_F8 -to GPIO_0[21]
set_location_assignment PIN_F9 -to GPIO_0[22]
set_location_assignment PIN_E9 -to GPIO_0[23]
set_location_assignment PIN_C9 -to GPIO_0[24]
set_location_assignment PIN_D9 -to GPIO_0[25]
set_location_assignment PIN_E11 -to GPIO_0[26]
set_location_assignment PIN_E10 -to GPIO_0[27]
set_location_assignment PIN_C11 -to GPIO_0[28]
set_location_assignment PIN_B11 -to GPIO_0[29]
set_location_assignment PIN_A12 -to GPIO_0[30]
set_location_assignment PIN_D11 -to GPIO_0[31]
set_location_assignment PIN_D12 -to GPIO_0[32]
set_location_assignment PIN_B12 -to GPIO_0[33]
set_location_assignment PIN_T9 -to GPIO_1_IN[0]
set_location_assignment PIN_F13 -to GPIO_1[0]
set_location_assignment PIN_R9 -to GPIO_1_IN[1]
```

```
set_location_assignment PIN_T15 -to GPIO_1[1]
set_location_assignment PIN_T14 -to GPIO_1[2]
set_location_assignment PIN_T13 -to GPIO_1[3]
set_location_assignment PIN_R13 -to GPIO_1[4]
set_location_assignment PIN_T12 -to GPIO_1[5]
set_location_assignment PIN_R12 -to GPIO_1[6]
set_location_assignment PIN_T11 -to GPIO_1[7]
set_location_assignment PIN_T10 -to GPIO_1[8]
set_location_assignment PIN_R11 -to GPIO_1[9]
set_location_assignment PIN_P11 -to GPIO_1[10]
set_location_assignment PIN_R10 -to GPIO_1[11]
set_location_assignment PIN_N12 -to GPIO_1[12]
set_location_assignment PIN_P9 -to GPIO_1[13]
set_location_assignment PIN_N9 -to GPIO_1[14]
set_location_assignment PIN_N11 -to GPIO_1[15]
set_location_assignment PIN_L16 -to GPIO_1[16]
set_location_assignment PIN_K16 -to GPIO_1[17]
set_location_assignment PIN_R16 -to GPIO_1[18]
set_location_assignment PIN_L15 -to GPIO_1[19]
set_location_assignment PIN_P15 -to GPIO_1[20]
set_location_assignment PIN_P16 -to GPIO_1[21]
set_location_assignment PIN_R14 -to GPIO_1[22]
set_location_assignment PIN_N16 -to GPIO_1[23]
set_location_assignment PIN_N15 -to GPIO_1[24]
set_location_assignment PIN_P14 -to GPIO_1[25]
set_location_assignment PIN_L14 -to GPIO_1[26]
set_location_assignment PIN_N14 -to GPIO_1[27]
set_location_assignment PIN_M10 -to GPIO_1[28]
set_location_assignment PIN_L13 -to GPIO_1[29]
set_location_assignment PIN_J16 -to GPIO_1[30]
set_location_assignment PIN_K15 -to GPIO_1[31]
set_location_assignment PIN_J13 -to GPIO_1[32]
set_location_assignment PIN_J14 -to GPIO_1[33]
```

```
set_location_assignment PIN_A2 -to qspb
set_location_assignment PIN_A8 -to qsa
set_location_assignment PIN_B8 -to qsb
set_location_assignment PIN_A12 -to ct[0]
set_location_assignment PIN_A5 -to leds[0]
set_location_assignment PIN_C11 -to ct[1]
set_location_assignment PIN_B6 -to leds[1]
set_location_assignment PIN_E11 -to ct[2]
set_location_assignment PIN_B7 -to leds[2]
set_location_assignment PIN_C9 -to ct[3]
set_location_assignment PIN_A7 -to leds[3]
set_location_assignment PIN_C8 -to leds[4]
set_location_assignment PIN_E7 -to leds[5]
set_location_assignment PIN_E8 -to leds[6]
set_location_assignment PIN_F9 -to leds[7]
set_location_assignment PIN_D5 -to kpc[3]
set_location_assignment PIN_A6 -to kpc[2]
set_location_assignment PIN_D6 -to kpc[1]
set_location_assignment PIN_C6 -to kpc[0]
set_location_assignment PIN_E9 -to kpr[0]
set_location_assignment PIN_F8 -to kpr[1]
set_location_assignment PIN_D8 -to kpr[2]
set_location_assignment PIN_E6 -to kpr[3]

set_location_assignment PIN_D9 -to rgb_din
set_location_assignment PIN_E10 -to rgb_clk
set_location_assignment PIN_B11 -to rgb_cs
set_location_assignment PIN_D11 -to rgb_dc
set_location_assignment PIN_B12 -to rgb_res

set_location_assignment PIN_G15 -to jstk_sel
set_location_assignment PIN_A10 -to adc_cs_n
```

```
set_location_assignment PIN_B10 -to adc_saddr
set_location_assignment PIN_A9 -to adc_sdat
set_location_assignment PIN_B14 -to adc_sclk
set_location_assignment PIN_B3 -to spkr
set_location_assignment PIN_D12 -to point

set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to jstk_sel
```

I2C Module

```
/*
I2C - Audio Serial

Credit to: AntonZero

https://github.com/AntonZero/WM8731-Audio-codec-on-DE10Standard-FPGA-board/blob/master/project/i2c.vhd

Two signals required:
1) clock
2) data

idle state: both signals high
start: data low, while clock high
stop: clock high while data low

ACKNOWLEDGE (ACK)

data line has pull-up resistor
line pulled high if: data state not actively controlled
line pulled low if, at next clk cycle,: I2C device receives address

Order of operations:
```

<start signal>

FPGA sends address of device it wants to talk to...

7-bit address, R/W bit sent (followed by ACK)

then data line sent (two bytes, each byte followed by ACK event)

After 3 ACK, stop condition is generated

<stop signal>

*data should only change when clk is low

*ACK condition checked when clk is high

*/

```
module i2c (  
    output logic i2c_busy,  
    output logic i2c_scl,  
    output logic i2c_done,  
  
    input logic i2c_send_flag,  
    input logic [7:0] i2c_addr,  
    input logic [15:0] i2c_data,  
  
    output logic i2c_sda,  
  
    input logic reset, clk  
);  
  
    logic i2c_clk_en = 0;  
    logic clk_en = 0;  
    logic ack_en = 0;  
    logic clk_i2c = 0;
```

```

logic get_ack = 0;

enum {s0,s1,s2,s3,s4,s5,s6,s7,s8} state;

int i2c_state = s0;

int clk_prs = 0;

int data_index = 0; //max 15

//generate two clocks for i2c and data transitions

always_ff@(posedge clk) begin

    if(clk_prs < 250)
        clk_prs <= clk_prs + 1;
    else
        clk_prs <= 0;

    if(clk_prs < 125) //50% duty cycle
        clk_i2c <= 1;
    else
        clk_i2c <= 0;

    // clock for ack when SCL==1(HIGH) (*ACK condition checked when clk is high)
    if(clk_prs == 62)
        ack_en <= 1;
    else
        ack_en <= 0;

    // clock for data when SCL==0(LOW) (*data should only change when clk is low)
    if(clk_prs == 187)
        clk_en <= 1;
    else
        clk_en <= 0;

end

```

```

always_ff@(posedge clk) begin

    if(i2c_clk_en == 1)
        i2c_scl <= clk_i2c;
    else
        i2c_scl <= 1;

    //check for ACK on SCL == 1(HIGH)

    if(ack_en == 1) begin
        unique case (i2c_state)
            s3: begin //get ACK

                if(i2c_sda == 0) begin
                    i2c_state <= s4; //ACK
                    data_index <= 15;
                end
            else begin
                i2c_clk_en <= 0;
                i2c_state <= s0; //not ACK
            end
        end

        s5: begin

            if(i2c_sda == 0) begin
                i2c_state <= s6;
                data_index <= 7;
            end
            else begin
                i2c_state <= s0;
                i2c_clk_en <= 0;
            end
        end
    end
end

```

```

        end

        s7: begin

            if(i2c_sda == 0)
                i2c_state <= s8;
            else begin
                i2c_state <= s0;
                i2c_clk_en <= 0;
            end
        end
    end
endcase

end

//check for data tranfer on SCL==0 (LOW)
if(clk_en == 1) begin
    unique case (i2c_state)
        s0: begin //standby
            i2c_sda <= 1;
            i2c_busy <= 0;
            i2c_done <= 0;

            if(i2c_send_flag == 1) begin
                i2c_state <= s1;
                i2c_busy <= 1;
            end
        end
    end

    s1: begin //start
        i2c_sda <= 0;
        i2c_state <= s2;
        data_index <= 7;
    end
end

```

```

s2: begin // send address

    i2c_clk_en <= 1; //start clocking i2c_scl

    if(data_index > 0) begin
        data_index <= data_index-1;
        i2c_sda <= i2c_addr[data_index];
    end

    else begin

        i2c_sda <= i2c_addr[data_index];
        get_ack <= 1;
    end

    if(get_ack == 1) begin
        get_ack <= 0;
        i2c_state <= s3;
        i2c_sda <= 'Z';
    end

end

s4: begin //send 1st 8 bits

    if(data_index > 8) begin
        data_index <= data_index - 1;
        i2c_sda <= i2c_data[data_index];
    end

    else begin

        i2c_sda <= i2c_data[data_index];
        get_ack <= 1;
    end

    if(get_ack == 1) begin
        get_ack <= 0;

```

```

        i2c_state <= s5;
        i2c_sda <= 'Z';

    end

end

s6: begin //send 2nd 8 bits

    if(data_index > 0) begin
        data_index <= data_index - 1;
        i2c_sda <= i2c_data[data_index];
    end

    else begin
        i2c_sda <= i2c_data[data_index];
        get_ack <= 1;
    end

    if(get_ack == 1) begin
        get_ack <= 0;
        i2c_state <= s7;
        i2c_sda <= 'Z';
    end

end

s8: begin //stop

    i2c_clk_en <= 0;
    i2c_sda <= 0;
    i2c_state <= s0;
    i2c_done <= 1;

end

endcase;

end

end

```

```
endmodule
```

Nios II Main C File

```
#include "system.h" // peripheral base addresses i.e. platform designer
component addresses
#include "i2c_opencores.h" //contains I2C_init, etc. functions
#include "altera_up_avalon_audio.h"
#include "altera_avalon_pio_regs.h"
#include "unistd.h" /* for usleep() */

#define WM8731_ADDRESS 0x1A // WM8731 chip address on I2C
bus
#define WM8731_REG_LLINE_IN 0x00 // Left Channel Line Input
Volume Control
#define WM8731_REG_RLINE_IN 0x01 // Right Channel Line Input
Volume Control
#define WM8731_REG_LHPHONE_OUT 0x02 // Left Channel Headphone
Output Volume Control
#define WM8731_REG_RHPHONE_OUT 0x03 // Right Channel Headphone
Output Volume Control
#define WM8731_REG_ANALOG_PATH 0x04 // Analog Audio Path Control
#define WM8731_REG_DIGITAL_PATH 0x05 // Digital Audio Path Control
#define WM8731_REG_PDOWN_CTRL 0x06 // Power Down Control Register
#define WM8731_REG_DIGITAL_IF 0x07 // Digital Audio Interface
Format
#define WM8731_REG_SAMPLING_CTRL 0x08 // Sampling Control Register
#define WM8731_REG_ACTIVE_CTRL 0x09 // Active Control
#define WM8731_REG_RESET 0x0F // Reset register

#define _WM8731_left_lineIn 0xF1 //disable left line input mute to
ADC,LLine input vol=0dB
#define _WM8731_Right_lineIn 0xF2
#define _WM8731_Left_hp 0x7F // Headphone settings : 6dB output
#define _WM8731_Right_hp 0x7F // Headphone settings : 6dB output
#define _WM8731_AnalogAudio 0x31
#define _WM8731_DigitalAudio 0x00 //disable dac softmute
#define _WM8731_power 0x00 // Disable Power down
#define _WM8731_DAIF 0x81 //slave mode, left-justified
mode,inverted bitclock
#define _WM8731_Sampling 0x00 // normal mode, 256fs
#define _WM8731_Activate 0x01 // Module is ON
#define _WM8731_Deactivate 0x00 // Module is OFF
#define _WM8731_Reset 0x00 // Reset value

void WM8731_CMD(unsigned int address, unsigned int cmd){
    unsigned int addr; //register address
    unsigned int cmdMSB = 0x100; //bit-mask 100000000b
    // B[15:9] Are Control Address Bits
    // B[8:0] Are Control Data Bits
    addr = address << 1; // Shift left for one positions
    // addr = addr | (hi(cmd) & 1);

    cmdMSB = cmd&cmdMSB; //extract MSB of 9-bit data
    cmdMSB = cmdMSB >> 8;
```

```

        addr = addr|cmdMSB; //insert as LSB of byte containing left-shifted
7-bit reg address

        //remove MSB (bit[8]) from cmd
        cmd = cmd & ~cmdMSB;

int nack=1;
while(nack==1){
alt_printf("i2c_start slave write...");
nack=I2C_start(I2C_OPENCORES_0_BASE,WM8731_ADDRESS,0);//slave addr
alt_printf("nack = %x\n",nack);
alt_printf("i2c_write reg addr %xh...",addr);
nack=I2C_write(I2C_OPENCORES_0_BASE,addr,0);//register addr 0001111
alt_printf("nack = %x\n",nack);
alt_printf("i2c_write data...");
nack=I2C_write(I2C_OPENCORES_0_BASE,cmd,1);//configure register bits
alt_printf("nack = %x\n",nack);
}

}

int main(){
    usleep(100000);

    //Initialise I2C master controller at 100kHz, 50MHz Cpu clock.
    I2C_init(I2C_OPENCORES_0_BASE,ALT_CPU_FREQ,100000);
    alt_printf("I2C master initialized.\n");

    alt_printf("reset register:\n");
    WM8731_CMD(WM8731_REG_RESET,0x00);//write 0 to reset

    alt_printf("power down control register:\n");
    WM8731_CMD(WM8731_REG_PDOWN_CTRL,0x10);//disable power down options
except bit5 (OUTPD)

        // WM8731_CMD(WM8731_REG_RESET, _WM8731_Reset);           // Reset
module
        WM8731_CMD(WM8731_REG_LLINE_IN, _WM8731_left_lineIn);     // Left
line in settings
        WM8731_CMD(WM8731_REG_RLINE_IN, _WM8731_Right_lineIn);  // Righth
line in settings
        WM8731_CMD(WM8731_REG_LHPHONE_OUT, _WM8731_Left_hp);     // Left
headphone out settings
        WM8731_CMD(WM8731_REG_RHPHONE_OUT, _WM8731_Right_hp);   // Right
headphone out settings
        WM8731_CMD(WM8731_REG_ANALOG_PATH, _WM8731_AnalogAudio); // Analog
paths
        WM8731_CMD(WM8731_REG_DIGITAL_PATH, _WM8731_DigitalAudio); // Digital
paths
        WM8731_CMD(WM8731_REG_DIGITAL_IF, _WM8731_DAIF);        // Digital
interface
        WM8731_CMD(WM8731_REG_SAMPLING_CTRL, _WM8731_Sampling); //
Sampling control
        WM8731_CMD(WM8731_REG_ACTIVE_CTRL, _WM8731_Activate);   //
Activate module

```

```
        //alt_printf("power down control register:\n");
        WM8731_CMD(WM8731_REG_PDOWN_CTRL,0x00);//disable all power down options
including OUTPD

        return 0;
}
```



```

lab4_2 u0 (
    .clk_clk (CLOCK_50),
    .reset_reset_n (KEY[0]),
    .i2c_opencores_0_export_scl_pad_io(SCL),
    .i2c_opencores_0_export_sda_pad_io(SDA),

    // .clk_clk(CLOCK_50), //pll 50MHz input
    .clock_12_clk(clock_12), //pll 12MHz output

    .audio_0_external_interface_ADCCDAT(ADCCDAT), //input to
audio core
    .audio_0_external_interface_ADCLRCK(ADCL), //input to audio
core

    .audio_0_external_interface_BCLK(SCK), //input
    .audio_0_external_interface_DACDAT(DACDAT), //output
    .audio_0_external_interface_DACL_RCK(DACL), //input
    .lrclk_lrclk(ADCL),
    //12.288MHz bitclock generated by audio clock module
    .clock_bridge_0_out_clk_1_clk(SCK),
    //PIO regs
    .led_export(LED)
);

irpulse irpulse0 (
    .clk(CLOCK_50),
    .clk38k(clk38k)
);
irreceiver irr0 (
    .irrow(irrow),
    .ircol(ircol),
    // .LED(LED),
    .clk(CLOCK_50)
);

```

endmodule

```

//audgen.sv
//generates left-right clock
//N=12MHz/48kHz = 250 <--prescale factor
//N=12.288MHz/48kHz=256
//for 50% (1/2) duty cycle: n>=(N-N/2)

```

```

module audgen
    #(N=256)
    (
        output logic lrclk, resetn, //sclk, resetn
        input logic reset, clk ); //12MHz reference clock bclk

    logic lrclk_next;
    logic [$clog2(N):0] n=0, n_next;

    always_comb begin

        n_next = n;
        //clkfs_next = clkfs;
        if(reset) begin
            n_next=N-1;

```

```

        lrclk_next=0;
    end

    if(n<=0) begin //if countdown finished
        n_next=N-1; //reset count
        lrclk_next=0; //indicate start of new sample
    end
    else begin
        n_next=n-1; //else decrement count
        lrclk_next = lrclk; //not start of new sample yet
    end

        //1/2 duty cycle:
    if(n >= (N-N/2))
        lrclk_next=0; //low
    else if(n <= N-N/2)
        lrclk_next=1; //high

    end

    always_ff @(posedge clk) begin //lrclk must change on falling edge of
bclk, will configure wm8731 to use inverted bclk instead
        n <= n_next;
        //clkfs <= clkfs_next;
        lrclk <= lrclk_next;
    end

    assign resetn = ~reset ;

endmodule

```

References

AntonZero. (n.d.). *WM8731-Audio-codec-on-DE10Standard-FPGA-board*. Retrieved 04 12, 2018, from GitHub: <https://github.com/AntonZero>

Vigil, B. M.-P. (2015). *FPGA based spectrum analyzer*. CASE.