EE2026 FPGA Design Project

Rishab Patwari A0184456W
Raghav Bhardwaj A0184445Y

Key Features

No.	Feature	Owner	FPGA inputs used	Feature Description	User Manual S/N
1.	Basic Voice Visualizer	Team	sw[0]	When sw[0] = 0, the VGA displays a voice waveform; When sw[0] = 1, the VGA displays a static ramp wave.	
2.A	Volume Indicator	Rishab	N.A.	{led[11:0]} are used to display the current volume (based on amplitude) of the voice waveform. From right to left, at higher volumes, more leds will light up. When the sound is too soft no LED will light up. {an[3:0], seg[7:0]} are used to display the current volume on the seven segment display as an integer from 0 - 12. Volume indicated on the display will match the volume indicated by the LEDs. The volume indicator works comparing maximum amplitude over 0.67 seconds against volume thresholds. The max amplitude is measured by sampling the waveform at CLK speed for 0.67 seconds and then comparing it against a series of threshold numbers corresponding to the different volumes. Then the combination of leds or the anode/segments corresponding to the volume number are turned on as output for display. Please refer to the Appendix for annotated code on how this was implemented.	1
2.B	VGA Display	Raghav	BTNL, BTNR, sw[0], sw[1], sw[2], sw[3], sw[15], sw[14]	The VGA display consists of a screen displaying the waveform of the current microphone reading in real time. The user can cycle through the 5 different colour schemes using BTNL and BTNR. Single pulse circuits (learnt in lab 4/assignment 4) were used to enable button input debouncing and prevent unwanted inputs to be registered by the user holding down the button. The user can also switch on and off individual graph features using the following: - sw[0]: waveform - sw[1]: grid - sw[2]: axes - sw[3]: ticks Additionally, the user can pause the real time microphone reading using sw[14] and can switch to a test waveform using sw[15]. Please refer to the Appendix for more information on how the theme selection and wave display was implemented.	2 - 5
3.	Improvement 1: Frequency Measure	Rishab	sw[12]	When sw[12] = 0, the seven segment display displays the volume of the voice waveform. When sw[12] = 1, the seven segment displays the frequency of the voice waveform. The accurate range of the frequency measurement is between 400Hz and 10kHz. The uncertainty in measurement is ±10Hz. The calculation process involves recording and storing the number of peaks and troughs in the sinusoidal waveform over a period of 0.5 second. 1) At a sampling frequency of 20kHz, 10000 samples are taken of the incoming sound wave and stored into a array. 2) Then, the value of each element in the array is compared to the previous and next element so as to obtain the count of the number of peaks and troughs. 3) Frequency is calculated using the formula: Frequency = (number of peaks + troughs). This formula differs from the normal frequency formula because both peaks and troughs were used so that the audio sample recording duration required could be minimised to 0.5 seconds while avoiding having to use the timewise costly division function in our calculation, if I were to calculate the average number of peaks. Please refer to the Appendix for annotated code describing how this was implemented.	7
4.	Improvement 2: Music Visualizer	Raghav	BTNC	The music visualiser consists of a bar graph-type visualisation that is based on the amplitude of the incoming sound input into the microphone. It also has 4 moving balls on the screen which react to the 13 volume levels of the sound input and change their velocity based on the sound.	6

				Collision detection was also enabled between the 4 balls and also with the edge of the screen, causing the balls to rebound and change direction once they experience a collision. Additionally, based on the volume, the bar graph and the balls cycle through a spectrum of 13 colours,making the visualisation more colourful and appealing. The position of the balls can be reset/randomized by the user at any time using BTNC. One big issue in the way of displaying multiple objects of different colours had to be addressed first.This has been elaborated below: One issue that was present in the initial version of the VGA_DISPLAY.v code given to us was that the different RGB channels (for example, the background and waveform channel) used a bitwise OR operator to set the value of VGA_RED, VGA_GREEN, and VGA_BLUE. This caused an issue illustrated in the following example: Suppose the ball colour was black, i.e. 12b'0000 0000 0000, and the background colour was white, i.e. 12b'1111 1111 1111. The bitwise OR operator on these numbers would result in 12b'1111 1111 1111. This causes the black wave to not appear on the white background - causing the user to just see a plain white background. This issue was especially problematic when we wanted to create black text boxes and wanted to add more complex features, for example with the game. To circumvent this, I designed a method to draw features in a hierarchy of layers, which removes the conflicts caused by the bitwise OR. I essentially added 1 bit in front of each colour, this bit represents whether I would like this particular layer to be displayed - If 1, then the layer will be displayed, and if 0 the layer will not be displayed and the program will look to the underlying layer and see if that needs to be displayed. This caused the wire that represents a colour to become 15 bits and look something like this (for the colour red): 15'b1 0000 1 0000 1 1111. This colour format was then brought to the rest of the modules as well, leading to an overhaul in the visual accuracy and quality of our display. Please refer to the Appendix for more information on how this was implemented	
5.	Improvement 3: Flappy-bird-esque Game	Team	BTNC	The objective of this game is to get a bird across the across the screen using your voice or loud noises as the trigger to make the bird jump. However we didn't want to set a predefined level for the player and instead chose to "randomize" the level by recording the noise in the environment before starting the level, and by using peaks from this recording as the pillars that the bird must jump over. We wanted the player to be able to lose, and so we gave the player three lives, which are indicated by the HP bar on the top right corner of the screen. Every time the bird crashed into a pillar, the HP would be reduced, until the HP ran out and it was game over. The user could then try again albeit with a different level - hence ensuring that every level felt different and repetitiveness was kept to a minimum. To start the game, the user presses BTNC, which then records sound, and prompts the user to press BTNC once again to start the game.	8

Feedback: What did you like most/ least about the project?

Most Like:

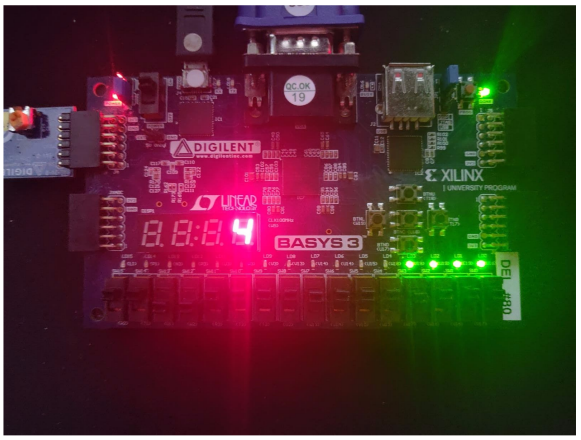
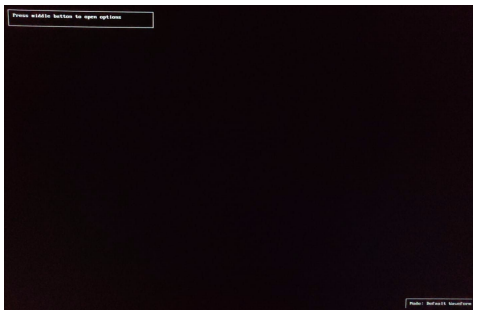
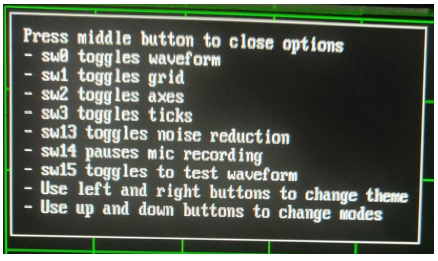
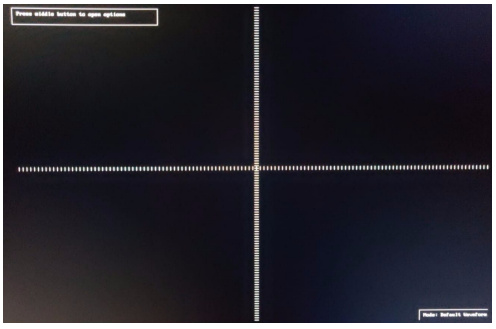
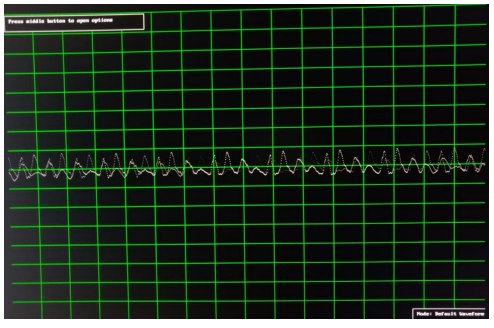
- Witnessing what we learnt in theory coming to life especially with the **visualization** really boosted our self confidence and also our passion for taking on Computer Engineering. We feel like we can actually create meaningful technical solutions that are relevant to the real world.

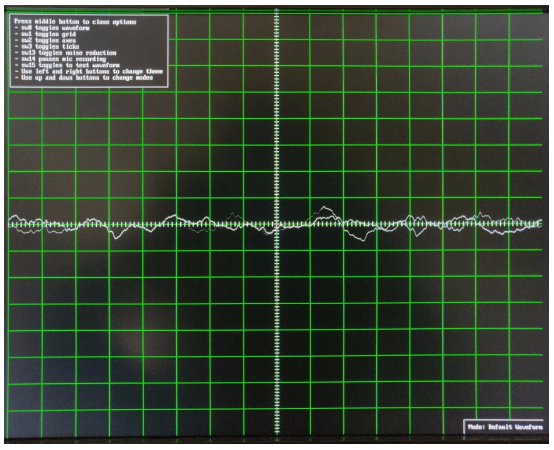
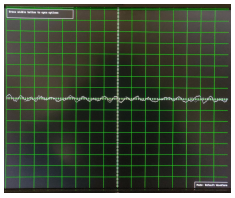
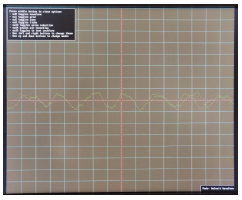
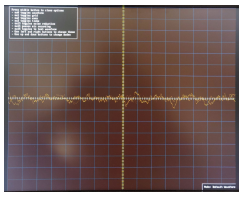
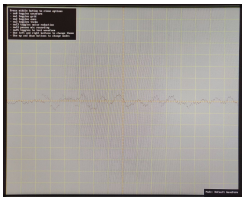
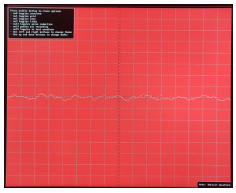
- The time challenge given to us to do this project, forced us to react and collaborate in sharing ideas and concepts that could help each other. For instance, we never thought that we could could come up with a game let alone a frequency calculator after doing simply lab 4, and honestly the past year projects seemed intimidating. However, though different groups pursued different ideas, as a cohort we would start sharing our challenges and discussing possible solutions. For instance when we were figuring out how to display text on the screen using the libraries, we would share our understanding of the code and links between the different groups. Similarly for the Frequency calculation, some groups used complicated noise cancelling approaches to get unfavourable readings, and were able to warn the other groups from going down the same path. The project bringing about healthy collaboration among the different teams is something really interesting that I do not really see in other modules.

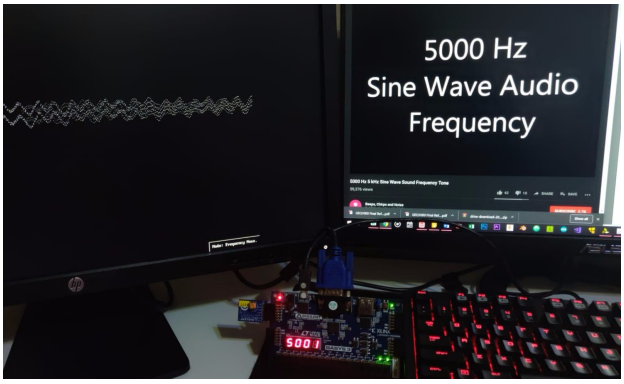
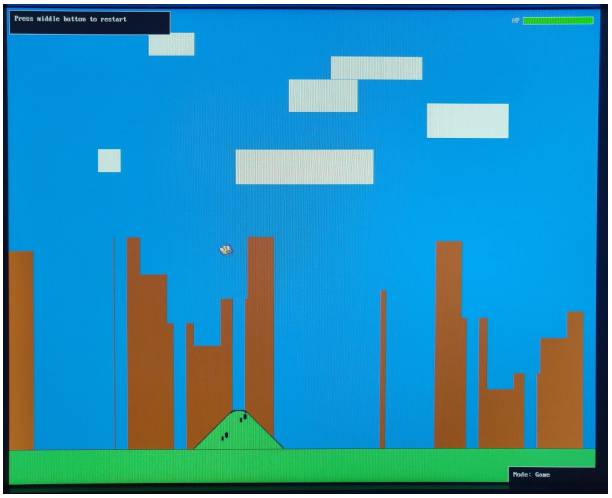
Least Like:

- We spent a lot of time simply figuring out how to use the common libraries like the text libraries available online. If this could have been introduced to us in a lab, it would have saved us a lot of time spent on the project, while juggling with workload of other modules

User Manual

S/N	Display	Description
1	BASYS BOARD VISUALIZATION : VOLUME INDICATOR 	**Volume Indicator** The volume of the voice waveform is visualized using the 12 leds from right to left on the basys board and the 7-seg display. The volume ranges from 0 - 12, 0 denoting a volume which is too soft, while 12 indicating a loud sound. The leds[11:0] light up based on the volume from right to left. The 7-seg display indicates the corresponding volume matching the leds lighting up
2		**Start Screen** Upon switching on our Basys Board, with all the switches (sw[15:0]) OFF, you will see a blank screen as seen on the left. We have an onscreen menu option to help you navigate through our system which is toggled on by pressing the **MIDDLE BUTTON** on the basys board. On the bottom right, the current mode being viewed is displayed.
3	 Menu Options  Enable Ticks  Enable Grid and Waveform	**Toggling Visualization of Waveform and Background** As seen from the menu options, you can toggle different visualization features by toggling the switches **sw[0:15]**. You can enable multiple switches at the same time to display multiple components simultaneously. The switches control the following: sw[0] : waveform sw[1] : 16x16 onscreen grid sw[2] : x and y axes sw[3] : ticks along the axes There are also additional features that allow further control of the waveform sw[13] : noise reduced wave (using moving average filter) sw[14] : freezes the waveform sw[15] : toggles between ramp wave and normal wave

4	BASIC VOICE VISUALIZER 	**Basic Voice Visualizer** There are in total 3 waveforms that can be visualized on the screen. 1. Raw Unfiltered Voice Waveform 2. Static Ramp Wave 3. Moving Average Filter Applied Waveform The Raw unfiltered Voice Waveform is the real time display of the input sound sample The Noise Cancelled Waveform applies a moving average filter helps to reduce noise and generate a more consistent waveform.
5	 1: Default  2: Greens and Blues  3: Day & Night  4: Modern & Crisp  5: Fun & Tropical	**Toggling Background Themes** We have 5 theme sets that change the following: • waveform colour • background colour • axes colour • grid colour • ticks colour Switching between the themes is done by pressing controlling the left and right buttons on the Basys board: • Pressing the right button will cycle to next increment in the sequence • Pressing the left button will cycle to the previous theme in the sequence.
6		**Music Visualizer** The music visualizer uses a bar graph and 4 moving balls to visualise any music (sound) input. The visualization is made more appealing by having each feature cycle through a spectrum of 13 colours. Each feature starts on a different point in the spectrum, thereby ensuring that each feature contrasts from the rest as the colours change based on the volume level. The user can interact with this visualization by pressing the middle button (BTNC) to reset/randomize the starting position of these balls.

7	BASYS BOARD VISUALIZATION : FREQUENCY  	**Frequency Measure** The basys board can measure frequency of the voice waveform and display the frequency on the 7-segment display. The frequency range that is accurately measured is between 400 Hz and 10khz. The accuracy of the measurement is very high (usually accurate to ±1 Hz, however the volume of the sound signal needs to be sufficiently high and the ambient noise in the environment need to be minimal. Our tests show the optimal volume to be volume level 4 and above to get a good, reliable frequency reading.
8	Recording sound to generate level:  Playing through the level as the bird  	**Flappy-bird-esque Game** To start the game, the user simply has to press the middle button (BTNC). This causes the game to record sound for 5 seconds, which it uses to generate the level that the user will soon play through. Once the recording is completed, the game prompts the user to start the game, by pressing BTNC. The user has 3 lives, represented by an HP bar in the top right corner of the screen. To win the game, the user has to reach the other side of the screen without hitting any of the generated pillars - all within these 3 lives. The user controls the bird by causing it to jump by producing any kind of loud noise (above volume 8). The game can reset at any point using BTNC.

Appendix

Commented Code for Volume Indicator

```
27      output  [7:0]  seg,
28      output reg [4:0] volumeval,
29      output reg [11:0] max_value = 0
30      );
31
32      //a 25 bit counter to allow timing upto 0.67 seconds
33      reg [24:0] counter = 0;
34      always @(posedge CLK) begin
35          if (counter[24] != 1) counter <= counter + 1;
36          else counter <= 0;
37      end
38
39      //At each clock cycle, the current max_amplitude is compared against current sound_sample and updated
40      always @(posedge CLK) begin
41          if (sound_sample > max_value) begin
42              max_value[11:0] <= sound_sample[11:0];
43          end
44          if (counter[24] == 1) begin
45             max_value <= 0;
46          end
47      end
48
49      //The max_value at the end of 0.67 seconds is compared against the thresholds corresponding to the correct volume
50      always @(posedge counter[24]) begin
51          if      (max_value >= 3980) begin block_sample <= 12'b111111111111;  volumeval <= 12; end
52          else if (max_value >= 3800) begin block_sample <= 12'b011111111111;  volumeval <= 11; end
53          else if (max_value >= 3620) begin block_sample <= 12'b001111111111;  volumeval <= 10; end
54          else if (max_value >= 3440) begin block_sample <= 12'b000111111111;  volumeval <= 9; end
55          else if (max_value >= 3260) begin block_sample <= 12'b000011111111;  volumeval <= 8; end
56          else if (max_value >= 3080) begin block_sample <= 12'b000001111111;  volumeval <= 7; end
57          else if (max_value >= 2900) begin block_sample <= 12'b000000111111;  volumeval <= 6; end
58          else if (max_value >= 2720) begin block_sample <= 12'b000000011111;  volumeval <= 5; end
59          else if (max_value >= 2540) begin block_sample <= 12'b000000001111;  volumeval <= 4; end
60          else if (max_value >= 2360) begin block_sample <= 12'b000000000111;  volumeval <= 3; end
61          else if (max_value >= 2180) begin block_sample <= 12'b000000000011;  volumeval <= 2; end
62          else if (max_value >= 2080) begin block_sample <= 12'b000000000001;  volumeval <= 1; end
63          else                        begin block_sample <= 0;  volumeval <= 0; end
64      end
```

Commented Code for Theme Selection

```
23  module theme_selector(
24          input button_clock,
25          input left_button_out,
26          input right_button_out,
27          // Registers below contain the concatenated data of the colour for each feature.
28          output reg [11:0] cur_theme_wave,
29          output reg [11:0] cur_theme_axes,
30          output reg [11:0] cur_theme_grid,
31          output reg [11:0] cur_theme_tick,
32          output reg [11:0] cur_theme_background,
33          input [1:0] mode
34      );
35
36      // reg array used to temporarily store colours.
37      reg [11:0] cur_theme[0:4];
38      // Counter below used to cycle through the different themes.
39      reg [2:0] theme_counter = 0; // This value goes from 0 to a max of 4.
40
41      always @ (posedge button_clock) begin
42          if (mode == 0) begin
43              if (left_button_out == 1 && right_button_out == 0) begin
44                  theme_counter = theme_counter - 1;
45              end
46              else if (right_button_out == 1 && left_button_out == 0) begin
47                  theme_counter = theme_counter + 1;
48              end
49
50              // Conditional to handle unwanted overflow, since we are only using
51              // 5 integers out of the available 8 integers in 3 bits.
52              if (theme_counter == 5) theme_counter = 0;
53              if (theme_counter == 7) theme_counter = 4;
54
55              // Switch statement below assigns colours to corresponding bits in the
56              //  cur_theme array.
57              case (theme_counter)
58                  0: begin //Default
59                      // Wave
60                      cur_theme[0][3:0] = 4'hf;
61                      cur_theme[0][7:4] = 4'hf;
62                      cur_theme[0][11:8] = 4'hf;
63                      // Axes
64                      cur_theme[1][3:0] = 4'h0;
65                      cur_theme[1][7:4] = 4'hD;
66                      cur_theme[1][11:8] = 4'h0;
67                      // Grid
68                      cur_theme[2][3:0] = 4'h0;
69                      cur_theme[2][7:4] = 4'hd;
70                      cur_theme[2][11:8] = 4'h0;
71                      // Tick
72                      cur_theme[3][3:0] = 4'hf;
73                      cur_theme[3][7:4] = 4'hf;
74                      cur_theme[3][11:8] = 4'hf;
75                      //Background
76                      cur_theme[4][3:0] = 4'h0;
77                      cur_theme[4][7:4] = 4'h0;
78                      cur_theme[4][11:8] = 4'h0;
79                  end
80                  1: begin //Playful greens and blues
81                      // Wave
82                      cur_theme[0][3:0] = 4'h8;
83                      cur_theme[0][7:4] = 4'ha;
84                      cur_theme[0][11:8] = 4'h4;
85                      // Axes
86                      cur_theme[1][3:0] = 4'h7;
87                      cur_theme[1][7:4] = 4'ha;
88                      cur_theme[1][11:8] = 4'ha;
89                      // Grid
90                      cur_theme[2][3:0] = 4'h7;
91                      cur_theme[2][7:4] = 4'ha;
92                      cur_theme[2][11:8] = 4'ha;
93                      // Tick
94                      cur_theme[3][3:0] = 4'hc;
95                      cur_theme[3][7:4] = 4'h0;
96                      cur_theme[3][11:8] = 4'h0;
97                      //Background
98                      cur_theme[4][3:0] = 4'h3;
99                      cur_theme[4][7:4] = 4'h4;
100                     cur_theme[4][11:8] = 4'h5;
101                 end
```

```
                2: begin //Day and Night
                    // Wave
                    cur_theme[0][3:0] = 4'hf;
                    cur_theme[0][7:4] = 4'h8;
                    cur_theme[0][11:8] = 4'h0;
                    // Axes
                    cur_theme[1][3:0] = 4'h0;
                    cur_theme[1][7:4] = 4'h3;
                    cur_theme[1][11:8] = 4'h5;
                    // Grid
                    cur_theme[2][3:0] = 4'h0;
                    cur_theme[2][7:4] = 4'h3;
                    cur_theme[2][11:8] = 4'h5;
                    // Tick
                    cur_theme[3][3:0] = 4'he;
                    cur_theme[3][7:4] = 4'hd;
                    cur_theme[3][11:8] = 4'h4;
                    //Background
                    cur_theme[4][3:0] = 4'h0;
                    cur_theme[4][7:4] = 4'h1;
                    cur_theme[4][11:8] = 4'h2;
                end
                3: begin //Modern and crisp
                    // Wave
                    cur_theme[0][3:0] = 4'h7;
                    cur_theme[0][7:4] = 4'h2;
                    cur_theme[0][11:8] = 4'h0;
                    // Axes
                    cur_theme[1][3:0] = 4'hc;
                    cur_theme[1][7:4] = 4'hd;
                    cur_theme[1][11:8] = 4'h6;
                    // Grid
                    cur_theme[2][3:0] = 4'hc;
                    cur_theme[2][7:4] = 4'hd;
                    cur_theme[2][11:8] = 4'h6;
                    // Tick
                    cur_theme[3][3:0] = 4'hd;
                    cur_theme[3][7:4] = 4'h7;
                    cur_theme[3][11:8] = 4'h2;
                    //Background
                    cur_theme[4][3:0] = 4'ha;
                    cur_theme[4][7:4] = 4'ha;
                    cur_theme[4][11:8] = 4'ha;
                end
                4: begin //Fun and tropical
                    // Wave
                    cur_theme[0][3:0] = 4'hf;
                    cur_theme[0][7:4] = 4'hf;
                    cur_theme[0][11:8] = 4'hf;
                    // Axes
                    cur_theme[1][3:0] = 4'h5;
                    cur_theme[1][7:4] = 4'ha;
                    cur_theme[1][11:8] = 4'ha;
                    // Grid
                    cur_theme[2][3:0] = 4'h5;
                    cur_theme[2][7:4] = 4'ha;
                    cur_theme[2][11:8] = 4'ha;
                    // Tick
                    cur_theme[3][3:0] = 4'h0;
                    cur_theme[3][7:4] = 4'h0;
                    cur_theme[3][11:8] = 4'h0;
                    //Background
                    cur_theme[4][3:0] = 4'h4;
                    cur_theme[4][7:4] = 4'h4;
                    cur_theme[4][11:8] = 4'hf;
                end
            endcase

            // Each component is then set to the appropriate element in the
            // cur_theme array.
            cur_theme_wave = cur_theme[0];
            cur_theme_axes = cur_theme[1];
            cur_theme_grid = cur_theme[2];
            cur_theme_tick = cur_theme[3];
            cur_theme_background = cur_theme[4];
        end
    end
endmodule
```

Commented Code for Frequency Calculation

```
//////////////////////////////////////////////////////////////////////////////////

module frequency_calculator(
    input CLK,
    input freq_20kHz,
    input [11:0] sound_sample,
    output reg [18:0] frequency = 0,
    output  [3:0] an,
    output  [7:0] seg
    );

   //This block stores the sample data for 1 second and determines the maximum and minimum values of the sample
   reg[15:0] i = 0;
   reg[15:0] i_max = 0;
   reg[15:0] max_amplitude = 0;
   reg[15:0] min_amplitude = 2048;
   reg[11:0] one_second_record[0:10000];
    reg[15:0] time_counter = 0; //If time_counter == 10000, termination condition (0.5 second completed)
   reg ender = 0;
   reg [15:0] max_counter = 0;
   reg [15:0] min_counter = 0;

  always @(posedge freq_20kHz) begin
        //This block stores recorded sample and its max, min amplitudes
        if (ender == 0) begin
            //This line resets the counter when it hits 10000 (time = 0.5 s) and toogled boolean register ender to 1
            if (time_counter == 10000) begin ender <= 1; time_counter <= 0; i_max <= i; i <= 0; end
            else begin
            time_counter <= time_counter + 1;
            //This line stores the sound sample record for 0.1 seconds
            one_second_record[i] <= sound_sample;
            i <= i + 1;

            //These lines store the maximum and minium amplitudes recorded
            if (sound_sample >= max_amplitude) max_amplitude <= sound_sample;
            if (sound_sample <= min_amplitude) min_amplitude <= sound_sample;
            end

        end
        //This block counts the number of peaks and troughs in 0.5 second
        //The frequency (number_peaks/troughs in 1 s)= 2*(number_peaks + number_troughs)/2 = (number_peaks + number_troughs)
        //This block iterates thorugh the recorded sample and compares adjacent elements to determine peak and trough
         if (ender == 1) begin
            i <= i + 1;

            //These lines count the number of peaks and troughs in recorded sample
            if ((one_second_record[i-1] < one_second_record[i]) && (one_second_record[i+1] < one_second_record[i])) max_counter
            if ((one_second_record[i-1] > one_second_record[i]) && (one_second_record[i+1] > one_second_record[i])) min_counter

            //Termination condition
            if (i == i_max) begin

                ender <= 0;
                frequency <= (min_counter + max_counter);
                max_counter <= 0;
                min_counter <= 0;
                max_amplitude <= 0;
                min_amplitude <= 0;
                i_max <= 0;
                i <= 0;

            end
        end
end

    wire freq_1525Hz;
    wire [3:0] interim_an;
    clk_div_1525kHz my_1525kHz(CLK, freq_1525Hz);
    FLICKER_ANODE fa(freq_20kHz, interim_an); //Updates the anode configuration at 1525kHz

    wire[7:0] interim_seg;
    SEGMENT_DISP sd (freq_20kHz,frequency,an,interim_seg);
    assign an = interim_an;
    assign seg = interim_seg;

endmodule
```

Commented Code for Music Visualizer

```
22  module ball_bounce (
23      input clk,
24      input freq_20kHz,
25      input [10:0] wave_sample,
26
27      input middle_button_out,
28      input button_clock,
29
30      input [3:0] volume,
31      input [11:0] VGA_HORZ_COORD,
32      input [11:0] VGA_VERT_COORD,
33
34      output [14:0] VGA_ball_colour,
35      output [14:0] VGA_ball_waveform
36      );
37
38      // Chunk of code used to generate the bar graph based on the wave input.
39      reg [43:0] ball_start_pos = 0;
40      reg [9:0] sample_memory[1279:0];
41      reg [10:0] k = 0;
42      always @ (posedge freq_20kHz) begin
43          if (k == 1279) begin
44              k = 0;
45          end
46          else k = k + 1;
47          sample_memory[k] <= wave_sample[10:1];
48      end
49      wire wave_condition = (VGA_HORZ_COORD % 40 >= 0 && VGA_HORZ_COORD % 40 <= 33) &&
50          (VGA_VERT_COORD >= 1024 - sample_memory[VGA_HORZ_COORD]);
51
52      // Storing an array of a spectrum of 12-bit colours.
53      reg [0:11] ball_colours [0:12];
54      initial begin
55          ball_colours[0] = 12'h0ff;
56          ball_colours[1] = 12'h09f;
57          ball_colours[2] = 12'h04f;
58          ball_colours[3] = 12'h60f;
59          ball_colours[4] = 12'hf0f;
60          ball_colours[5] = 12'hf05;
61          ball_colours[6] = 12'hf00;
62          ball_colours[7] = 12'hf60;
63          ball_colours[8] = 12'hfe0;
64          ball_colours[9] = 12'h3f0;
65          ball_colours[10] = 12'h0f8;
66          ball_colours[11] = 12'h0fd;
67          ball_colours[12] = 12'h0ff;
68      end
69      // These wires will be later used to access elements of the array. Note the use
70      // of the modulo operation along with different added constants to ensure that
71      // none of the visualization features are of the same colour at any point in
72      // time.
73      wire [3:0] b1_colour = volume;
74      wire [3:0] b2_colour = (volume + 2 > 12) ? (volume + 2) % 13 : volume + 2;
75      wire [3:0] b3_colour = (volume + 4 > 12) ? (volume + 4) % 13 : volume + 4;
76      wire [3:0] b4_colour = (volume + 6 > 12) ? (volume + 6) % 13 : volume + 6;
77      wire [3:0] bar_colour = (volume + 8 > 12) ? (volume + 8) % 13 : volume + 8;
78
79      // Used to store the current x-y position of each ball.
80      // Bits [10:0] to represent x, and [21:11] to represent y.
81      reg [21:0] b1_pos;
82      reg [21:0] b2_pos;
83      reg [21:0] b3_pos;
84      reg [21:0] b4_pos;
85
86      // Used to store the direction vector that each ball is
87      // travelling in
88      reg [1:0] b1_dir = 2'b11;
89      reg [1:0] b2_dir = 2'b01;
90      reg [1:0] b3_dir = 2'b10;
91      reg [1:0] b4_dir = 2'b00; //1 is negative, 0 is positive
92
93      // Counter to update ball position.
94      reg [21:0] movement_counter = 0;
95
96      // This defines the boundary area which the balls should not exceed. This is
97      // later used for purposes of the collision detection.
```

```
    wire boundary_area = ((VGA_HORZ_COORD >= 0 && VGA_HORZ_COORD <= 1279) &&
        (VGA_VERT_COORD >= 0 && VGA_VERT_COORD <= 1023)) &&
        ~((VGA_HORZ_COORD >= 50 && VGA_HORZ_COORD <= 1229) &&
        (VGA_VERT_COORD >= 50 && VGA_VERT_COORD <= 973));

    // The following wires define the 4 balls (circles) and their start positions.
    wire b1_condition = ((VGA_HORZ_COORD - b1_pos[10:0]) * (VGA_HORZ_COORD - b1_pos[10:0])
        + (VGA_VERT_COORD - b1_pos[21:11]) * (VGA_VERT_COORD - b1_pos[21:11])) < 1800;

    wire b2_condition = ((VGA_HORZ_COORD - b2_pos[10:0]) * (VGA_HORZ_COORD - b2_pos[10:0])
        + (VGA_VERT_COORD - b2_pos[21:11]) * (VGA_VERT_COORD - b2_pos[21:11])) < 1800;

    wire b3_condition = ((VGA_HORZ_COORD - b3_pos[10:0]) * (VGA_HORZ_COORD - b3_pos[10:0])
        + (VGA_VERT_COORD - b3_pos[21:11]) * (VGA_VERT_COORD - b3_pos[21:11])) < 1800;

    wire b4_condition = ((VGA_HORZ_COORD - b4_pos[10:0]) * (VGA_HORZ_COORD - b4_pos[10:0])
        + (VGA_VERT_COORD - b4_pos[21:11]) * (VGA_VERT_COORD - b4_pos[21:11])) < 1800;

    // This is the collision detection. These wires become 1 when a ball's position conditional outputs 1
    // and if the boundary_area or any of the other ball conditionals output 1 as well for the same
    // VGA_HORZ_COORD and VGA_VERT_COORD.
    wire b1_collision = (b1_condition) & (boundary_area                | b2_condition | b3_condition | b4_condition);
    wire b2_collision = (b2_condition) & (boundary_area | b1_condition               | b3_condition | b4_condition);
    wire b3_collision = (b3_condition) & (boundary_area | b1_condition | b2_condition               | b4_condition);
    wire b4_collision = (b4_condition) & (boundary_area | b1_condition | b2_condition | b3_condition              );

    initial begin
        b1_pos[10:0] = 300;
        b1_pos[21:11] = 300;
        b2_pos[10:0] = 640;
        b2_pos[21:11] = 512;
        b3_pos[10:0] = 640;
        b3_pos[21:11] = 850;
        b4_pos[10:0] = 400;
        b4_pos[21:11] = 750;
    end
    reg button_pressed = 0;
    always @ (posedge button_clock) begin
        if (middle_button_out == 1) begin
            button_pressed = 1;
        end
        else button_pressed = 0;
    end
    always @ (posedge clk) begin
        if (movement_counter[21] == 1) begin
            if (button_pressed == 1) begin
                b1_pos[10:0] = 300;
                b1_pos[21:11] = 300;
                b2_pos[10:0] = 640;
                b2_pos[21:11] = 512;
                b3_pos[10:0] = 640;
                b3_pos[21:11] = 850;
                b4_pos[10:0] = 400;
                b4_pos[21:11] = 750;
            end
            else begin
                // The following chunks of code handle the volume
                // based movement of the ball.
                b1_pos[10:0] = b1_dir[1] ?
                    b1_pos[10:0] + (volume * 3)
                    : b1_pos[10:0] - (volume * 3);
                b1_pos[21:11] = b1_dir[0] ?
                    b1_pos[21:11] + (volume * 3)
                    : b1_pos[21:11] - (volume * 3);
                if (b1_pos[10:0] >= 1229) begin
                    b1_pos[10:0] = 1228;
                end
                if (b1_pos[21:11] >= 973) begin
                    b1_pos[21:11] = 972;
                end
                if (b1_pos[10:0] <= 50) begin
                    b1_pos[10:0] = 51;
                end
                if (b1_pos[21:11] <= 50) begin
                    b1_pos[21:11] = 51;
                end

                b2_pos[10:0] = b2_dir[1] ?
                    b2_pos[10:0] + (volume * 3)
```

```
                    : b2_pos[10:0] - (volume * 3);
                b2_pos[21:11] = b2_dir[0] ?
                    b2_pos[21:11] + (volume * 3)
                    : b2_pos[21:11] - (volume * 3);
                if (b2_pos[10:0] >= 1229) begin
                    b2_pos[10:0] = 1228;
                end
                if (b2_pos[21:11] >= 973) begin
                    b2_pos[21:11] = 972;
                end
                if (b2_pos[10:0] <= 50) begin
                    b2_pos[10:0] = 51;
                end
                if (b2_pos[21:11] <= 50) begin
                    b2_pos[21:11] = 51;
                end

                b3_pos[10:0] = b3_dir[1] ?
                    b3_pos[10:0] + (volume * 3)
                    : b3_pos[10:0] - (volume * 3);
                b3_pos[21:11] = b3_dir[0] ?
                    b3_pos[21:11] + (volume * 3)
                    : b3_pos[21:11] - (volume * 3);
                if (b3_pos[10:0] >= 1229) begin
                    b3_pos[10:0] = 1228;
                end
                if (b3_pos[21:11] >= 973) begin
                    b3_pos[21:11] = 972;
                end
                if (b3_pos[10:0] <= 50) begin
                    b3_pos[10:0] = 51;
                end
                if (b3_pos[21:11] <= 50) begin
                    b3_pos[21:11] = 51;
                end

                b4_pos[10:0] = b4_dir[1] ?
                    b4_pos[10:0] + (volume * 3)
                    : b4_pos[10:0] - (volume * 3);
                b4_pos[21:11] = b4_dir[0] ?
                    b4_pos[21:11] + (volume * 3)
                    : b4_pos[21:11] - (volume * 3);
                if (b4_pos[10:0] >= 1229) begin
                    b4_pos[10:0] = 1228;
                end
                if (b4_pos[21:11] >= 973) begin
                    b4_pos[21:11] = 972;
                end
                if (b4_pos[10:0] <= 50) begin
                    b4_pos[10:0] = 51;
                end
                if (b4_pos[21:11] <= 50) begin
                    b4_pos[21:11] = 51;
                end
            end
            movement_counter = 0;
        end
        else begin
            // The following conditionals detect any collisions between the balls
            // and cause the affected ball to invert its direction vector, causing
            // collided balls to begin moving in the opposite direction.
            if (b1_collision) begin
                b1_pos[10:0] = b1_dir[1] ?
                    b1_pos[10:0] - 3
                    : b1_pos[10:0] + 3;
                b1_pos[21:11] = b1_dir[0] ?
                    b1_pos[21:11] - 3
                    : b1_pos[21:11] + 3;
                b1_dir = ~b1_dir;
            end

            if (b2_collision) begin
                b2_pos[10:0] = b2_dir[1] ?
                    b2_pos[10:0] - 3
                    : b2_pos[10:0] + 3;
                b2_pos[21:11] = b2_dir[0] ?
                    b2_pos[21:11] - 3
                    : b2_pos[21:11] + 3;
                b2_dir = ~b2_dir;
            end
```

```
257
258             if (b3_collision) begin
259                 b3_pos[10:0] = b3_dir[1] ?
260                     b3_pos[10:0] - 3
261                     : b3_pos[10:0] + 3;
262                 b3_pos[21:11] = b3_dir[0] ?
263                     b3_pos[21:11] - 3
264                     : b3_pos[21:11] + 3;
265                 b3_dir = ~b3_dir;
266             end
267
268             if (b4_collision) begin
269                 b4_pos[10:0] = b4_dir[1] ?
270                     b4_pos[10:0] - 3
271                     : b4_pos[10:0] + 3;
272                 b4_pos[21:11] = b4_dir[0] ?
273                     b4_pos[21:11] - 3
274                     : b4_pos[21:11] + 3;
275                 b4_dir = ~b4_dir;
276             end
277         end
278         movement_counter = movement_counter + 1;
279     end
280
281     // Assign statements to output colours of all the features to the
282     assign VGA_ball_colour[4:0] = b1_condition ? {1'b1, ball_colours[b1_colour][0:3]}
283         : b2_condition ? {1'b1, ball_colours[b2_colour][0:3]}
284         : b3_condition ? {1'b1, ball_colours[b3_colour][0:3]}
285         : b4_condition ? {1'b1, ball_colours[b4_colour][0:3]}
286         : {1'b0, 4'h0};
287
288     assign VGA_ball_colour[9:5] = b1_condition ? {1'b1, ball_colours[b1_colour][4:7]}
289         : b2_condition ? {1'b1, ball_colours[b2_colour][4:7]}
290         : b3_condition ? {1'b1, ball_colours[b3_colour][4:7]}
291         : b4_condition ? {1'b1, ball_colours[b4_colour][4:7]}
292         : {1'b0, 4'h0};
293
294     assign VGA_ball_colour[14:10] = b1_condition ? {1'b1, ball_colours[b1_colour][8:11]}
295         : b2_condition ? {1'b1, ball_colours[b2_colour][8:11]}
296         : b3_condition ? {1'b1, ball_colours[b3_colour][8:11]}
297         : b4_condition ? {1'b1, ball_colours[b4_colour][8:11]}
298         : {1'b0, 4'h0};
299
300     assign VGA_ball_waveform[4:0] = wave_condition ? {1'b1, ball_colours[bar_colour][0:3]}
301         : {1'b0, 4'h0};
302     assign VGA_ball_waveform[9:5] = wave_condition ? {1'b1, ball_colours[bar_colour][4:7]}
303         : {1'b0, 4'h0};
304     assign VGA_ball_waveform[14:10] = wave_condition ? {1'b1, ball_colours[bar_colour][8:11]}
305         : {1'b0, 4'h0};
306 endmodule
```

Commented Code for Improved Layering in VGA_DISPLAY.v

```
17  module VGA_DISPLAY(
18      input CLK,
19      output [11:0] VGA_HORZ_COORD,
20      output [11:0] VGA_VERT_COORD,
21      output CLK_VGA,
22      input [1:0] mode,
23      input [14:0] VGA_mode_selector,
24
25      output reg [3:0] VGA_RED, // RGB outputs to VGA connector (4 bits per channel gives 4096 possible colors)
26      output reg [3:0] VGA_GREEN,
27      output reg [3:0] VGA_BLUE,
28      output reg VGA_VS, // horizontal & vertical sync outputs to VGA connector
29      output reg VGA_HS,
30
31      input [14:0] VGA_mode_one_text,
32      input [14:0] VGA_mode_one_waveform,
33      input [14:0] VGA_mode_one_tick,
34      input [14:0] VGA_mode_one_grid,
35      input [14:0] VGA_mode_one_back,
36
37      input [14:0] VGA_game_end_text,
38      input [14:0] VGA_game_text,
39      input [14:0] VGA_game_player,
40      input [14:0] VGA_game_cliff,
41      input [14:0] VGA_game_cloud,
42      input [14:0] VGA_game_back,
43
44      input [14:0] VGA_freq_text,
45      input [14:0] VGA_freq_wave,
46      input [14:0] VGA_freq_back,
47
48      input [14:0] VGA_ball_text,
49      input [14:0] VGA_ball_colour,
50      input [14:0] VGA_ball_waveform,
51      input [14:0] VGA_ball_back
52      );
53
54      // VGA Clock Generator (108MHz)
55      CLK_108M VGA_CLK_108M(
56          CLK,   // 100 MHz
57          CLK_VGA     // 108 MHz
58      );
59
60      // COMBINE ALL OUTPUTS ON EACH CHANNEL
61      // The displayed mode layer is always displayed, hence it is always on top.
62      wire [3:0] VGA_RED_CHAN = VGA_mode_selector[4] ? VGA_mode_selector[3:0]
63      : (mode == 0) ? // This checks the current mode that the user is in
64      (VGA_mode_one_text[4] ? VGA_mode_one_text[3:0] // First part of each line checks if layer is being used
65              : VGA_mode_one_waveform[4] ? VGA_mode_one_waveform[3:0] // If not, next layer is checked in the same way
66              : VGA_mode_one_tick[4] ? VGA_mode_one_tick[3:0] // If yes, the 4-bit colour is set.
67              : VGA_mode_one_grid[4] ? VGA_mode_one_grid[3:0]
68       : VGA_mode_one_back[3:0]) // If all layers are not being used, black background is displayed.
69      : (mode == 1) ? // Game layers are different from the other modes' layers
70              (VGA_game_end_text[4] ? VGA_game_end_text[3:0]
71              : VGA_game_text[4] ? VGA_game_text[3:0]
72              : VGA_game_player[4] ? VGA_game_player[3:0]
73              : VGA_game_cliff[4] ? VGA_game_cliff[3:0]
74              : VGA_game_cloud[4] ? VGA_game_cloud[3:0]
75              : VGA_game_back[3:0])
76          : (mode == 2) ?
77              (VGA_freq_text[4] ? VGA_freq_text[3:0]
78              : VGA_freq_wave[4] ? VGA_freq_wave[3:0]
79              : VGA_freq_back[3:0])
80          :
81              (VGA_ball_text[4] ? VGA_ball_text[3:0]
82              : VGA_ball_colour[4] ? VGA_ball_colour[3:0]
83              : VGA_ball_waveform[4] ? VGA_ball_waveform[3:0]
84              : VGA_ball_back[3:0]);
85      // Same process as above is done for the remaining channels as well
86
87
```

Commented Code for Game: Bird/Player Jump

```
193     // Player movement
194     wire player_condition = (VGA_HORZ_COORD >= player_horz_lower && VGA_HORZ_COORD < player_horz_upper) &&
195         (VGA_VERT_COORD >= player_vert_lower && VGA_VERT_COORD < player_vert_upper);
196     always @ (posedge CLK_VGA) begin
197         // Setting start positon of bird upon restart.
198         if (restart == 1) begin
199            player_horz_lower = 5;
200            player_horz_upper = 35;
201            player_vert_lower = 320;
202            player_vert_upper = 350;
203            game_over = 0;
204            game_won = 0;
205            HP = 1265;
206         end
207         else if (mode == 1 && game_running == 1 && (game_over == 0 && game_won == 0)) begin
208             // Performing the jump for a short period of about 0.5 seconds.
209             if (jumped == 1) begin
210                 if (jump_counter[23] == 1) begin
211                     jumped = 0;
212                     jump_counter = 0;
213                 end
214                 jump_counter = jump_counter + 1;
215             end
216             else if (cur_volume >= 256 && jumped == 0) begin
217             // 255 corresponds to 000011111111 pattern for the led register. This means that
218             // a jump is performed when the sound is above the volume level 8.
219                 jumped = 1;
220             end
```

References

1. Text Library: https://github.com/Derek-X-Wang/VGA-Text-Generator
2. Moving Average Filter: https://www.allaboutcircuits.com/technical-articles/implementing-a-low-pass-filter-on-fpga-with-verilog/