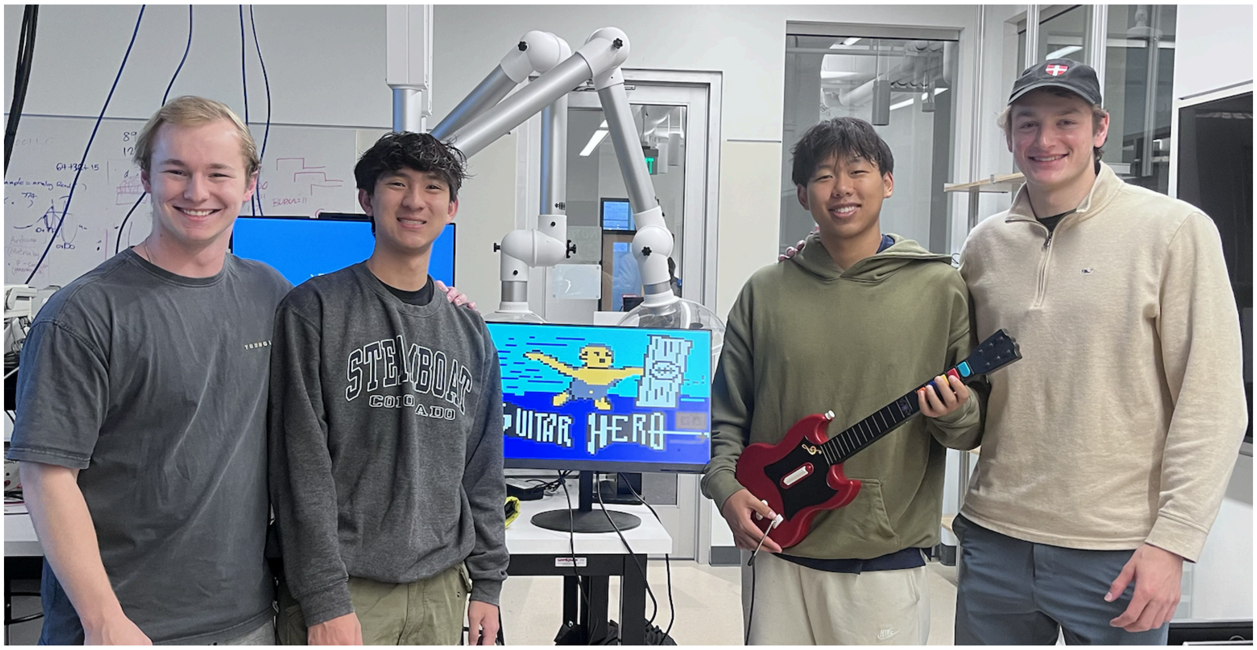
ES4 FA24 - Guitar Hero

Paul Wang, Michael Chou, Cooper Bailey, Maxwell Harrington



Overview:

Our main objective was to develop a “Guitar Hero” - esque game using an FPGA, VGA screen, and PlayStation guitar controller. Our goal was to have falling blocks on the screen as a form of sheet music for the player. If the player played a correct key combination when the falling blocks reached the bottom, they would be rewarded with a point. At the end of the game the total score was shown. Each key combination corresponds to a tone when played, so that if all key combinations on screen are played correctly, the notes of Kurt Cobain’s legendary 1991 “Smells Like Teen Spirit” guitar solo is blasted through an amplifier using a series of 1s and 0s.

To do this, we divided our project into three sections, the top module, the graphics, and the tone generation. In the top module, we created a state machine to switch between three different states based on inputs from the guitar or signal outputs from the VGA. This allowed us to show a start screen, then shift to an in-game screen based on a guitar input, then shift to the end screen when the game was finished. When a player was ready to restart, they could simply press one key, the strummer, on the guitar to return to the start screen. The top module also involved a variety of component mappings between the different modules.

The graphics proved to be one of the hardest components of the project, headed by Michael Chou. In the graphics, falling blocks had a variety of factors to choose from to meet our needs. In the end, 15+ different blocks were hard-coded into the FPGA to fall at specific intervals. These blocks ranged from eighth notes to half notes to align with the half-speed guitar solo, and to correspond to the buttons pressed on the guitar. The screen was divided into 5 sectors (one for each functional button on the guitar), and blocks were scheduled to fall down the screen in each sector. One major challenge was that two of the same block could not be displayed on the screen at the same time. To combat this, we reduced the tempo of the song, doubled the pixel-height of each block, and added extra blocks in worst-case scenarios to limit the number of blocks needed.

Finally, the guitar itself was simply connected to a few pull-up resistors so that whenever a key was pressed, the FPGA would read 1s for all keys unpressed and 0s for the keys which were pressed. These keys could be converted to audio tones using a square wave generator written into the code. Together these components effectively combined to create a guitar solo embodying Nirvana, while creating a challenge for the player to play it at even half the speed.

Component List / Table of Contents:

1) Top Module

State Machine + Component Mapping:

A large part of our program was integrating all the individual pieces together once they were working inside of a single top module. Before adding any module into the final program, we made a note of any signals which needed to be made and any inputs/outputs which were vital to the program's success. One thing which really helped was keeping the same names for signals used in the new program as the original program we were taking from. This ensured our port mapping and component declaration would include all the necessary information and not give us errors or bugs. When adding modules from another program, we copied all the necessary information from one person's code to the final code with the hope that we would only have to touch the original code once and then debug in the final module. This also ensured we didn't forget a vital part of the project. To figure out the necessary input and output signals inside the top module, we sat down and figured out how to run the interface of the screen, controller, and audio. Everything else which didn't directly affect those components were made into signals which would take inputs or outputs from the other modules. We also had to take a look at what modules were necessary to implement inside of top. Things such as backgrounds and numbers weren't used inside of the top module of the VGA display code, so we kept them separate and included them within the patterngen module. The actual top module is full of component declarations for other modules which depend upon the inputs from the user or signals which are created from a user's input.

When testing the final program, we only made one change at a time as this would ensure we know exactly what went wrong and could easily revert to a working program. This practice was very beneficial, as we were able to weigh the change against the previous version of the program and decide if the change was necessary if it caused other parts of the program to look worse. One example of this is when implementing the game screen, the bars looked weird after adding another note to play with the hopes a user could play a note which wasn't part of the actual song but would add an extra twist to the game. This change caused the bars to look significantly worse when falling down the screen leading us to immediately take this change out and resynthesize the game back to the original.

The state machine was designed to be straightforward. We realized that the main dependence on the state of the game is purely the graphics. As such, to prevent unwarranted problems, we designed the state machine in the patterngen.vhd file. The state machine determined the rgb output onto the screen. The implementation of the state machine is visible below:

```
case curr_state is
            when start_screen =>
                  game_start <= '0';
                  if (go_to_game = '0') then
                        curr_state <= game_state;
                  else
                        curr_state <= start_screen;
                  end if;
            when game_state =>
                  curr_state <= game_state;
                  game_start <= '1';
                  if (go_to_end = '1') then
                        curr_state <= end_screen_state;
                  end if;
            when end_screen_state =>
                  game_start <= '0';
                  if (go_to_start = '0') then
                        curr_state <= start_screen;
                  else
                        curr_state <= end_screen_state;
                  end if;
      end case;
end if;
```

The state machine comprises three states, and there is only one condition in each state that moves the state of the game. This was done to avoid unnecessary bugs. The condition go_to_game was decided to be all possible buttons on the guitar pressed at once. The condition go_to_start is just the strummer. We designed different buttons for different changes of states to avoid skipping states. The condition go_to_end is outputted by the game.vhd module. This allows us to go to the end screen after the game has finished playing. The variable *curr_state* helps determine which rgb value that we send out. As seen below, we can create a mux where the state is the input and the rgb value is the output. State_screen holds the value of the rgb and gets outputted out of pattern gen.

```
state_screen <= start_screen_rgb when curr_state = start_screen else
game_rgb when curr_state = game_state else
end_screen_rgb;
```

2) Graphics

Falling Blocks and VGA:

It's important to understand how the VGA screen works. The VGA screen populates the screen pixel by pixel, by using x and y values to populate the screen horizontally, then vertically. Essentially, VGA divides the screen into a grid of pixels (640x480) and scans them sequentially, row by row, at a high frequency to create a smooth, continuous image.

The main component for the actual game was the falling blocks, which indicates to the player when and which keys to press on the guitar controller. The hardest part about this was to have the timing of the blocks exactly right, so that it actually plays the correct rhythm of the song. This was done with the fact that the VGA screen populates all of the pixels at 60Hz. By implementing a counter inside of "`if x = 640 and y = 480 then`", we can be sure that the counter is updating once every frame. The if statement basically means whenever all the pixels are populated on the screen, update the counter. We slowed the tempo of our song to 60 clock ticks per quarter note to make things simple for us, and easier to play for the user (we realized nobody could play the actual tempo of the song). This made it easy to implement a counter that counts up 30 times for every eighth note. Each bar would "turn on" and fall at its respective time to the counter. For example, if we wanted a note to fall after the first beat of the song, it would turn on and start falling at counter = 60 (one beat = two eighth notes for a song in 4/4). When the counter reached the end of the song, we reset the counter and made a signal turn on to go to the end screen.

Now that timing is out of the way, we created different types of bars for each type of note. In the guitar solo of "Smells like Teen Spirit," there are eighth note beats, quarter note beats, and dotted quarter note beats (equal to 3 eighth notes). Because there are 5 bars falling for the 5 inputs on the guitar, we initially created 15 different bars. This worked great until we realized that some of the bars weren't falling. The reason for this was that a bar would get sent down, but while it was in the process of falling through the screen, the counter would try to turn that same bar on to fall again, which wouldn't happen since the bar is still falling (the bar is still on, thus cannot be turned on again). This forced us to create a whole extra 8 bars that serve as "backups" to deal with this problem. These backup bars would act as copies of the original bar, but would be available to fall while the original bar was falling.

The logic of the bars falling was simple. Each bar had a specific x and y signal, for tracking its position on the screen. The x position was fixed for where we wanted the bars to fall, and every time the bar was turned "on", the bar's y position was updated by how many pixels we wanted it to fall until it reached the bottom. Once the first part of the bar touched the bottom of the screen, the bar would send out a signal until the bar finished falling that would get compared to the guitar controller's input for the scoring.

The rgb values of the bar is what actually allows us to visualize the bar on the screen. The way it works is that an rgb signal (a specific color) gets sent out for the pixels that the bar is located in, using the width of the bar, and the length of the bar, so that color only shows up for when the bar is on the screen. For example:

```
rgb <= "000011" when (y > bar_position1y-BAR_HEIGHT or
(bar_position1y-BAR_HEIGHT>bar_position1y)) and (y < bar_position1y) and (x >=
bar_position1x and (x < bar_position1x + BAR_WIDTH)) else
```

What this means in simple terms is, as long as the current y pixel position is between the y position bounds of the bar (with two conditions based on if the bar is still falling), and between the x pixel position bounds, then send out the specified color. Originally we had the screen filled with black as the alternative for if there is no color being sent out for that specific pixel (if there is no bar). But then we decided to implement a background screen. This was easy, as we already implemented a background screen before with ROM which we will explain next. All we had to do was replace the “when others” statement with the background game screen logic.

Start, End, Game Screens, Numbers:

In order to make the game look more like the real guitar hero, we decided to implement different screens for start, the actual gameplay, an end screen, and numbers to display a user's score at the end of their turn. In order to do this, we used pixel art editors on a canvas size 60 x 80. This was done purposefully as this size screen could be expanded up to fit the actual display size and keep the number of bits required to “paint” the screen down.

The bulk of the work to display the background and numbers was done by a python script. This script took in a .png file and turned it into a .txt file which contained the definitions of what rgb value should be used for a given 13 bit address. The script divided the png colors by 85 so we can get a 6 bit rgb value, which is what the VGA needs. The script looked at all the pixels in the original file, takes the the rgb value from the current pixel, creates an address from the first 6 bits of the y-coordinate and first 7 bits of the x-coordinate and adds the following statement: '   when "' + addr + '" => rgb <= "' + color + '";\n' into the txt file. In order to compress the file for space concerns, we ran the script once without any filters and briefly looked for the color we guessed took up the most space, and then added an if statement such as: if (color != "111111") to get only the lines where other colors occurred, for this example the most frequent color would be white. For the final when others case in the case statement, we made the rgb value equal to the most frequent color. This data compression saved lots of lines, as instead of having ten different numbers and three backgrounds all containing 4,800 lines of a case statement, they were able to be closer to 3,000 for the backgrounds and 200-400 for the numbers.

Another process of optimization for our code was by downsizing the png images. The screen is 480 bits by 640 bits, if we hard code every single pixel with our ram code, we would run out of look-up-tables. To avoid this critical issue, we sized down all the screens to save space. Our start and end screens are 60 by 80 bits. This means that at most, the maximum number of lines of code would be only 4800. This also means that the x coordinates were only 7 bits and the y coordinates were only 6 bits. To size up, we only observed the 7 most significant digits for the x coordinates and the 6 most significant digits for the y coordinates. This can be seen below:

```
end_screen1 : end_screen port map (clk => clk, y_in => y(8 downto 3), x_in =>
x(9 downto 3), rgb => end_screen_rgb);
```
This effectively created blocks of colors which saved significant logic space. We implemented this with both the start and end screens.

Counter:

The counter system was a tricky bit to implement. The main issue for the counter was very similar to the button problem in the seven seg dual display lab. Our counter has a very simple function; during the game, if you press the right button/buttons at the right time, you will be granted a point. The score is the accumulation of points over the course of the song. Fortunately, the process of finding when the player plays the right note at the right time is fairly easy. The game.vhd file sends out an expected key variable that is 5 bits. This is then compared with the input keys of 5 bits through an equal sign.
```
correctNote <= '1' when expectedkey_in_keyCompare = not
inputkey_in_keyCompare else '0';
```
However, the very important part to consider is that our clock is 60Hz. This means that if you check this every clock signal, then a single eighth note, which is half a second, would result in 30 plus points, because the clock is thinking you are playing the right note 30 times. As a result, we also need to account for when someone plays the right note, and it hasn't been played right already. We came up with this solution below.
```
process (clk) begin
      if rising_edge(clk) then
      key_previous <= key_current;
            if (reset_count = '1') then
                  counter <= 6d"0";
            end if;
            if(expectedkey_in_keyCompare /= "00000") then
                  if (key_current = key_previous) then
                        if (is_new_key = '1') and correctNote = '1' then
                              counter <= counter + 1;
                              is_new_key <= '0';
                        end if;
```

```
                    else
                          is_new_key <= '1';
                    end if;
              else
              end if;
        end if;
end process;
```

First we created a d-flip-flop to compare the previous note and the current note. Then we check if the current keys aren't all unpressed, because it wouldn't make sense to award a point if you don't do anything. If keys aren't all unpressed, then we check if the current and previous keys are the same. If they are, it tells us that the same note is still being played. This lets us fix the crux of the problem. The code below is how we award points.

```
“if (is_new_key = '1') and correctNote = '1' then
                              counter <= counter + 1;
                              is_new_key <= '0';
                        end if;”
```

We first check if the note is new, or hasn't been played right yet. Concurrently, we check if we are playing the correct note. If both are true, then we add to the counter, and say the key is no longer new. This tells us that we have awarded the point for this specific note. Now zooming out, if the previous and current key aren't the same, then we know the key has changed, or a new key has appeared. As a result, this allows us to reset the is_new_key value back to 1. This counter is then outputted through a decimal converter to get a value of a ones and tens digit.

The display of the counter was quite monotonous. To display all digits, we had to individually draw out all digits from 0-9. To not make our numbers seem wonky, we had to align the digits so that they were all level with each other. The size of the image for the digits were 20 bits wide and 40 bits tall. This is too small to see; as a fix, we sized the bits up by 2 so that it would be easier to see.

```
one1 : one port map (clk => clk, y_in => y(6 downto 1), x_in => x(5 downto 1),
rgb => one_rgb);
```

The process to display the score was to prioritize it over the other rgb values. We did this manually by finding the x and y coordinates for where we wanted the ones and tens digits.

```
rgb1 <=
    tens_digit_rgb when (y > "101111100" and y < "111100000") and ( x >
"1000000011" and x < "1000101011") and curr_state = end_screen_state else
      ones_digit_rgb when (y > "101111100" and y < "111100000") and ( x >
"1000111010" and x < "1001100010") and curr_state = end_screen_state else
      game_rgb when curr_state = game_state else
      state_screen;
```

As seen above, prioritized the ten_digit rgb when we specifically were in the end_screen and specific x and y range on the screen. Moreover, the actual tens_digit_rgb was determined by the output of the counter through the dual decimal converter.

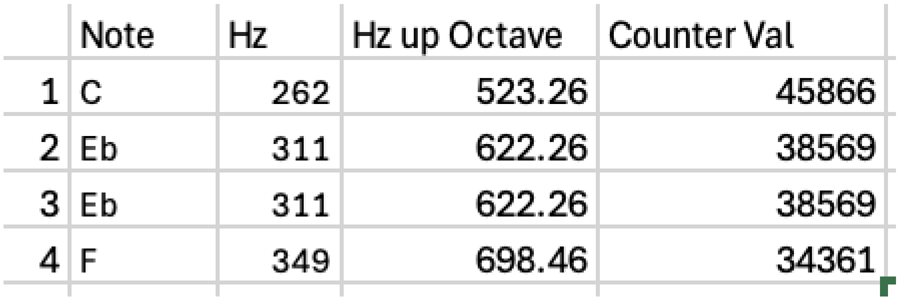
3) Audio

Guitar Hero Controller:

The Guitar Hero controller proved simpler than expected. Each of the six buttons on the controller corresponded to one of the wires on the output cable, and there was one more wire for ground. We connected a circuit so that when no buttons were pressed, the input to the FPGA would be HIGH (5V -> resistor -> FPGA). This was possible, because by default the circuit through the guitar button would have very high resistance. However, when a button was pressed on the guitar, it would short over the resistor, encouraging the FPGA input to go LOW (5V -> resistor -> guitar -> GND), thus the FPGA got a LOW signal on the buttons which were pressed. This worked specifically with 330 Ohm resistors and a 5V signal, any other changes resulted in unclear signals to the FGPA. We also noticed that the GREEN key did not work after a full disassembly of the guitar, so we excluded it from the game.

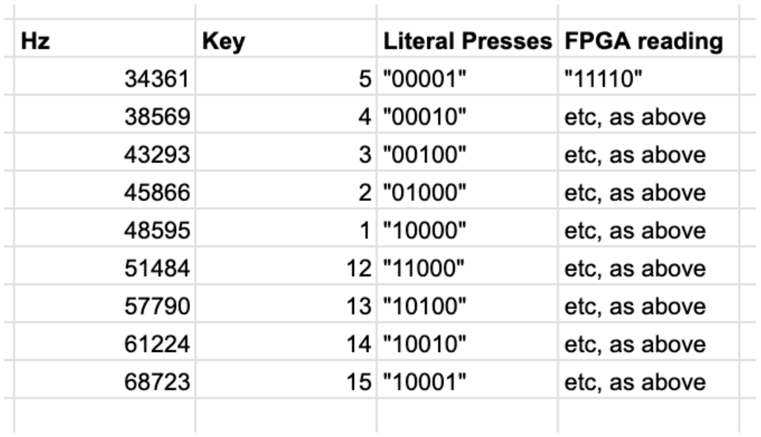
Tone Generation:

A key part of the project was pairing up the melody of the guitar solo into its notes (ie. A, C, C, Eb, etc.) then into its Hz values (440 Hz, 261 Hz, 261 Hz, etc.) and then into tone limit counter values (54545, 27272, 37862, etc.). Given a 25.125 MHz clock, we used an excel sheet to convert every note's Hz value into the required tone limit counter value. This tone limit counter value would allow us to set the output data HIGH when the counter was below the tone limit, and LOW when above the tone limit value and below two times the tone limit. This created a square wave. For example, for a 262 Hz tone (a "C") we shifted it up an octave to make the tone appear louder, (523 Hz) then divided 25.125 MHz by 523 to get the required number of counts before reaching the tone limit. Thus, higher tones (F, 698Hz) would have lower counter values (34361) as the square wave would have to happen more times each second than in the case of a lower tone (C, 523 Hz, 45866 counter value).



	Note	Hz	Hz up Octave	Counter Val
1	C	262	523.26	45866
2	Eb	311	622.26	38569
3	Eb	311	622.26	38569
4	F	349	698.46	34361

There were a total of 9 different notes in the solo, and thus 9 different keys were needed. As there were only 5 on the guitar, we added an option for 2 keys simultaneously pressed to result in a tone. Key Encodings and the associated code to generate tones is included below.



Hz	Key	Literal Presses	FPGA reading
34361	5	"00001"	"11110"
38569	4	"00010"	etc, as above
43293	3	"00100"	etc, as above
45866	2	"01000"	etc, as above
48595	1	"10000"	etc, as above
51484	12	"11000"	etc, as above
57790	13	"10100"	etc, as above
61224	14	"10010"	etc, as above
68723	15	"10001"	etc, as above

```
limit <= 18d"35972" when inputkey_in_Audio = "11110" else
        18d"40377" when inputkey_in_Audio = "11101" else
        18d"45323" when inputkey_in_Audio = "11011" else
        18d"48016" when inputkey_in_Audio = "10111" else
        18d"50873" when inputkey_in_Audio = "01111" else
        18d"53898" when inputkey_in_Audio = "00111" else
        18d"60498" when inputkey_in_Audio = "01011" else
        18d"64094" when inputkey_in_Audio = "01101" else
        18d"71944" when inputkey_in_Audio = "01110" else
        18d"1";

process (VGAclock_in_Audio)
begin
        if rising_edge(VGAclock_in_Audio) then
                count <= count + 1;
                        if inputkey_in_Audio = "00000" then
                                tone <= '0';
                        else
                                if count < limit then
                                        tone <= '1';
                                elsif (count < limit&"0") and (count > limit - 1) then
                                        tone <= '0';
                                else
                                        count <= (others => '0');
                                end if;
                        end if;
        end if;
end process;
end;
```

Results:

Our game worked especially well with minor bugs. For example, a few strange lines appeared in the VGA screen at certain points, which we are still uncertain of their origins. Also, the game was so incredibly hard, that most people could not score more than 15-20 points out of a possible 71, with only 10% scoring more than ~50. This was with the solo played at half speed as well.

Gameplay, Audio Recording of Solo, and Loud version with Backtrack are found here:
https://drive.google.com/drive/folders/1Wtv4NriQDH2I1d97AGIThwLV2oHUskPO?usp=sharing

Reflection:

Overall the project was a success. The team worked great together and the end product worked especially well. In general, we did a great job dividing up roles to conquer specific components of the project. Certain things can only be truly coded by a single person at once given the nature of Radiant, but it was especially helpful to have two people with eyes on the code at the same time. This allowed errors to be caught, and, especially when in a 100-200+ line excel sheet of numbers, keys, and counter values, helpful to stay on track. A few minor adjustments we would make for next time would be to create a better timeline of the work at the beginning of the project. A headstart on the project in Lab 7 allowed us to stay on track with the audio component, and our early failure on the SD Card allowed us to cut our losses early and focus on more important components instead. We didn't need to stay up too late on the final night of the project, and had an MVP by 11:30pm. However, knowing projected tasks and estimated timelines in the start would have been helpful for coordination purposes.