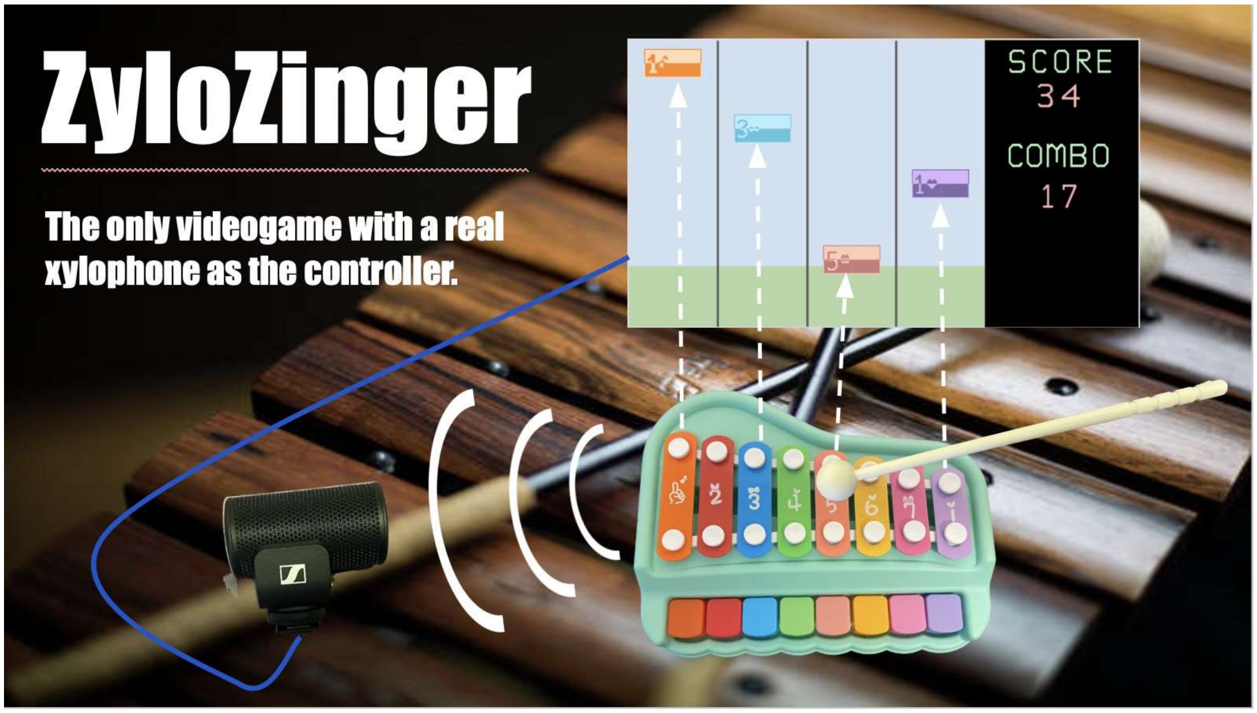
Final Report - ZyloZinger

Sienna Brent - scb2197

Alex Yu - asy2126

Riona Westphal - raw2183

Rajat Tyagi - rt2872



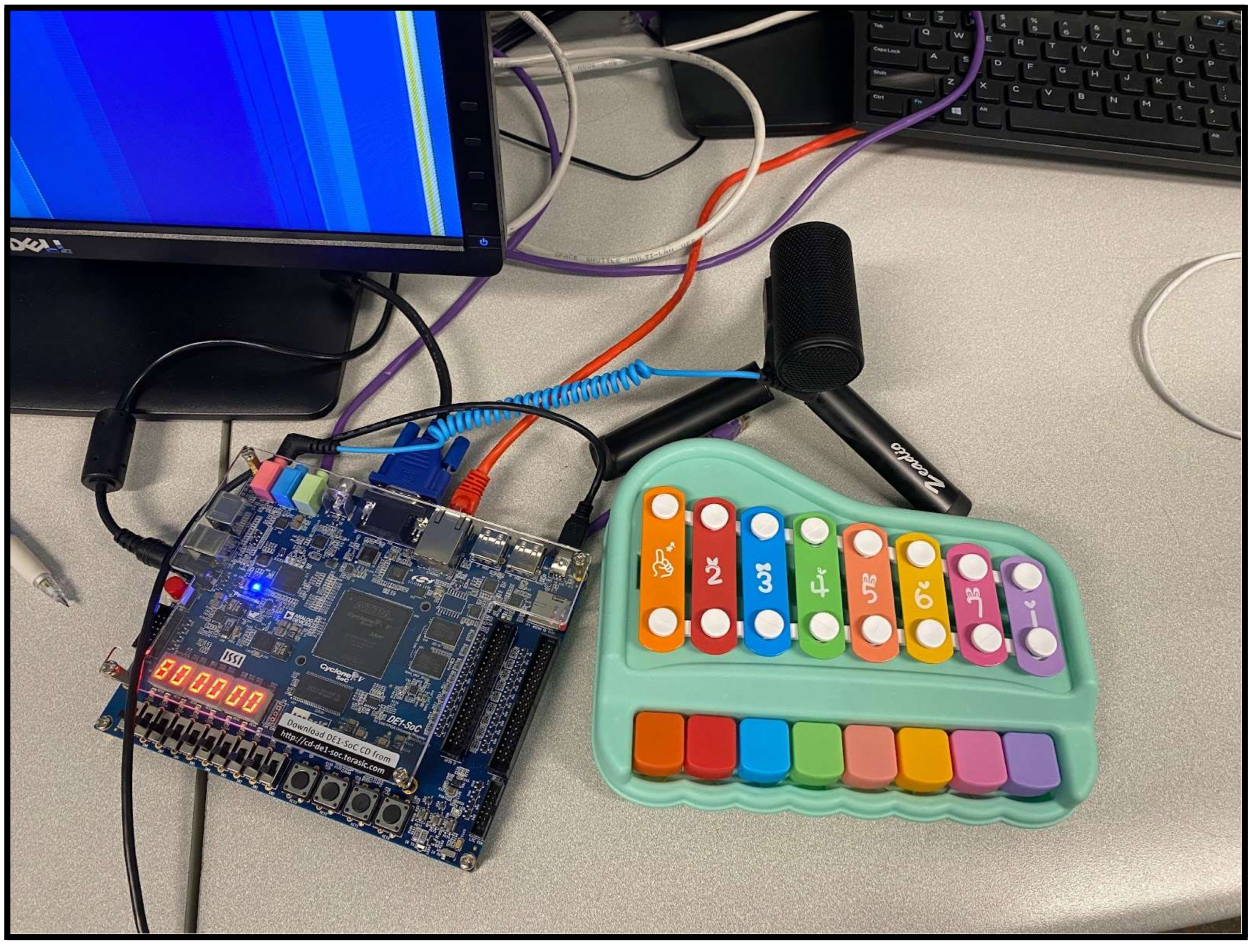
TOC

1 Project Overview

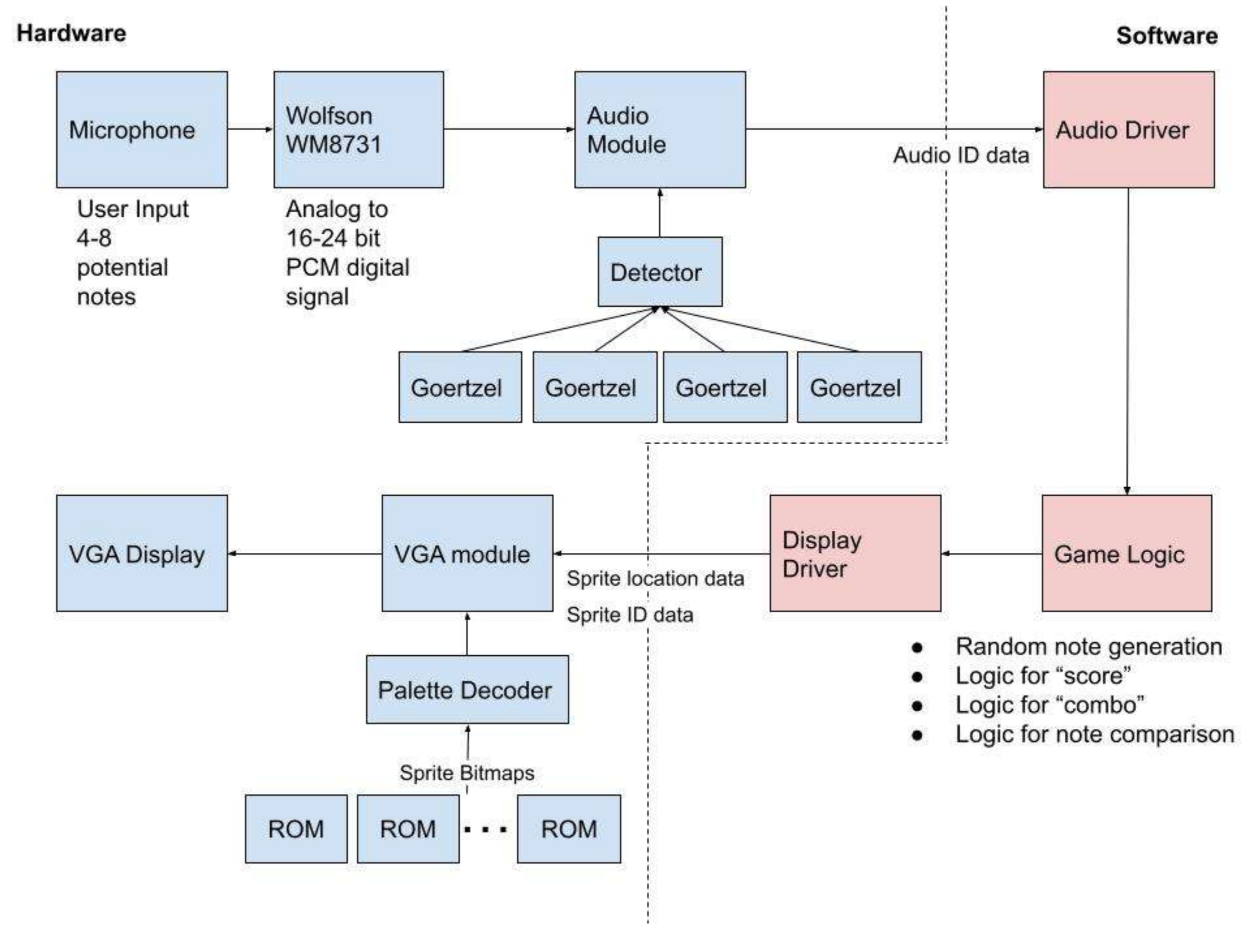
Zylo Zinger is a device that uses a microphone connected to the FPGA through the mic-in port to recognize four distinct tones being played on an external instrument. Attached to it is a videogame that utilizes sprite display to perform a guitar hero style rhythm game. The incoming audio will be used in a video game simulation on the VGA display to earn “points” for the user if the correct notes are played at the correct time. Thus it is similar in spirit to guitar hero, trombone champ, or even DDR. However, unlike those games listed, the game does not provide the music to play a beat to, but rather visually shows a beat and music prompts for the player to play. Simply put, if the player is bad, the sound is bad, if the player is skilled, the sound is good. The project has significant memory savings by utilizing sprite graphics and uses line buffers instead of full frame buffers when drawing the image.

The video display will show falling objects that represent the note the game wishes for the player to play on their instrument. In our case, it is a baby xylophone piano combo toy. The toy covers a range of notes from high g7 to g8, thus the product was inaccurately named and should be called a baby glockenspiel piano in consideration of its significantly higher pitch.

The game is played on the toy and the microphone picks up audio generated by the toy. The role of the FPGA is to be both a signal processing device, process the game logic in hardware, and to support vga sprite graphics for the game display.



2 System Design and Components



The basis of the project is a pitch recognition machine with a visual feedback display based on the outputs of the pitch recognition. At a high level, this translates to a hardware -> software -> hardware design, in which audio data is gathered in hardware, sent on an Avalon bus to the software in the "readdata" signal, which is then used to update the graphics data. Finally, graphics data is sent back on another Avalon bus on the "writedata" signal. The two hardware components were controlled using separate device driver modules.

3 Hardware

3.1 Audio

3.1.1 Design goals

- Output 4 bits of data
- Always available result
- Can differentiate between a played note and silence

- Sample audio at 48khz

Audio detection was done in two parts. First is to receive analog data from the microphone into the audio Codec chip and return a 24 bit PCM digital data stream. The second part was to use this data stream and identify if the target frequencies created by the physical children's toy piano xylophone were there. For reference, 48khz is the Nyquist frequency of the human ear. For those unfamiliar with signal processing, the Nyquist frequency is the minimum frequency required to discreetly sample data points of an analog signal to replicate it perfectly without losing data at half the playback frequency.

3.1.2 Audio CODEC Interface

The audio CODEC is typically configured using the I2C bus. Because our team was on a faster pace deadline, we opted to use a prepackaged audio CODEC interface driver [1] from Altera which was customized by Professor Scott Hauck at Washington University and used previously by the 2019 TNShazam project. The outputs of the top-level file from this library were adc_right_out and adc_left_out, which we turned into a signal stream with audioInMono = (adc_right_out>>1) + (adc_left_out>>1). The readout from hardware was stored in a set of registers called buffer. Buffer would change to BRAM output from the tone detector output based on how it is configured, an option sent by software. By default, it reads from the tone detector.

3.1.3 Audio Collection and Verification

To collect audio, we used a Sennheiser MKE 200 directional microphone plugged into a 3.5mm cord into the audio in port on the FPGA board. Much of the early stages of the project was to ensure that the data received by the microphone and output by the audio codec chip was what we expected. To check for this, a BRAM module was created to store 65536 24bit words of alleged 24 bit pulse code modulated audio data and a userspace program would read from this BRAM and write its contents to a .txt file which the team could then read and extract from the board for later analysis and playback on non-FPGA devices. The final audio.sv file still has the BRAM instantiated and a write to and read from procedure still exists in the device driver aud.c and aud.h files.

Their function is to have a userspace program to send a desired amount of samples for the hardware to store and then once done storing, allow for software to read from BRAM in order to have properly time stamped samples. The BRAM module used was the two port BRAM shown in Professor Edward's memory lecture slides.

The primary reason for using this approach instead of directly sending the output of the Wolfson-WM8731-audio-CODEC was because there was no way for software to tell if the data read skipped any samples or reread the same sample or not. By utilizing BRAM in this way, the team ensured that each audio sample was read only once and each and every one was stored in BRAM for the number of samples desired. The use case was not just a simple FIFO, software could request data from any specified address but in this case it basically was a FIFO.

Once a txt file of a single second worth of audio data sampled at 48khz was created, it could be analyzed with external software like MATLAB and Audacity where we could perform things like spectrum peak analysis and come up with methods for filtering..

3.1.4 Goertzel Algorithm

Primarily, the ZyloZinger relies on the Goertzel single tone-detection algorithm [2]. This algorithm takes in 1024 samples at a sampling rate of 48kHz and produces a single-bin DFT output. A 2-tap IIR feedback calculation is performed for every incoming signal, and after 1024 signals have been processed a single feed-*forward*, FIR-like calculation is performed. The output of this complex feed-forward calculation is the amplitude of a single given spectral component.

In order for these calculations to be feasibly performed by an FPGA at the speed we needed, we made two major adjustments: First, we hardcoded the coefficient that was required for the feedback calculations. Second, we modified the Goertzel algorithm to output the *power* present within a single given spectral component, instead of the amplitude. To achieve this, the entire equation must be squared and as a result, the complex component of the feed-forward calculation is removed.

The Goertzel algorithm was implemented in our program by instantiating four “Goertzel” modules in the top-level audio module, each one differentiated by a different coefficient parameter which corresponded to a different spectral component’s bin number. The incoming signal stream from the CODEC feeds into all four instantiations, and four power outputs are being updated every 1024 incoming signals.

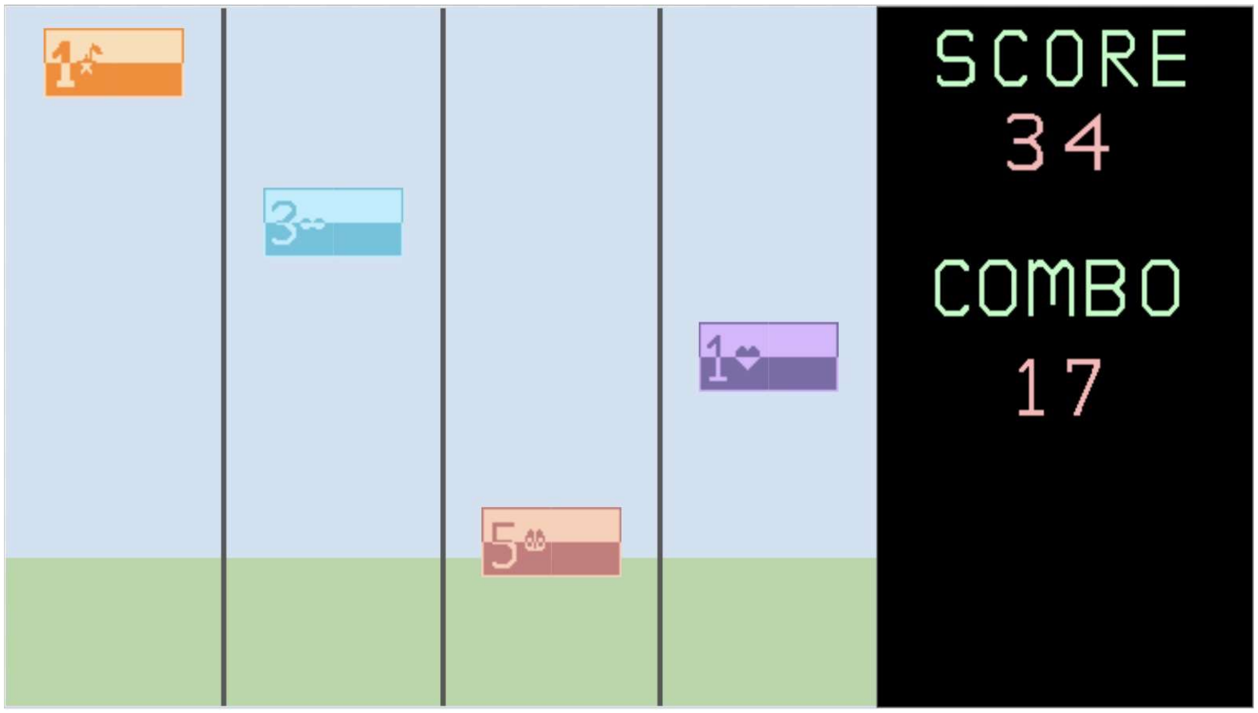
3.1.5 Detector Algorithm

The detector algorithm works by taking in the four power outputs from the four Goertzel modules, and comparing them in order to output an “overall result” that would ultimately be sent to hardware. This comparison works by instantiating several “ratio” variables, one for each combination of powers. For example, the ratio_12 variable represents the power detected at Goertzel_1 divided by the power detected at Goertzel_2. Additionally, a set of latches is instantiated which function as a FIFO queue.

When each set of 4 new results come into the detector from the Goertzels, the counter increments by one and the ratio variables are checked to determine which Goertzel, of any, had the power with the largest magnitude. If none of the ratio variables are sufficiently high, our detector reads this as silence. This new result, the number of the Goertzel with the largest power magnitude (or silence), gets input to our “queue”, and the oldest result of the queue is discarded.

When the counter reaches a preset value (defined as `INTERVAL) equal to the number of latches in the queue, a simple summing calculation is done on the members of the queue to determine which Goertzel's power had the largest magnitude the most often. This "overall result" is sent back to the top-level audio module.

3.2 Video

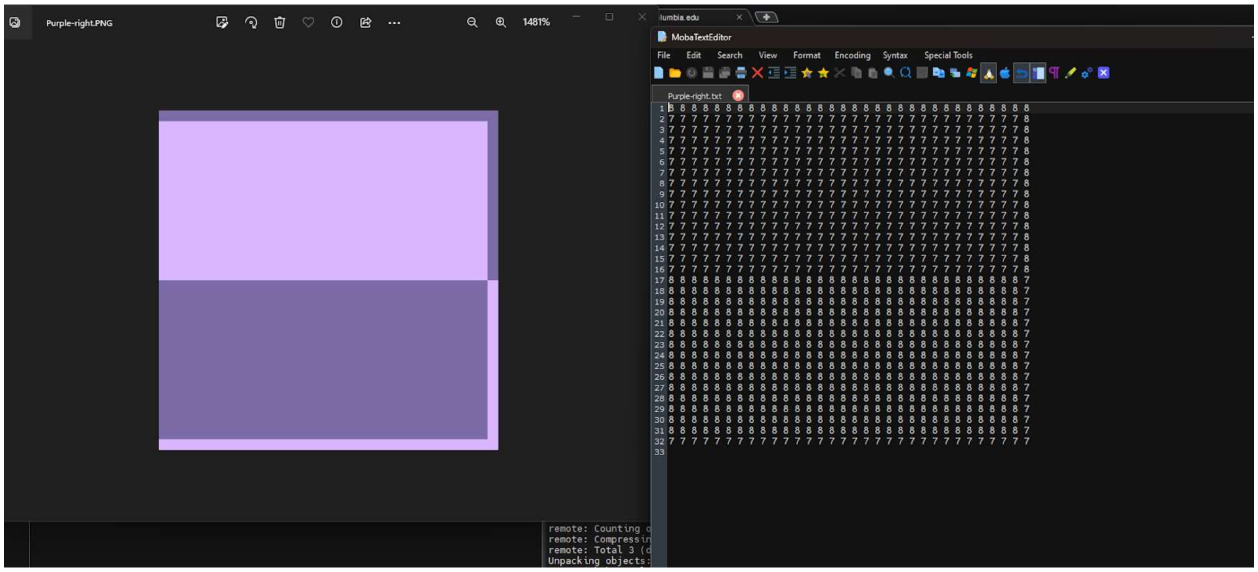


3.2.1 Sprite Generation

To achieve the desired display, the "background was first drawn statically using the positions of hcount and vcount from the vga_counters module. The features drawn with this technique include the green "valid" region, the shades of blue in the background, the black rectangle of the menu, and the vertical lines down the display. For everything else, including the noteblocks, letters, and numbers, sprite graphics were used.

Using three smaller BRAM modules to store 64 x and y coordinates, and sprite names, it is essentially the sprite attribute table where each sprite is the address to the BRAMs. Thus to call on a sprite, the system checks for the sprite index or address and from these BRAM modules find their appropriate information. The x and y coordinates where both 10x64 bits and the ids were 4x64 bits. This means our system could support the storage of up to 64 individual active sprites on the screen.

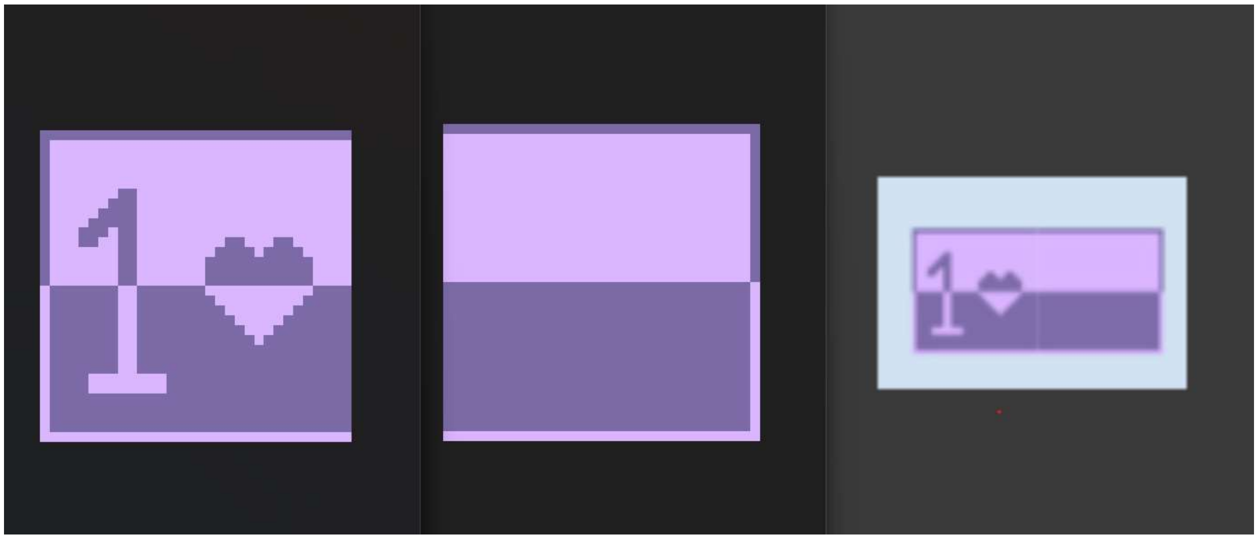
The system does not generate sprites in layers but rather checks if a sprite is in the raster line sequentially by their index from least to greatest. This means that if a higher indexed sprite will overwrite a lower indexed sprite should they share the same coordinates. It should be noted that while this feature exists, our final design and game logic never has a case of overlapping sprites.



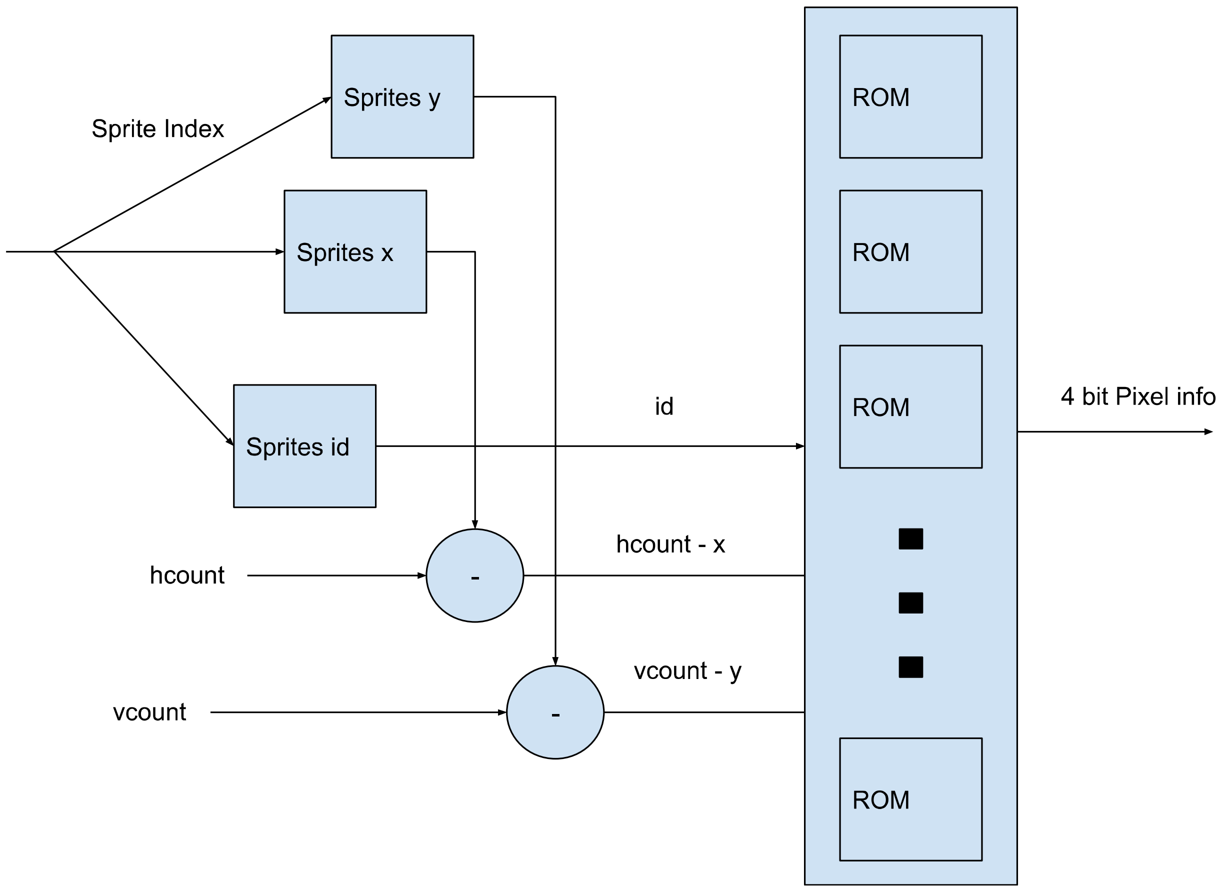
```
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 7
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
```

```
remote: Counting o
remote: Compressin
remote: Total 3 (d
Unpacking objects:
```

We created custom sprites by drawing the pixel art for them in a 32x32 pixel space. They were stored as .ng files and we used a python script to parse through the files to generate a .txt file containing the color information of each pixel in an array. These .txt files would be read with the $readmemh function during “make quartus” to create ROM modules preloaded with these pixel maps. This technique was taken from [3]. Larger sprites, such as the noteblocks, were represented with multiple 32x32 sprites adjacent to each other.



3.2.2 Line Buffers



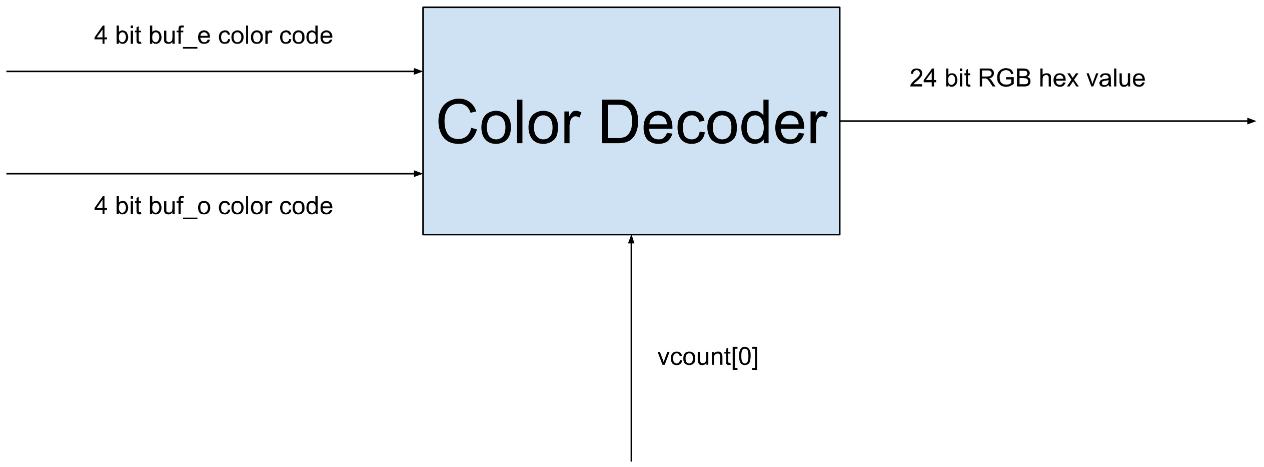
To have the four note blocks fall from the top of the screen to the bottom smoothly without jitter, sprite mapping needed to be on a high resolution frame buffer. However, a full frame buffer for 4 bit pixels would take up 1228800 bits each. While it was probably possible to fit it fully for 2 or 3 frame buffers, a simpler solution of using alternating line buffers was found, also at [3], and used.

The two line buffers each were tri-state state machines: during each raster line, line buffer was being written to while the other prints its contents to the screen by being read to VGA_R, VGA_G, and VGA_B. The line buffers were 640*4 register arrays so they could efficiently flush and draw the background in a single clock cycle instead of using multiple sprite layers. Thus technically, the overall system was a 4-state state machine.

There are 1600 clock cycles per line drawn. So in those 1600 cycles, the state machine looks up into the 3 BRAM modules containing sprite x and y coordinates and id based on the index which went into the address of these memory modules. This information is used to check if the sprite was on the line and if it was, over the next 32 clock cycles, to draw its matching line pixel by pixel in the current line buffer: the one not being printed to the screen. The sprite's pixelmap information is held in ROM memory modules, one for each sprite. These are looked up using the sprite ID and read from in order to write to the line buffer. It then cycles to the next sprite index to repeat the process until the next line is reached. The limitation to this implementation is that if there were more than 45 sprites present in any given line, then there would not be enough clock cycles to finish drawing them all in the line drawing phase of the buffer. Luckily, our use case did not call for that many sprites to be present in the same line.

Functionally, the 3 xy and id BRAM modules act as the sprite attribute table and the index is the sprite drawn.

3.2.3 Color decoding



By storing color in each sprite ROM and line buffer as only 4 bits instead of the full 24, memory usage was effectively reduced by nearly a factor of 6. The decoder is a simple mux. It takes in the 4 bit color code as a selector and outputs a color RGB value based on that. The color decoder in vga_zylo takes in two 4 bit color inputs from both the even line line buffer and odd line line buffer and choses which one to read based on the selector bit vcount[0] indicating if the vga screen is currently drawing a even or odd numbered line.

4 Software

4.1 User Space Game

Like the well known IP Guitar Hero, our game is played by striking the correct keys at the correct times. When to strike is visually related to the player with falling “notes” as we call them. The notes travel down the screen at a fixed speed allowing the player to anticipate and know when to strike the corresponding note once it reaches the green section of the screen. Strike when the note is in the green because green is for good and blue is for bad. Each correct note will result in a point, potentially boosted by the current combo count- the number of correctly hit notes striked in a row. Three values, “score”, “combo”, and overall maximum combo abbreviated to “max” are kept track of.

To go about achieving this, a number of things needed to be done.

- Refresh screen function for new game
- Create an array of sprite structs containing x, y, dx, dy, id, hit status

- Score, combo, and max combo counter
- Check of sprites in the valid region
- Update sprite x and y based on dx and dy

These functions were relatively simple to implement. The way software kept track of sprites and sprite positions was to create sprite structs that contained a set of ints representing their x and y positions, dy- a constant 1 or 0, dx- 0 unless struck to fly off screen, id- to tell hardware if the sprite is a letter, number, or note component, and hit status- which is relevant to check if this is a note to be checked for a match with the audio input or not.

At the start of a new game, when the program is called to run, all sprite objects are refreshed to a default state. The first 22 sprites are reserved for SCORE, 3 score digits, COMBO, 3 combo digits, MAX, and 3 max combo digits. The rest of the available sprite indexes are to represent note blocks. During the refresh all these 22 sprites are set to predefined positions and never moved again. The 9 sprites representing the digits however can swap sprite ids to show different numbers on the screen.

In every game cycle thereafter, all notes are programmed to increase their y-coordinate by one (visually represented as the noteblocks falling down the screen). All of the letters do not update their positions or IDs at all. But while the update for the noteblock and letter sprites is relatively simple, additional logic is required in order to update the number sprites for SCORE, MAX, and COMBO appropriately.

Every game cycle, a function runs that searches through the array of sprite structs for the y-coordinates of all sprites with an ID matching that of one of the four noteblocks. If any of these y-coordinates are within a certain range (visualized on the screen as a green strip), the software begins reading the data coming in on the readdata attribute of the Avalon bus (it should be noted that this data is always being delivered to software regardless of game state, but that the software only chooses to read it at specific conditions as determined by game logic). If the value that's read matches the value of the sprite ID with the y-coordinate in the valid region, a "hit" counter is incremented by one. If this hit counter reaches eight (though this is a number we picked based on qualitative testing of the game's sensitivity), the note is counted as being hit.

When a note is hit, several things happen: first, SCORE is incremented by keeping the same position but switching the sprite IDs at that position. Normally, this is a simple increment by one, but depending on the magnitude of COMBO it may be more. For every 5 points of your combo score, the score delta is increased by an extra point. For example, if you have a combo of 17, you may expect your score to be increased by 1 + 3 = 4 points total.

COMBO can be updated on one of two conditions. Either a note is hit, or the y-coordinate of a note becomes equal to the y-coordinate of the first non-valid pixel (i.e. the first blue row on the display). A "combo flag" variable, either 0 or 1, helps us keep track of whether the last noteblock was also hit or not. If the current note has been hit according to the above logic and the combo flag is raised, combo is incremented by one and the flag remains raised.

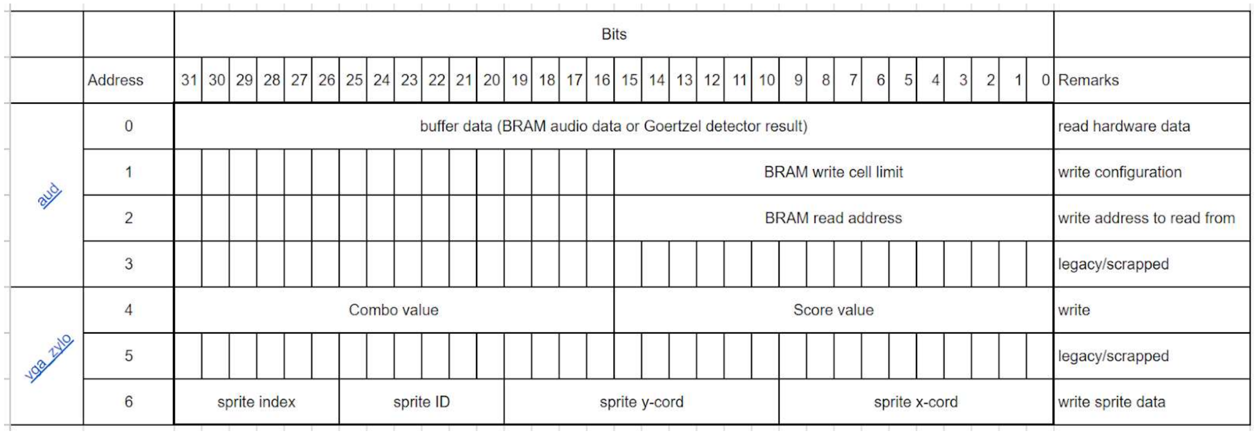
However if the note was not hit, i.e. it left the valid region without a positive detection, the combo flag goes to zero and COMBO is updated to be zero as well.

The logic for MAX is extremely simple: each game cycle, MAX compares its value to COMBO, and if COMBO > MAX then MAX will update its value to be equal to that of COMBO. Else, MAX retains its old value.

When all the sprites have been updated according to this logic, the data for each sprite struct in the array is packaged into a simple 32-bit array to be sent to the hardware. The first 6 bits of each 32-bit element of the packaged array represent the index, the next 6 represent the ID, the next 10 are for the y-coordinate and the final 10 are for the x-coordinate.

5 Software Hardware Interfacing

5.1 Register Map



		Bits																															
	Address	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Remarks
aud	0	buffer data (BRAM audio data or Goertzel detector result)																																read hardware data
	1																	BRAM write cell limit																write configuration
	2																	BRAM read address																write address to read from
	3																																	legacy/scrapped
vga_zylo	4	Combo value																Score value																write
	5																																	legacy/scrapped
	6	sprite index						sprite ID						sprite y-cord										sprite x-cord										write sprite data

5.2 Driver files

Hardware and software communicated with each other using the Avalon bus. Our device drivers were functionally the same as they were when used in vga_ball lab except a single ioread function used by aud.c.

5.2.1 aud

The aud.c driver both read and wrote to the board. In the early stages of the project, there was a need to store and extract audio data from the microphone without losing samples. The solution is mentioned above but to put it simply, audio data was written into a specified number of BRAM cells and read back by software at its own pace. Thus because the readback is controlled by software and the write is controlled by hardware, there is no risk of losing data because the two cannot be synchronized.

Even though the software is sending ints, only the least significant 16 bits were read by hardware as the BRAM created only held 0xFFFF addresses. It could have held more but this much was enough for the purpose of frequency spectrum peak analysis to formulate the parameters needed by the Goertzel modules.

Three IO functions were made, read_memory() to read from BRAM at its current read address, write_limit() to tell the board that it wished to store audio data into BRAM a specific number of cells, and write_address() to set the BRAM write address. The read_memory() function does not necessarily read from memory but rather from the readdata port of the Avalon bus; the name is from back when the module and driver only read from BRAM.

5.2.2 vga_zylo

The vga_zylo.c driver handled sending sprite data. However, to simplify the role of hardware, score and combo sprite positions were handled in software rendering addresses 4 and 5 unused. It was originally anticipated that it was necessary to keep an address open for each sprite we wished to render but we later realized that by keeping an array in a struct and looping the call to write sprite data for each sprite was more code efficient. In vga_zylo.c the function is called write_packet() because it writes a packet of sprite data to hardwares writedata port.

6 Closing Thoughts

6.1 Potential future work

There was some disappointment in that we did not implement a complete video game with a user interface beyond the gameplay itself. It also is not very much extra work to allow for the game to read from a predetermined list of notes and rests but the idea of needing to transpose actual music into our game note by note seemed like a ton of work that no one had the time or patience to do with the limited amount of time we had to work.

It would also not have been difficult to implement four more Goertzel algorithms such that each key of the piano would be able to have been detected. Initially, we were planning on shrinking the size of the FFT from which the Goertzel modules were subsampling, from 1024-point to 512-point, to allow for a greater margin of error in the tone detection. With a 512-point FFT, the tones of all 8 keys on the piano became too close together to separate into distinct spectral components, and for this reason we stuck to four. However, because ultimately our algorithm did not need the extra room for error, we stuck to 1024-point and only have four keys still as a result of lack of time.

7 Sources

[1] Huack, Scott, and Kyle Gagner. “Audio Tutorial - Class.ece.uw.edu.” UW Electrical & Computer Engineering, University of Washington, 18 May 2015, https://class.ece.uw.edu/271/hauck2/de1/audio/Audio_Tutorial.pdf.

[2] Staff, Embedded. “The Goertzel Algorithm.” *Embedded.Com*, 9 Nov. 2022, www.embedded.com/the-goertzel-algorithm/.

[3] Green, Will. “Hardware Sprites.” Project F, 26 Mar. 2023, https://projectf.io/posts/hardware-sprites/.

8 Appendix: Code Listing

All code and peripheral files can be found on https://github.com/lerntuspel/ZyloZinger.git