CSEE 4840 - Embedded Systems
Design Document: Piano Heroes

Anita Bui-Martinez (adb2221), Michael John Flynn (mf3657),
Zakiy Tywon Manigo (ztm2106), Robel Wondwossen (rw3043)

Spring 2025

1 Introduction

Our project's goal is to create a video game inspired by *Piano Tiles - Don't Tap the White Tile* created by Hu Wen Zeng in 2014. We will expand the game to include up to two octaves of a piano in which the player will have to play the correct corresponding notes on the USB piano keyboard. Users must play the correct keys in the right order before they scroll off the screen. When each key is pressed, a piano note will be played, and the pressed note will be signified on the display. Otherwise, the player will lose the game.

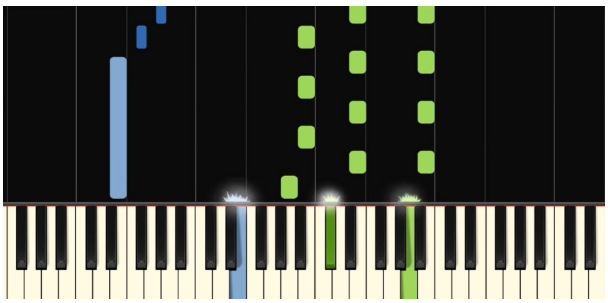

Figure 1: Display Inspiration

2 Block Diagram

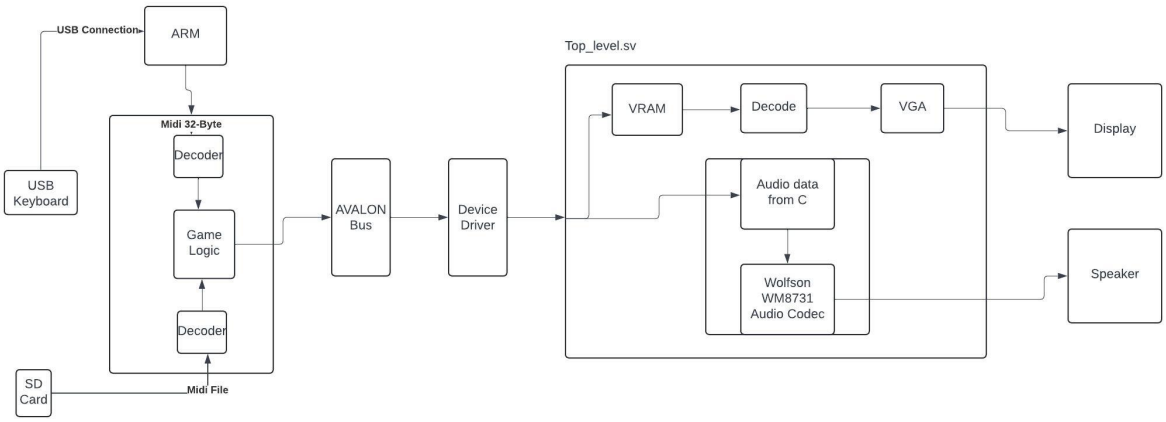

Figure 2: Project Block Diagram

3 Algorithm

1. **Game Logic**

   - Our project will allow users to choose a song to attempt to play, along with the choice of different levels of difficulty. Once a song is chosen, the game will begin.
   - The correct notes for the chosen song will scroll down from the top of the screen. As the notes get to the bottom of the screen, the user must play the corresponding notes in the correct order on the USB keyboard. When notes are played by the user, the game will also play the audio of the corresponding note. If the user plays an incorrect note or doesn't play the correct note in time, they will lose the game.
   - While the game displays the falling notes on the VGA display, it will also wait for input from the USB keyboard.
   - The vertical speed of each tile, v, is computed from the song's tempo (BPM) and the frame rate f:

     $$v = \frac{\text{pixels\_per\_beat}}{f \times \left(\frac{60}{\text{BPM}}\right)}$$

     In this context, `pixels_per_beat` denotes the number of vertical pixels a tile must move for each musical beat (quarter note) so that the scrolling animation stays in sync with the song. This will depend on our play field height and how many beats of "lead time" we want before notes reach the hit line. For example, our play field is 480px tall and we would like tiles to take 4 beats to traverse it, we would set

     $$\text{pixels\_per\_beat} = \frac{\text{playfield\_height}}{\text{beats\_of\_lead\_time}} = \frac{480 \text{ px}}{4 \text{ beats}} = 120 \text{ px/beat}.$$

With this value, the per-frame descent

$$v = \frac{\text{pixels\_per\_beat}}{f \times (60/\text{BPM})}$$

ensures each tile moves exactly 120 px every beat (i.e. every 60/BPM seconds).

2. **Data Flow - Reading Input from Keyboard**

- The FPGA receives this data through two addresses:
  - 0x00001000 for MIDI data
  - 0x00001004 for the timestamp
- To facilitate communication between the Linux software and the FPGA, physical memory is mapped into the user-space program using /dev/mem. The FPGA exposes registers beginning at base address 0xFF200000, with a total mapping span of 2 MB (0x00200000).
- Relevant addresses:
  - MIDI Data Register: 0xFF201000
  - Timestamp Register: 0xFF201004
- Each MIDI message + timestamp can be written in packed 32-bit words or individually:
  - FPGA base addr — 0xFF200000
  - Register span — 0x00200000 (2 MB)
  - MIDI data offset — 0x00001000
  - Timestamp offset — 0x00001004

Byte	Value	Meaning
0	0A	USB header (CIN = 0x0A) – "Note On" message, not part of core MIDI
1	90	Status byte: 0x90 = Note On, Channel 0
2	3C	Note number: 0x3C = 60 = Middle C (C4)
3	7F	Velocity: 0x7F = 127 = maximum key press strength

Table 1: Example Output Explanation

3. **Displaying the Keyboard**

- We can use a 2D array of 3-bit values where the entries are the hcount and the 6th vcount bit. The 3 bits to encode are the corresponding colors: WHITE, BLACK, LIGHT BLUE, DARK BLUE, LIGHT GREEN, DARK GREEN, LIGHT RED, and DARK RED.
- We also only need to store one octave and repeat the display of that octave.

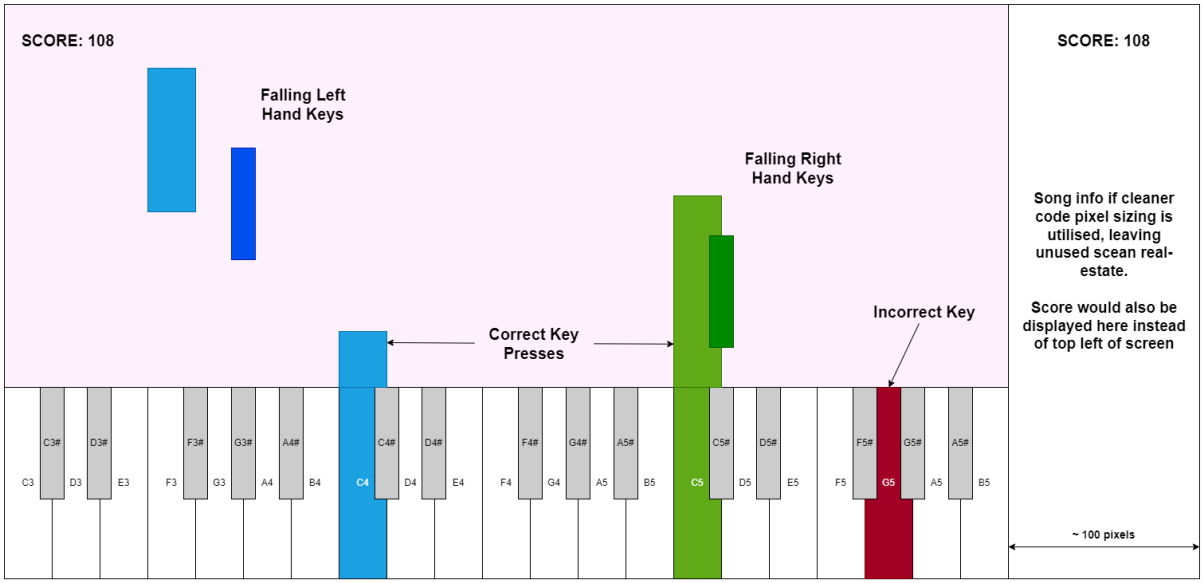

Figure 3: Display Map

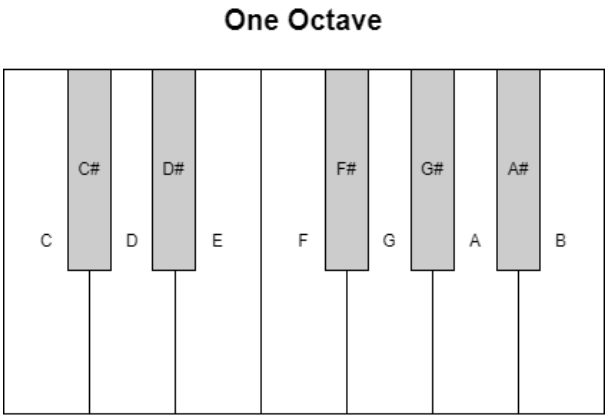

Figure 4: One Octave with Corresponding Notes

4. **Displaying the Falling Tiles**

- **Tile Data Structure**
  - We would maintain an array or circular buffer of active tiles.
  - Each tile stores:
    - `column` ($0 \ldots N-1$, where N is the number of piano keys)
    - `spawn_time` (timestamp when tile appears at $y = 0$)
    - `speed` (pixels/frame, derived from BPM and frame rate)
    - `color` (3-bit code matching the key's color)
- **Position Calculation**
  On each VGA frame, we would read the current time t_{now}. For each active tile we would have to compute

$$y \;=\; \big(t_{\text{now}} - \text{spawn\_time}\big) \times \text{speed}.$$

If y exceeds the play field height (480 px), mark the tile "missed" and recycle its entry.

- **Hit-Detection Overlay**
  We will draw a "hit line" at fixed `vcount` $= v_{\text{hit}}$ by forcing that scanline to a special indicator color whenever `vcount` $== v_{\text{hit}}$.

5. **Playing the Sounds**

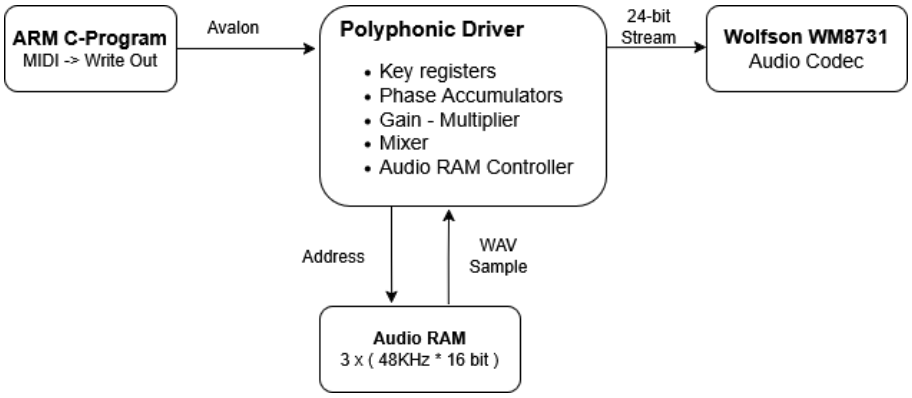

Figure 5: Audio Production Flow

The ARM C program captures MIDI events, and collects them into a temporal set of keystrokes before writing the data to the Avalon bridge. On the FPGA side, the polyphonic driver updates one of eight (supporting 8 simultaneous key presses, subject to change) phase accumulator key registers containing information on the pitch, gain, and note-on/off status. This information is utilised to modify one of 3 WAV samples stored in RAM, this is possible as modification of various C notes (C3, C4, C5, C6, C7, etc...) will faithfully recreate the other notes within an octave. Our reasoning for this design choice is to maximise audio quality with 48KHz 16 bit audio samples, while saving valuable space within the RAM. As every key stroke invokes the register to re-read these same three samples and pitch-shifts them over their associated octaves, total RAM usage for audio is less than 300 KB, far less than storing a separate waveform per key. On each edge event of the Wolfson WM8731 codec's 48 kHz advance signal, the interface cycles through the active keys, fetches the appropriate sample, applies gain, accumulates the results in a mixer, and finally outputs 24-bit left/right signals to the codec.

4 Resource Budget

1. **SONGS:**

   - Midi file that gets transferred from the C code.

2. **VIDEO:**

   - Each key is 8 or 16 bits across - 16x24 keys
   - Using the 2D array of 3-bit values, each 3 bits corresponding to the different colors we will use and given that the keyboard will be 128 pixels tall, it will take 3.4KB to store all the states of the keyboard.

- There will be 1 pixel wide vertical lines between each whole note to differentiate them by making every 24th pixel of the background black.
- The text on the right side of the screen indicating the score and the song information would be part of the frame buffer.

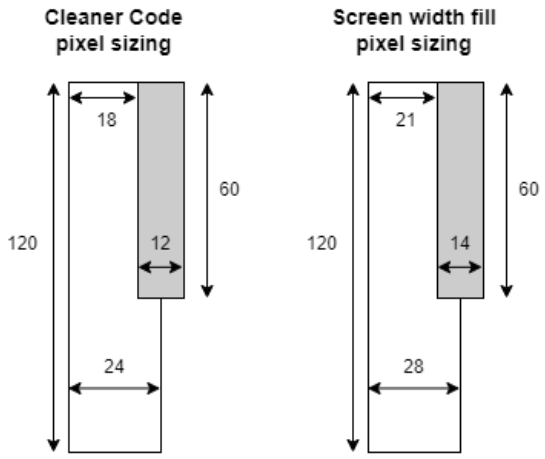

Figure 6: Pixel Sizing

3. **AUDIO:**
   - 3 x 1 second 16-bit samples of C notes (C3, C4, C5), at 48KHz.
   - Approximately 280 KB of RAM, jsut over half of the available on chip FPGA RAM.

5 Hardware-Software Components

1. Hardware
   (a) **DE1 SoC FPGA Board**
   The FPGA implements the core game engine logic in hardware. It receives parsed note data and associated timestamps and uses them to spawn visual elements, perform hit detection, and compute player scoring in real time.
   (b) **VGA Output Display**
   The VGA monitor connected to the board displays the gameplay by showing falling notes and reactive feedback when the player presses keys at the correct times.
   (c) **USB Keyboard - Novation Launchkey Mini [MK3]**
   This keyboard is connected to the DE1-SoC FPGA development board, which runs an embedded Linux environment. USB MIDI messages from the keyboard are captured using the libusb library
2. Software
   (a) **C** - The language we will use to create the game logic is C
   The software running on the Linux side includes a lightweight C-based MIDI logger program. This program uses several libraries:

i. libusb-1.0: Used for USB communication with the Launchkey Mini, enabling direct access to raw MIDI packets over a bulk endpoint.

ii. sys/time.h: Provides access to microsecond-accurate timestamps, allowing the system to measure note timing precisely.

iii. fcntl.h and sys/mman.h: Used to perform memory mapping from user-space to the FPGA's memory-mapped registers.

iv. unistd.h: Used for system-level utilities such as introducing brief delays to avoid overwhelming the FPGA with data bursts.

These libraries together allow for real-time parsing, logging, and communication of MIDI input from the USB device to the hardware logic.