Gridthym: 6.205 Final Project Report

1st Giuliana Cabrera
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
gpcs@mit.edu

2nd Carlos Sanchez
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
cjsanche@mit.edu

3rd Rafael Chavez
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
rbchavez@mit.edu

***Abstract*—We present the design of an FPGA-based interactive rhythm and piano-learning game that integrates real-time multimedia processing and hardware-accelerated gameplay. The system utilizes a modular hardware architecture to manage song storage via an SD card, keyboard input for note and pitch detection, audio playback in WAV and MIDI formats, and dynamic game visualization. Players interact with falling tiles corresponding to musical notes, creating an immersive experience.**

Our implementation leverages the FPGA's parallel processing capabilities to handle custom song parsing, keyboard input processing, and audio synthesis while driving a high-resolution visualization module. The SD card interface supports seamless loading of user-defined songs, and the audio pipeline ensures high-quality playback with low latency. The game visualization replicates a piano-learning interface, enabling precise synchronization of gameplay elements with audio output.

We evaluate the system's performance in terms of responsiveness and accuracy, discuss its scalability for additional features such as advanced visual effects, and outline future improvements for enhanced educational and interactive experiences.

***Index Terms*—FPGA, MIDI, WAV, HDMI, SD**

I. Hardware and Design Overview

The hardware in our project consists of:

- **2 GB SD Card:** Formatted with the FAT file system, the SD card serves as the primary storage for MIDI and WAV files, providing the audio tracks and timing data necessary for game-play.
- **Keyboard Interface:** An external keyboard is used to input notes and pitches. This is facilitated by a custom `keyboard_controller.sv` module which processes an 11-bit signal. The module implements checks for start and stop bits, along with parity for error handling. The output, an 8-bit ps/2 hexcode, is then processed later on for game controls.

At a high level, the system operates by reading audio data from the SD card, processing it for synchronization, and rendering it visually as falling tiles on an HDMI display. Players interact with the system by pressing the corresponding keys on the keyboard, which are processed in real-time to evaluate gameplay performance. The DAC ensures precise audio reproduction, aligning with both visual and input components for a cohesive experience.

The block diagram provides an overview of the interactions between these components as shown in figure 1, as shown below.

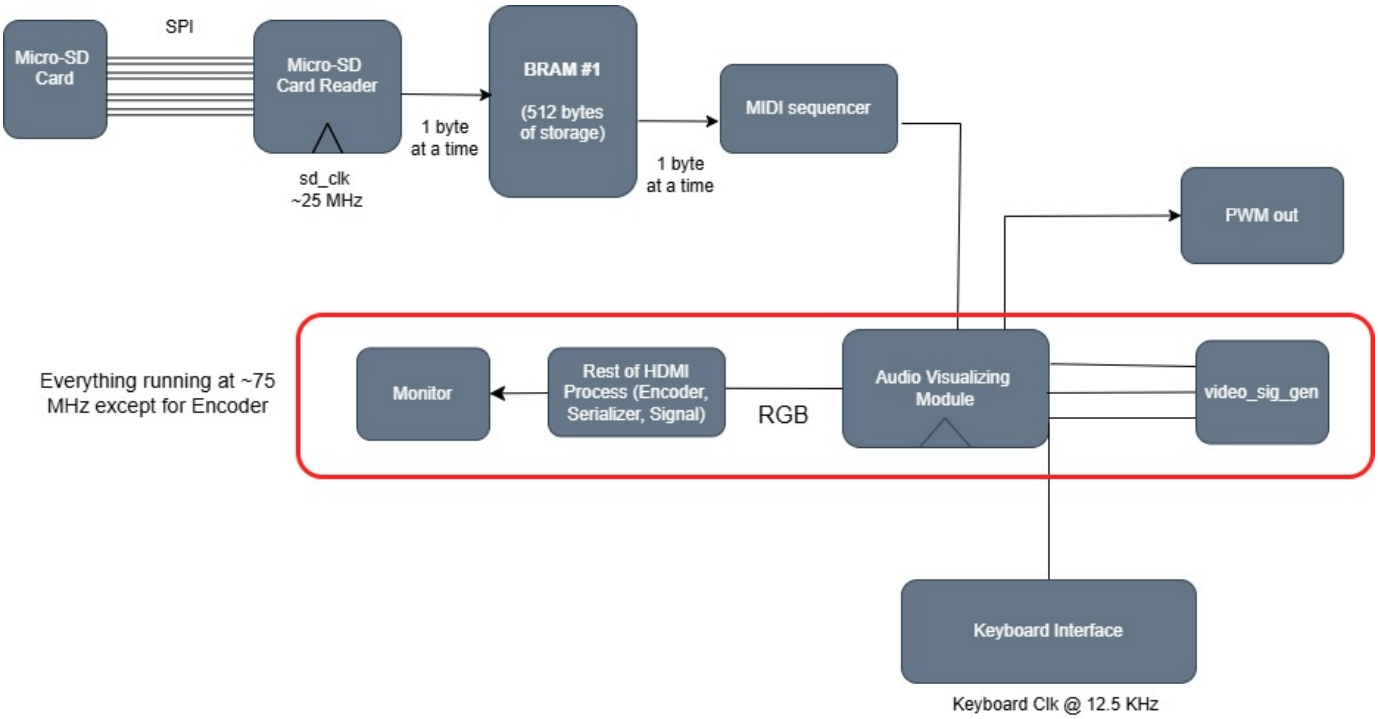

Fig. 1. Fig 1. Overview diagram of the Gridthym system, illustrating the integration of SD card storage, keyboard input, audio processing, and real-tie visualization.

II. User Interface

A. Display [Carlos]

In selecting the game, the user will be able to use one of the buttons already built into the FPGA to circle through several MIDI files. We are using the seven-segment display in the Urbana Board to show which file we are reading from; File 1 is the Pokemon RGB midi file, File 2 is Creep by Radiohead, File 3 is Billie Jean by Michael Jackson, and File 4 is a hex-code used for testing. Pressing button 3 once will change the song to play. The press (debounced) is used as a song trigger and changes the number that appears on the display to signify which song you are playing as described above.

B. SD-Card [Carlos]

The sd controller module we use is from a previous project, all credit goes to Jonathan Matthews' group [2]. The sd controller module operates at 25 MHz and essentially facilitates communication between the FPGA and the card through SPI. The module handles initializing the SD card, creating 'ready' statuses between the SD card and peripherals further down in the block diagram, and reading/writing to the SD card. Our project utilizes only the read feature. The read operation is performed in 512 byte increments beginning at an initial address.

The controller currently feeds into a BRAM with a width of 8 bits and depth of 512 entries (just enough for a single 'read' operation).

However, due to the varying sizes of the MIDI files on the SD card, we implemented another module to increment the initial address based on the sector length of each individual file. This `file-selector.sv` module acts as a mapping that changes the accessed sector within the SD card to read from and write to the BRAM each trigger (trigger being either a song change or an external pulse provided from any of the audio processing modules). The initial trigger is automatic and given by the song selection which would be button 3.

C. Keyboard [Carlos]

The second prominent piece of hardware was the ps/2 keyboard. The keyboard requires 4 connections: a Vcc of 5V, GND, Data, and Clk. Our setup holds the keyboard in a device-host relationship with the FPGA. The FPGA can only provide 3.3V for Vcc, which proved to be sufficient for operation. Additional probing revealed that the keyboard's clock, an input into the FPGA, is 12.5 kHz, vastly slower than the 25 MHz clock everything is running on. Knowing this, we develop our module to focus on determining the falling edge of the Data signal from the keyboard and slowly fill an 11-bit register with the bits from Data. Our ps2keyboard reader follows an FSM similar to UART, but with an additional state to check the odd parity as shown in Figure 2 (on the right). Once in the STOP/TRANSMIT state, a combinational circuit switches bits around and outputs a valid 8 bit long, ps/2 hex code that would be read by the game mechanics 'section' of our project.

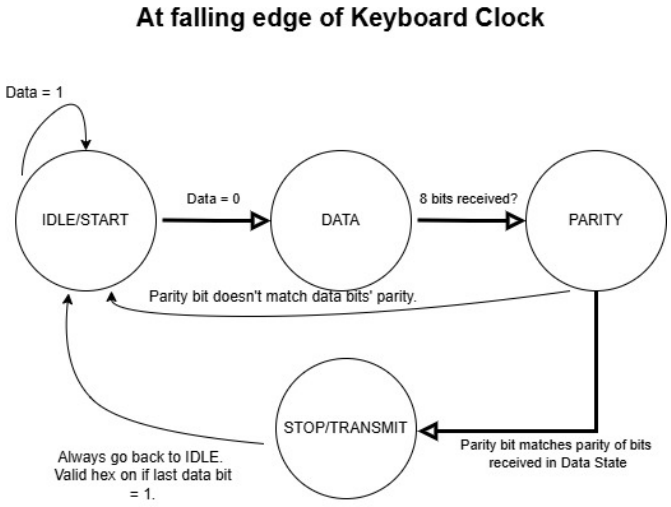

Fig. 2. $PS2keyboard - reader$ FSM

D. Challenges and Future Changes

- **Multiple Key Press Detection:** Our keyboard interface only reads one key at a time due to the limitations of a PS/2 protocol. Future projects should have multiple outputs to account for simultaneous keypresses.
- **Synchronous SD Card Reading:** All of our modules run at a 25MHz clock. Since most notes in the MIDI files last longer than 0.5 seconds, we can leverage our speed to read from the SD card and output its bytes, all while still being on time. However, there may be files that require synchronous reading such as WAV files with large amounts of data.

III. Audio Processing

Once the file is selected from the SD card, the audio processor will synthesize and analyze the audio data based on the corresponding file format. To add complexity, the only preprocessing done on the files is to turn them into Type 0 MIDI files, which are suitable to process sequentially. To avoid having a large synchronization buffer from our display game, we pass the note values from our processor to the display game, which will output audio when the rendered tiles reach the portion of the screen that takes the keyboard input.

A. MIDI File Sequencing

MIDI files are structured to convey musical note information in a compact, event-based format. Each file consists of messages specifying pitch, velocities, and durations, along with optional, yet frequent meta events that need to be ignored. For this project, MIDI files are parsed to extract the note events and convert them into waveforms, while reading and ignoring meta events. More specifically, each MIDI file is divided into:

- **Header and Track Chunks:** Byte chunks that are usually at the beginning of the file and contain crucial timing information for our playback. By extracting our ticks per quarter note and the microseconds for each quarter note,

we can calculate the number of clock cycles for each MIDI tick, and even change them while keeping the same song structure. For our case, we set the ticks per quarter note to 96 and

- **Ticks Per Quarter Note:** Extracted from the MIDI header, which defaults to 480 or 960 ticks per quarter note if unspecified. To allow for the song to be easily tested with existing midi sequencers and playable for our user, we lowered this value to 96, resulting in about 5200 microseconds, or 130,000 clock cycles per tick.
- **Delta Time Parsing:** Variable-length byte sequences are decoded to determine how many ticks to wait before executing the next MIDI event. The lower 7 bits of each byte are concatenated until the MSB is high.
- **NoteEvent[0]:** Contains a note on or off value in the upper nibble, along with the lower byte containing the track channel of the MIDI sound. Notably, this byte contains a 1 on its MSB.
- **NoteEvent[1]:** Contains the pitch of the note in the lower 7 bits, and a 0 in the MSB.
- **NoteEvent[2]:** Contains the velocity of the note in the lower 7 bits, and a 0 in the MSB.
- **Meta Events** : These variable length events give you information about the song's structure, but don't tell you what notes to play. While these Events are helpful for rhythm analysis, they don't add helpful information that a beginner could take advantage of, so we don't output these events

Because we can't access the entire MIDI file from the FPGA, we utilize BRAM that is updated by the SD card to parse the MIDI file. We stream the BRAM at each clock cycle unless we are at the waiting stage of our sequencer, in which case we wait until we reach the stage where we need to decode the MIDI event.

Once 512 bytes of the MIDI file are processed, we wait for the BRAM to be updated in the top level before accepting a new byte. Since a tick alone takes about 130,000 clock cycles, waiting for a BRAM refresh does not cause issues with approximating the melody and visual of a song. We use the note value's signal, and feed it to the display game in real time to create a visual of the song being played.

B. Audio Output using PWM

When the player presses the correct key according to the display, the corresponding column should enable audio to be played at the frequency of the corresponding note. To do this, we map each column to a MIDI value ranging from 60-71, and a sine wave at the corresponding frequency. To do this, we compute $\text{clock cycles per sample} = \frac{\text{clock frequency}}{(\text{note frequency} * \text{samples per cycle})}$ Lastly, we step through an 8 bit sine table with respect to this number of clock ticks. We use the amplitude as the duty cycle for our PWM, which produces our desired sound. Both approximations made use of Python files that generated the look-ups to avoid time-consuming operations.

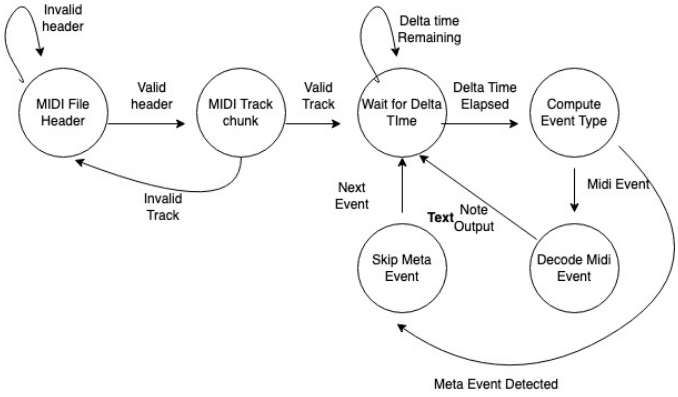

Fig. 3. MIDI Sequencer FSM

The extracted note's frequency values are precomputed on the FPGA, allowing for efficient access during gameplay directly from Data Byte 0. These mappings were also inexpensive on the memory, only needing 255 bytes for the sine table and 512 bytes for the MIDI to frequency mapping

To create sounds from MIDI files, we use a waveform generator to create different types of waveforms at the specified frequencies to allow for variations in sound. Waveform types include sine, sawtooth, and square waves.

C. Challenges and Future Work

- **Harmonic Pitch Detection:** Because we were limited by hardware that made it difficult to detect multiple notes on the keyboard and display, we only considered taking the most recent note as our note value. Future sequencers should output every note that is being played to allow for higher difficulty songs.
- **Instrument Synthesis from MIDI files:** Currently, the audio output from MIDI doesn't resemble the sound of an instrument. It would be ideal to gather pre-processed sounds from instruments to make the game more intuitive.
- **Integration Testing:** Although complete MIDI files have been analyzed, there still needs to be testing to ensure that the audio portion will continue to work with the display data buffer and output notes as close to the actual tempo of the song as possible.
- **I2S audio:** To improve the audio quality of the audio output, we can use an I2S protocol to a DAC and a speaker with AUX.

IV. Visualization of the Game - Giuliana

The visualization aspect of the Gridthym project focuses on rendering the interactive rhythm game on a 1280x720 resolution HDMI display.

A. Background and Design Inspiration

The visualization approach draws inspiration from classical sprite-based game designs, such as those described in Will Green's "Hardware Sprites" methodology [1]. Key goals include:

- Efficient rendering of piano tiles using hardware sprites.
- Real-time synchronization between game visuals and audio cues.
- Scalability to support dynamic game elements and potential resolution enhancements.

Each piano tile is represented as a bitmap sprite for independent movement and rendering, with the tiles appearing depending on audio cues. Since the piano tiles are bitmaps, this made it easier also to be able to change the look of the tiles by having different bitmaps depending on the desired output. This was also done using a JPG to REM tool which converted a JPG image to a 16-bit bitmap that we could use to "draw" the tile.

B. Hardware Setup and Integration

The visualization system is implemented on an FPGA platform and integrates the following hardware components:

- **HDMI Display Module:** Generates high-resolution output (640x480) and precise synchronization signals (horizontal and vertical sync).
- **Sprite Engine:** Handles efficient rendering and movement of piano tiles using hardware sprites to minimize redundant screen updates.
- **Bitmap Storage:** Pre-defined bitmap patterns for tiles and other visual elements are stored in ROM, loaded during initialization.
- **BRAM Framebuffer:** Stores pixel data for the current and next frame, using double buffering to eliminate tearing and ensure smooth animations. This wasn't able to be achieved for this project, but would prevent tearing in future improvements of the project.

C. Technical Overview of Rendering

The rendering for the falling tiles in our game were implemented using block RAM (BRAM) within the FPGA, with one bitmap configured to hold pixel data of 12-bit depth.

The rendering process follows these steps:

1) **Sprite Positioning:** Audio cues determine the X and Y coordinates of each tile, which are mapped to corresponding pixel positions in the framebuffer.
2) **Tile Rendering:** Sprite bitmap data is fetched from ROM and written into the active framebuffer. A simple blending operation ensures that overlapping tiles are rendered correctly.
3) **Background and Effects:** Additional elements, such as the background and scoring effects, are drawn into reserved regions of the framebuffer. For instance, a separate memory region handles dynamic score updates, which are overlayed during the final composition could've been done.

V. Dynamic Tile Placement and Interaction

Tiles are dynamically positioned based on real-time audio signals. These signals are mapped to one of 12 positions across the horizontal axis, representing musical notes. Tiles corresponding to higher-pitched notes appear in the rightmost columns, while lower-pitched notes are placed on the left.

VI. Challenges and Solutions

Key challenges and their resolutions include:

- **Synchronization:** Ensuring precise timing between the audio module and visualization was addressed through shared clock domains and handshake signals.
- **Resource Optimization:** Efficient BRAM utilization was achieved by just using one BRAM bitmap for all the tiles.
- **HDMI Timing Accuracy:** The rendering pipeline was carefully aligned with HDMI synchronization signals to maintain a consistent refresh rate.

VII. My Contributions

One of the biggest challenges was creating a reliable clock wizard that generated the necessary clocks (100 MHz, 25 MHz, 75 MHz, and TMDS) for different parts of the project. Initially, the keyboard wouldn't interact properly with the display due to synchronization issues. The clock wizard I developed resolved this by ensuring all modules operated with the correct timing rather than relying on pipelining alone.

I worked on modules split into three parts: static tiles, falling tiles, and game logic:

- **Static Tiles:** These were easier to implement as they did not require BRAM. Instead, I used 'hcount' and 'vcount' along with the tiles' 'x' and 'y' coordinates to color and activate them.
- **Falling Tiles:** The falling tiles involved several submodules: 'TileBitmapBRAM', 'TileRenderer', and 'TileManager'.
  - **TileManager:** Managed signals from the audio module and activated new tiles. Handling multiple tiles in the same column efficiently was challenging. I used a 2D array of active tiles and a counter to track the number of active tiles per column, allowing successful rendering of multiple tiles in the same column.
  - **TileRenderer:** Continuously checked the 2D tile array to render active tiles. Using 'hcount' and 'vcount', it placed tiles correctly on the screen.
  - **TileBitmapBRAM:** Stored bitmaps of tiles and provided pixel

VIII. Achievement of Goals and Potential Use Cases

A. Goals Achieved

The Gridthym project aimed to meet several key milestones, classified into minimum, ideal, and stretch goals.

1) Minimum Goals: Our primary goal was to synchronize keyboard input with the display. This was achieved by implementing a robust clock wizard that provided the necessary clock signals (100 MHz, 25 MHz, 75 MHz, and TMDS). Initially, the keyboard failed to interact correctly with the display due to improper synchronization. The clock wizard resolved these issues by ensuring that all components operated in sync, thereby meeting our minimum goal.

2) Ideal Goals: We aimed to enhance the visual experience by adding multiple sprite flags, such as opacity and background graphics. This required precise timing control to ensure smooth rendering of sprites without any visual glitches. By meeting stringent timing requirements, we achieved a high-quality visual output that exceeded our basic requirements.

B. Potential Use Cases

Beyond the current implementation, our design can be adapted for various applications with minimal changes:

- **Expanded Game Mechanics:** Adding new game elements such as power-ups or animated characters can be easily integrated by leveraging the existing sprite management system.
- **Educational Tools:** The system can be adapted for interactive learning tools, providing real-time feedback and engaging visuals for educational purposes.
-

The minimum milestone was barely reached because the timing issues were underestimated and ended up causing more issues than anticipated.

REFERENCES

[1] F. Buss, "Yet Another Graphics Controller (YaGraphCon)," GitHub repository, July 2012. [Online]. Available: https://github.com/FrankBuss/YaGraphCon. [Accessed: Nov. 27, 2024].

[2] J. Matthews Group, "SD Audio for FPGA-based Projects," MIT Department of Electrical Engineering and Computer Science, 2019. [Online]. Available: https://web.mit.edu/6.111/volume2/www/f2019/tools/sd_audio.pdf. [Accessed: Nov. 27, 2024].

[3] W. Green, "Exploring FPGA Graphics - Framebuffers and Animation," Project F website, Jan. 2024. [Online]. Available: https://projectf.io/posts/. [Accessed: Nov. 27, 2024].