Music Visualizer and Synthesizer Final Report

1st Ann Liu
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
annliu22@mit.edu

2nd Anthony Acevedo
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
aace217@mit.edu

***Abstract*—We present a design for an interactive audio-visual system that uses real-time data acquisition and audio synthesis to replicate a Musical Instrument Digital Interface (MIDI) piano with visual feedback. We implement this design using a hardware stack with two facets: MIDI intake / audio processing to play notes and object tracking / video processing to set the beats per minute (bpm). The hardware stack interfaces with a MIDI device to receive data, a camera to track a baton, and a 720p HDMI monitor to display data.**

***Index Terms*—Musical Instrument Digital Interface, Digital Systems, Digital Signal Processing, Object Tracking, Real-Time Audio Visualization**

I. The Problem

Real-time audio synthesis and visualization interfaces are difficult to create purely in software because of the overhead of higher-level programming languages. Therefore, we have developed a hardware-based real-time audio visualization and synthesis platform to provide valuable feedback to the user in a professional music or educational setting. Our design is intended to provide this audio and visual feedback in a understandable and intuitive way while allowing itself for expansion. The difficulty with addressing this problem is two-fold: the audio is synthesized from scratch, and we are not using a pre-existing music display driver to draw the notes in the proper location.

As a refresher, the requirements of the project were the following: decoding MIDI messages, using a baton to set BPM, using BPM, integrating multiple sounds to create a "summed" PWM, and integrating the audio/video into a unified system. Our design has met these requirements because instantaneous audio feedback is provided for up to 4 notes at once, and instantaneous visual feedback in the form of a staff that demonstrates the note being played.

II. High-Level Explanation of the Design

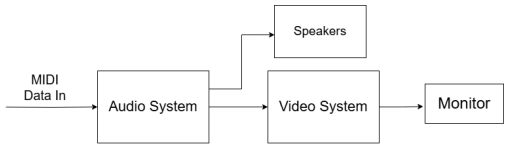


Fig. 1. A high level view of the system

At a high level, our project takes in MIDI data from any kind of MIDI device, and it parses it into a sound and visual representation on a musical staff. The key module on the audio side is `pwm_combine`, and this effectively generates the noise. On the other hand, the key module on the video side is `note_storing_run_it_back`, and this effectively draws all the notes. Innovative techniques of our design include the real-time note detection and display, lightweight audio synthesis, and dynamic note generation on the display.

III. MIDI Keyboard Interfacing & Synthesizer

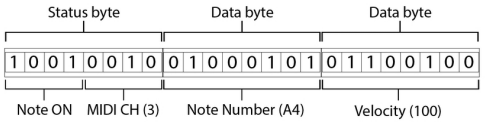


Fig. 2. The breakdown of the bytes within a midi message. [3]

MIDI is the digital protocol that we are working with, and it is a specific version of UART (Universal Asynchronous Receiver/Transmitter) that typically transmits 3 messages in a burst at baud rate of $31,250$ Hz [1]. Some devices, like the keyboard we are working with, send messages known as active sensing and clock messages. These messages do not adhere to the previously described packet structure. Rather, they are only 8 bits (10 including UART start/stop) [1], and this presented some challenges because they are typically sent more quickly than the note data. In total, the entire MIDI message packet that we are working with consists of 30 bits, and the breakdown is as follows: 6 UART start/stop bits, 8 velocity bits, 8 note number bits, 4 channel bits, and 4 status bits (on/off message type) 2.

The physical setup consists of the following parts:

1) MIDI Breakout Board
2) Perf board for wiring
3) MIDI interface cable
4) MIDI-OX Software
5) loopMIDI Software

Note: A schematic of the hardware setup is available in Section IX.

A. Data Intake

When a note in pressed on the keyboard, the note is sent over USB-MIDI (a different, more complex protocol) to a laptop. Inside the laptop, loopMIDI and Midi-OX route the

data to the standard MIDI Out cable for receipt by the FPGA. On the FPGA, several things occur:

- The `midi_decode` module receives the start bit of the first UART message; then, it decodes the message. If this message is not a note on or note off message, then the message is ignored, and the module returns to the IDLE state. (This logic ensures that the active sensing and clock messages are ignored.)
- If the message is valid, then the module awaits for the rest of the information: the note and its velocity which are encoded in the next 20 bits (2 UART packets).
- Once this data is received, it is sent to the `midi_burst` module where it is packaged with other data for further processing.
- Within `midi_burst`, a 5 element "note" array is updated each cycle if new data comes in from `midi_decode`. Additionally, a 5 bit valid array is output by the module to indicate which indices have valid note data. If the message received from `midi_decode` is a note off, then the valid array bit gets set to zero, and the data in the array at that index is set to zero. On the other hand, if a note on message is received, then the module searches for a "free" index from MSB to LSB in the valid array to place the note. For instance, if the valid array is: `5'b11010`, then the data will be placed in the 2^{nd} index, resulting in: `5'b11110`.
- The data from `midi_burst` is received by `pwm_combine`, and it is parsed as described below:

  1) A value, n, between 0 and 127 (inclusive) is received by the module.
  2) $n \mod 12$ is calculated by continuously subtracting 12. Each time 12 is subtracted, a counter m increments by 1.
  3) In the end, the result of $n \mod 12$ is the note number between 0 and 11 that is sent to `pwm_combine`, and m is the octave between 0 and 10.
- Within `pwm_combine`, the data is sent to the `sine_machine` module for audio synthesis, and it also outputs an 5 element array for usage by the video pipeline.l

At this point in the pipeline, the data goes in two directions: the visual half of the system and the audio half. The visual half of the note's traversal will be discussed in the next section.

B. Audio Synthesis (3)

The audio synthesis is done purely within `sine_machine`; however, this signal remains internal to `pwm_combine` as it is responsible for adding the signals. A high level overview is given below:

- In this `sine_machine`, sine data stored in flash, and it is read from a BRAM at a rate determined by the note and octave, and this data is sent back to `pwm_combine`. This process will be discussed in further detail below.
- Within `pwm_combine`, the appropriate number of sine waves are combined through weighted signed addition depending on another form of data from the MIDI format. Finally, this result is sent to the PWM module. The final PWM signal comes out of the PWM module and into the speaker, completing the audio system.

To expand upon the sine generation, the sine wave stored in flash is a 440 Hz A4 note with 8000 samples. Moreover, each sample is only 8 bits. To read a certain note, each sample from the sine wave needs to be held for a certain amount of time. Therefore, samples from the sine wave can only be extracted every x cycles. This value depends on the clock frequency, sample rate, and frequency of the original sine wave. Its formula is given below where f_{clk} is clock frequency that the module is run at and f_d is the desired frequency of the resulting sine wave.

$$x = \frac{f_{clk} \cdot 440Hz}{(8000 \text{ samples}) \cdot f_d}$$

As an example, if we wanted to read the 440 Hz sine wave back from flash, x would be equal to $12,500$. A key aspect of this relationship is that this computed value x can be shifted to the right or left to change the octave and correspondingly change the frequency. Going back to the example value of $12,500$ if this note is actually an A5, then the module will read every $6,250$ cycles to create the higher frequency note.

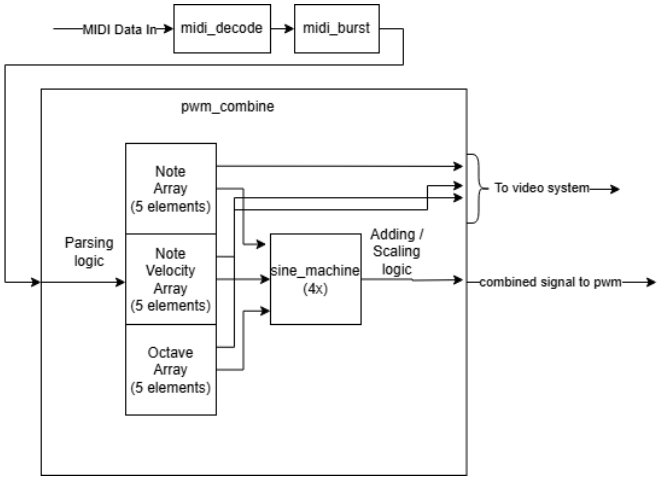


Fig. 3. The MIDI Data pipeline modules with abstracted inputs and outputs

Once the four sine waves are generated 3, they are combined in a weighted sum. As part of the MIDI protocol, there is an attribute called the note velocity, and this is a value that ranges from 0 to 127. For each note, each value of the sine wave is multiplied by this value then normalized (divided by 255 through a 7 bit right shift).

IV. Visual Input & Pipeline

A. Visual Input/Outputs & Video Mux

Our system has a visual output that is controlled by the `note_storing_run_it_back` module. As notes are

received from the keyboard and MIDI inputs, they will be displayed on a staff. Notes will be able to be overridden and changed; multiple staff lines to be played simultaneously are planned to be added as well.

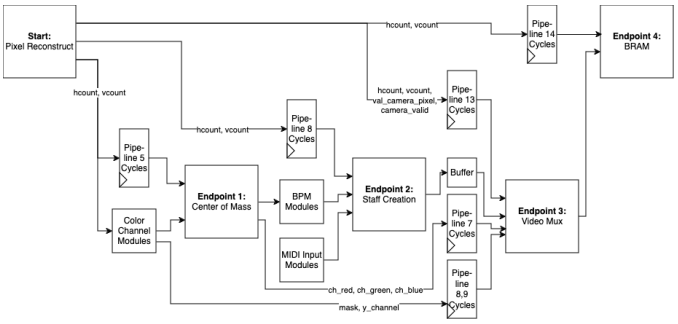


Fig. 4. Simplified Block Diagram With Pipelines

B. Pipelining

Due to the time-sensitive nature of the video pipeline, all the modules need to be pipelined properly as shown in figure 4. Moreover, the WNS and TNS were consistently an issue for us during the design. Therefore, pipelining was integral. The four pipeline endpoints needed are at Center of Mass, Note Storing, Video Mux, and the frame buffer BRAM modules; these modules all require the use of signals from pixel reconstruct as well as from downstream modules.

C. Beat Detection

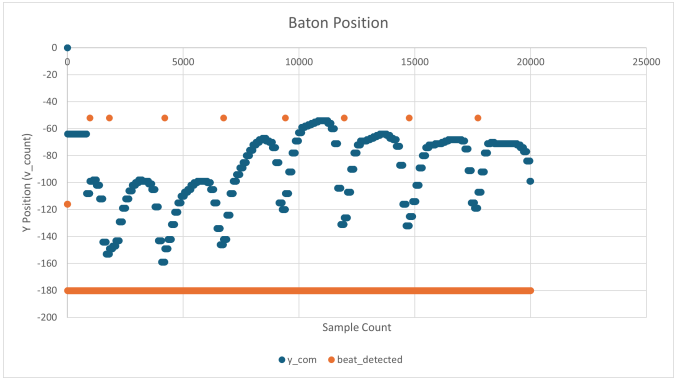


Fig. 5. Plotted Y Position and Beat Detected

The baton tracker module takes the y position from the center of mass module to detect extrema as detected beats. A beat is detected when the y position has a 0 first derivative and a positive second derivative; in other words, when the baton goes down then up. In practice, the output of the beat detection module is set high when the previous change in y position is negative while the current change in y position is positive.

Because center of mass is continuously calculated in real time, the y position may marginally fluctuate even in a still frame. These fluctuations are filtered out by restricting the aforementioned conditions: if the previous change in y position is a negative number less than -2, and the current change in y position is a positive number greater than 2, then the output is set high; the threshold value 2 was determined empirically. With these restrictions, small fluctuations in change in y position from -2 to 2 are not considered as valid detected beats. If the current change in y is within this invalid range, the module waits until a valid change in y position value is received to replace the previous change in y position value.

Furthermore, mask color for the data that feeds in the center of mass module is hardcoded to find pink to match the color of the baton.

D. BPM Generation

The BPM generation module is a state machine with the states IDLE, OVERRIDE, BATON, and DEFAULT. The states are mainly determined by two FPGA switches encoding up to four states, with some exceptions.

The IDLE state remains the same unless the switch configuration changes. If the switches encode for the OVERRIDE state, the state transitions to OVERRIDE. However, if the switches encode for BATON to be the next state, the state transitions to BATON only if the immediate previous state was IDLE (in addition to the current IDLE state). This logic prevents the state machine from indefinitely cycling between IDLE and BATON as long as the switches encode for BATON, continually overriding previously-set BPMs. Each time the switches are changed to the BATON configuration, BPM can only be generated once until the switches are updated to another state, then back to BATON.

In the OVERRIDE state, BPM is set to whatever number is encoded in the top 8 switches on the FPGA.

The BATON state gives the user 15 seconds to set a tempo. The number of beats detected (by the baton tracker module) in the 15 seconds will be multiplied by 4 to attain the BPM (of the quarter note), then multiplied by 4 again to calculate the BPM of the $\frac{1}{16}$ note. At the start of this state, 15 LEDs light up. One LED turns off each second, visually counting down the remaining time. Once the countdown finishes, the state transitions back to IDLE, and a new BPM can only be generated by the baton if the switches are changed to something else, then back to BATON.

Lastly, the DEFAULT state is simply a convenience for the user to set bpm equal to a common BPM of 60.

E. Note Duration

Based on the note information received from the audio system, the duration of each note in the ”MIDI burst” is determined, and this data is forwarded to the note-storing module for drawing. This module is relatively simple in its logic compared to note storing, but it keeps a running cycle count of how long a certain note is held for. For instance, if A3 comes in one cycle, and is held for 100_000 cycles, then this module module will measure this. It stops its count if any of the four following conditions are met:

- The octave changes.
- The note itself changes.

- The note valid array at that specific note's index changes.
- The duration has passed a specific threshold of $\frac{db}{2} \geq 23\_999\_999\_999$ where d is the current duration and b is the bpm. Note that 24 billion cycles is the length of a whole note, and it is the longest note we are representing.

If none of those conditions are met for a given note, then the duration simply counts up for that note based on the data that comes from the MIDI system. Moreover, we will explain the third point more clearly with an example. Suppose there is a valid array of `5'11111`, and this module is currently determining the note duration for the fourth index. If the valid array on the next clock cycle is `5'01111`, then the duration count for the note at index 4 will be set to 0 on the next clock cycle.

F. Note Storing

This module receives the data from note duration, and it parses the kind of note based on this value. The duration for a quarter note is 60 sec · 100,000,000 cycles. One fourth of this value is the number of cycles for a sixteenth note. To compare the input from note duration to the thresholds, we would have to use division (comparing the input note duration in clock cycles to $60 \cdot \frac{100{,}000{,}000}{\text{bpm}}$), but we avoid this by comparing the input times bpm to the threshold by comparing in duration times bpm to 60 · 100,000,000.

Based on the rhythm of the note, we index into the note sprite image to store the correct type of note into the frame buffer to be drawn by the hdmi monitor; this note pixel data is muxed with camera data, so whichever can be displayed on the monitor

Additionally, based on pitch, using the note C8 as the top note that is can be displayed on the screen, we can calculate the y position of the note. Each note slot on the staff is 6 pixels, and there are 7 possible note locations (as sharp notes are on the same position as their non-sharp counter parts). The octave determines how many repetitions of 6 pixels · 1 octave · (7 notes) down to draw the note, and this is offset by the particular pitch.

Then, we use a state machine to take the input notes and store them into memory. First we initialize the background, then the staff with rests. Then, we wait remain in the IDLE state until we detect a note. A note can only be detected every 1,500,000,000 cycles (incremented by ¼ of the bpm) as this is the cell duration for a sixteenth note. In short, we can only detect a note once every sixteenth note. This is counted by staff cells; each staff cell represents a possible slot for a 16th note to be drawn; there are then 64 staff cells for 16 16th notes per measure times 4 measures. The staff cell is always looping through as places where detected notes can be drawn.

A note is only drawn if there is a change in rhythm (incoming detected is compared to a matrix). There are two cases: 1. The rhythm has changed to a 16th note which means that the note has turned into a new note (either pitch or octave has changed or note has turned off; or the note has maxxed out at a whole note), and 2. The rhythm has changed to anything else, which means that the current held note is being extended and continued to be held. In the case that the note is being held, we need to override the previously drawn note. We do this by not only keeping track of the current staff cell, but also the "start" staff cell, the cell where the note data is held. When a sixteenth note is detected, the staff cell that it is detected in is set as the "start staff cell" of that note. If that sixteenth note is held and extended into a longer rhythm, we can access this start cell and change it to store the data of an eighth note instead of a sixteenth; the rest of the staff cells between the current and start staff cell are filled with NULLS as to not repeat data.

Then we use this start staff cell as the starting location to draw the note (multiplied by 5 as each staff cell is 5 pixels wide).

V. Evaluation of the Design

The code of the design can be found here. General evaluation notes will be given then more specific details will be explained. To provide context for the time constraints for the audio side of the project, the clock that all the modules in the audio system run off of is the 100 Mhz clock, but the throughput of the data is restricted by the 31,250 Hz baud rate of the MIDI messages. This limitation, however, also provides plenty of leeway for logic because each bit within the midi message is held for $\frac{100\_000\_000}{31\_250} = 3200$ cycles. Therefore, the timing requirements for the audio aspect of the project are relatively relaxed compared to the video portion. Moreover, all of the MIDI files that were used to test as songs were from this website.

On the video side, the BPM module was tested by acquiring data and sending it over UART over Python. Furthermore, the note storage module was tested via test benching to ensure that notes with certain durations were properly held. Since the video aspect was visual, the majority of the testing was done on the monitor, and we ensured that notes, stems, dots, and sharps were drawn in the correct location.

Speaking about the entire system, we ensured that the audio matched the notes that were being plotted on the monitor. Additionally, we ensured that the specific MIDI messages from our digital keyboard were giving the audio and video pipelines the same data. We ran into an issue with a valid array not updating for the video system, and this was discovered through this evaluation.

1) Latency and Throughput: The latency and throughput are mostly limited by how fast the MIDI data can be acquired. Even though the FPGA's clock is running at 100 Mhz, the MIDI data is transmitted at a baud rate of $31,250$ Hz. There is some logic that takes a couple of clock cycles before this data is available. However, it is relatively small compared to the 3200 cycles that each bit in the MIDI data stream is held for. Moreover, since this is an audio / visual system meant to operate at the scale of human hearing and vision, it is noteworthy to point out that there is no detectable delay in the audio or video output. On the audio side, a very small overhead that we incur is combining the notes into "midi bursts". This takes a few clock cycles for extra logic and

valid data detection; however, it significantly expanded the capability of the project by allowing the system to generate audio and visual feedback from concurrent notes.

2) Memory and DSP Usage: The system uses 150 BRAMs and 120 DSPs. This resource usage could certainly be optimized further in a couple of ways. Firstly, there is an instantiation of the `sine_machine` module (which has a BRAM) that is not being used because only 4 sine waves are being combined. Moreover, in `staff_saver`, a module that we were not able to complete, there is some logic to reform the MIDI message and feed in back into the system. The DSP usage could be reduced by using the data provided to the module in a way that does not require all these computations to be redone.

3) Timing: As mentioned, this system is not meant to be a high-speed audio. video processing platform. As a result, keeping delays / timing issues under the human-perceptible level of delay (5 - 10 ns) was the goal. We fairly tightly adhered to this requirement but faltered slightly with our WNS of -1.385 and TNS of -84.324. Despite the negative slack, the audio and video systems did not have a perceptible issue with processing the MIDI data.

4) Use Cases: The audio aspect of the final product aligned strongly with the deliverables outlined in the project checklist because of how many notes that it is able to handle. The deliverable detailed acquiring, decoding, and synthesizing audio for 5 separate notes over the MIDI protocol. The acquisition and decoding were achieved for 5 notes, and this data is forwarded to the video portion of the project. However, only 4 notes were synthesized for the audio output to ensure that the combined signal (with all the notes) sounded well. More specifically, adding the 5^{th} sine wave was possible, but the volumes of the other 4 notes would have had to be reduced. However, it does not decode all the types of MIDI messages. For instance, pitch bends (which control the pitches of all the notes that follow) are not recognized. Other messages that were not decoded were notes on channels other than channel 0, and the timing clock messages (they were ignored). There were some minor aspects of the project that could have been modified to extend functionality. For instance, multiple MIDI channels could have been read from, and this could be achieved by having multiple instantiations of `pwm_combine` for each channel then summing the results. Furthermore, with the final version of `midi_burst`, the system could have been extended to handle up to 16 notes.

VI. Reflections

There were several aspects of the project that could have been improved upon. Firstly, we would have more thoroughly integrated the MIDI protocol into the project from the beginning. More specifically, we believe that interesting visual effects could have been tied with specific MIDI messages. For instance, the pitch bend MIDI message certainly could have been used to modify the pitch of the output audio, but we think that it could have been interesting to introduce visual distortions based on the pitch bending. Furthermore, there could have been multiple staffs that output the note based on the MIDI channel that the information was received on. As a general note, we would have started the project earlier to implement more of these features.

VII. Contributions

The project was mostly divided along the audio and visual lines. Anthony was responsible for the MIDI data intake, including understanding and mapping out the breakout board and getting the data from the MIDI device. During the initial stages of the project, he conducted research regarding the different types of MIDI messages, and he worked on viewing the digital signal on an oscilloscope. Moreover, Anthony also organized the MIDI data into a data structure for usage in the video pipeline. With the MIDI data, Anthony extracted the note and octave and used this information to generate the frequencies of each note. He generated these notes using his sine generation module. Then, he combined the outputs of multiple of these modules to provide the final audio output for the project. Finally, he was responsible for the initial draft of a notable portion of the final report and the LaTeX formatting of the entire document.

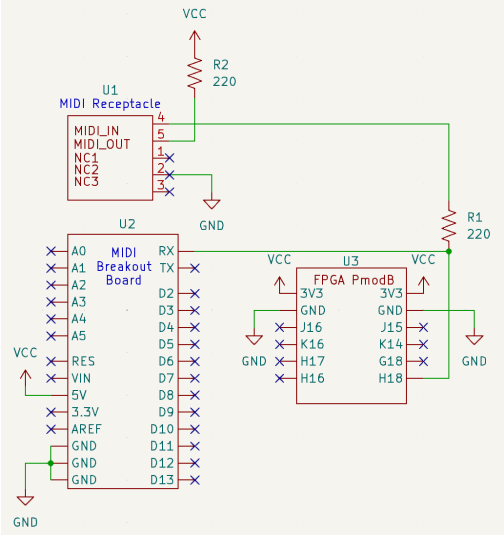
On the other hand, Ann was responsible for two things related to the visualization of the notes: she worked on the baton tracking module to enable the interactive setting of the BPM. By recording and counting the extrema of the path of the baton over a set period of time, she calculated the beats per minute. Additionally, she received the MIDI data from Anthony to draw the notes on the HDMI output. Given a note, Ann first kept track if its continuous duration; based off the length of the duration, she assigned a rhythm to the note. Additionally, the note pitch and octave were used to determine the y position of the note. The rhythm and pitch were then used to index into a note sprite map and save the right note sprite to memory to display.

This was accomplished by indexing into a note sprite map that she created. Furthermore, she developed the logic for determining the type of the note (whether it was a sixteenth, eight, fourth, etc.). Beyond developing this logic, she also worked on placing the notes in the correct location on the staff depending on the note (A,B,C, D#, etc..) based on its octave. To emphasize, drawing the notes on the HDMI output was not a straightforward task because of the depth of the project. The notes dynamically changed, sharps were included, stems were also drawn, and the notes were placed in the correct place on the staff. In short, she developed the machinery to "splice" the sprite map of all the notes, stems, and sharp symbols and output them in the correct place on the staff.

VIII. Acknowledgments

We would also like to thank Eran Egozy for graciously providing us access to a MIDI keyboard for preliminary testing. Additionally, we would also like to thank Joe Steinmeyer for assisting during the ideation, implementation, and creation of both aspects of the project. Moreover, we would like to thank Justin Malloy for assisting us with recording our video.

IX. APPENDIX A - MIDI DATA RECEIPT SCHEMATIC



REFERENCES

[1] MIDI Assocation. (2024, February 14). Summary of MIDI 1.0 messages. MIDI.org. https://midi.org/summary-of-midi-1-0-messages.

[2] Online Sequencer. (2024, December 11). onlinesequencer.net.

[3] MIDI Explained. (2024, December 11). Pirate MIDI. https://learn.piratemidi.com/learn/lesson/1-midi-explained