FPGA Omnichord

1st Rachel Chae
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
rchae@mit.edu

2nd Keilee Northcutt
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
keilee@mit.edu

***Abstract*—Inspired by Omnichords in the '80s which has since been discontinued, we propose a design for a music synthesizer. Our implementation uses input from the keyboard to select a chord, then uses input from an external ribbon touch sensor to select exactly which note within the chord the person is playing. The touchpad automatically updates with chord selection to reflect 12 notes in that chord progression, making it easy to hit notes in harmony. The use of the SD card and incorporation of an external touch sensor adds to the technical complexity of the project and producing audio consistently without phase error or a significant delay is a non-trivial challenge to tackle. We used the functionalities of the FPGA to allow note changes at every 0.25-second interval and produce continuous, in-phase notes. The instrument is also capable of playing two notes simultaneously to create 3rd and 5th harmonies depending on the FPGA switch input.**

I. High-Level Overview

Within our system, we use PS2 Decoder, Ribbon Stabilizer, and Ribbon Decoder modules to process the inputs from a keyboard and a softpot ribbon sensor. The softpot ribbon sensor uses a system of potentiometers to interpret the coordinate of a touch on the surface of the ribbon. The Ribbon Decoder and Ribbon Stabilizer translate a touch on the sensor into a single stable value out of twelve sections along the ribbon.

The Note Selector module takes the output from the Ribbon Stabilizer and the PS2 Decoder to interpret the two parts of the note (the chord and the note within the chord). Then, the Note Selector converts them into an address that corresponds with the chosen note's placement in the .wav file on the SD card.

The SD Reader functions in two ways; it either reads one note, or it can blend two notes depending on the values of the switches on the Xilinx board. This is possible through the use of two FIFOs. The FIFO is a First In First Out queue which outputs the data from the SD card at the appropriate 44-kilohertz sample rate. In our implementation, the two FIFOs can either both store values of one note or store two separate notes. If using two separate notes, the notes are blended by dividing them and then adding the values before being passed to the PWM module. Within the SD Reader Module, the SD Controller module is used to read the note from the SD card.

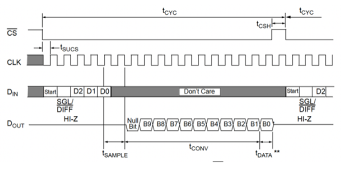

Fig. 1. Communication protocol used to retrieve sensor signal from MCP3008.

This note is then output at the appropriate speed through the FIFO module to the PWM module.The FIFO's output is sent to the Xilinx board's mono audio with the PWM module which uses pulse width modulation.

II. Physical Construction

A. Softpot Ribbon Sensor

The softpot ribbon sensor was connected to a resistor and then wired to channel 1 on the MCP3008 10-bit A/D converter as shown in Figure 1 [3]. Then, the MCP3008 was used to convert the analog signal to a 10-bit digital signal and communicate it to the FPGA through external PMOD ports [1].

III. Decoding

A. PS2 Decoder

The PS2 Decoder module was used to decode outputs from the keyboard. Its implementation was kept mostly the same from Lab 2. A small adjustment was made in the top level module to disregard 0xF0 values, which signals a key being unpressed. This modification allowed the FPGA to retain the value of the previous key until a new key was pressed.

B. Ribbon Decoder

The Ribbon Decoder module was used to communicate with the MCP3008 and decode the outputs of the softpot ribbon sensor. A 200 kHz clock cycle was used to communicate with MCP3008. Every 20 clock cycles, the CS value was set to zero and the FPGA sent out 5'b11001 sequentially from the most significant digit to request a single-ended reading from channel 1. Then, MCP3008 outputted a null bit followed by 10 bits of reading from the ribbon touchpad, sent starting from

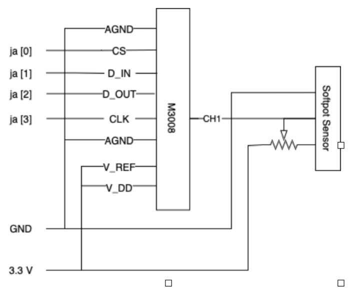

Fig. 2. Circuit diagram for Softpot Ribbon Sensor and M3008 A/D converter.

the most significant bit. The values were stored in a buffer until all 10 bits were received, at which point the value was discretized from 0 to 12. Then, this value was channeled to the stabilizer module. With these conditions, the FPGA gets a new value from the touchpad at the rate of 10 kHz.

C. Ribbon Stabilizer

To reduce fluctuations in readings from the ribbon sensor, a stabilizer module was implemented. The Ribbon stabilizer takes in outputs from the ribbon decoder module on a 200kHz clock and outputs the maximum value of its readings every 4096 clock cycles.

IV. Note Selection

The Note Selection module runs on a 100MHz clock. It takes the outputs from the PS2 Decoder and Ribbon stabilizer in real-time and calculates the address of the selected notes in the SD card. For instance, selecting U on the keyboard and 1 on the touch pad yields inputs of 0x3C and 0x2, respectively. Then, those inputs combine in the note selector module to output 0x5800, the starting address of the C#4 note (first note of the A chord) in the SD card.

V. SD Card

A. Audio Synthesis

We started by using the online catalog of omnichord sounds on Online Omnichord but later switched to synthesizing our own audio. Synthesizing our audio using sine waves in python allowed us to generate in-phase audio, which in turn allowed for the continuous playing of notes. Each of our notes was generated to be .wav files of 0.25 seconds in length. We set our audio-generating python script to calculate the number of full-wave periods that can fit in 0.25 seconds and set the audio value to -32768 once the maximum number of periods was reached. This ensured that the out-of-phase audio values are 0 once the .wav files were converted to an Unsigned 8-bit PCM format using Audacity.

After converting the audio files to an appropriate format using audacity, the individual notes were stitched together to store on the SD card using the AudioSegment library in Python. We included 27 chords and 12 different notes for each chord, resulting in a total of 324 notes or 81 seconds of audio [2]. The SD card is sectioned into chunks of 512 bytes, therefore, the SD Controller module will only begin reading

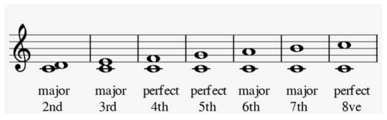

Fig. 3. 3rd and 5th harmonies in music theory

at an address that is a multiple of 512 (since it is designed to read out full sectors). Because of this, we padded our singular .wav file with zeros so that our project uses to begin each note at the next available address which is a multiple of 512. This ensures that every note can be accessed in full by the SD Controller. Once all audio processing steps were completed, we wrote the .wav file containing all notes directly to the SD card using a laptop.

B. Alternating SD Reading for Harmonies

To play multiple notes at the same time, we changed the SD Reader to alternate the address it's reading every 512 bytes. The readings are then stored in two different FIFO queues, which are described more in detail in a later section. The FIFO to which the SD card reading is stored is also switched every 512 bytes, creating one FIFO that only contains audio from note 1 and another that only contains audio from note 2. By switching between the address for note 1 and note 2, we're able to create harmonies without using an additional buffer to store audio data.

When no switches are on, the SD reader only reads data from one address on the SD card, producing one consistent note. When the sw[1:0] is 1, however, the SD reader alternates between reading from the current note address and the address of the note adjacent to it in the chord progression, creating a major third harmony. Alternatively, the SD reader creates a major fifth harmony when the sw[1:0] is 2 (Figure 3).

C. SD Controller

We use the SD Controller module written by Jonathan Matthews [5] in order to facilitate reading from the SD Card through a Serial Peripheral Interface (SPI). The main channels to and from this module that we use for reading are - rd, ready, byte_available, address, and dout. We are able to trigger reading from the indicated address on the SD card by sending rd (read enable) high whenever the module indicates that the SD card is ready to read or write. We can then use byte_available to figure out when a new byte has been outputted (on dout).

D. SD Reader

The SD Reader module uses the SD Controller module along with modules Divider and PWM in order to read out a full note from the SD card and output it through the mono audio channel on the Nexys. The Divider module is used to generate a 25 MHz clock for the SD Controller module. Essentially, the module as a whole looks for the SD Controller module to be ready and then triggers reading at a valid address. A single note is approximately 22 sectors (with 512 bytes being a sector) in total. Two counters are used to keep up with where the SD Controller is in the reading process to ensure that the address is increased by 512 after an entire sector is read, keeping the reading process progressing through an entire note. On every clock cycle, the module checks for the rising edge of the byte_available signal from the SD Controller to ensure we don't read from the same address multiple times, and when we see the rising edge of byte_available, we send the new byte of data to the PWM module to output audio [4].

E. First In First Out (FIFO)

The FIFO modules were generated using the Vivado IP Catalog to properly pipeline the data from the SD card at the correct rate for our .wav file sample rate of 44 kilohertz into a buffer of depth 2048 bytes. To output at the proper rate, The FIFO module reads out a byte of data every 2268 100 MHz clock cycles. Without this module, the data is read from the SD card much too quickly to be recognized properly. When generating harmonies, the output value from the two FIFOs are divided by two then added together to mix the two notes.

Furthermore, the SD reader is configured to only write new bytes into the FIFO if the new bytes are not zero. Since we had synthesized the audio .wav files to default to zero when the audio waves are at the end of a period. Since all the zero bytes that come after an in-phase note are discarded we can play a continuous note indefinitely without any out-of-phase artifacts.

VI. Mono Audio

A. Pulse Width Modulation

We used a simple PWM module to generate output for the mono audio channel on the Nexys. The module has an 8-bit counter (counting up to 255 clock cycles for each PWM period) which is increased by one every clock cycle. When the newest audio byte is passed into the module, it compares it against the counter. If the counter is greater than or equal to the audio byte's data, the module sends out a low signal. Otherwise, it sends out a high signal. Within the Top Level SD Reading module, if the signal from the PWM module is low, a 0 is sent through the audio channel, if it's high, a signal of high impedance (Z) is sent [6].

VII. Timing Requirements and System Limitations

The maximum amount of notes our system can play at one time is 4. This is because the FIFO outputs every 2268 clock cycles, and in modes where multiple notes are played in harmony, our design uses an individual FIFO for every single note and will only output to the PWM module if all FIFOs have data inside of them when it is time for the FIFO to read out. Therefore, the max number of notes that can be played at once without a delay in audio output is 2268/512 since 512 bytes per note are written to each FIFO at the time which takes approximately 512 clock cycles. Implementing 4 notes in harmony would cut timing as close as possible with it taking 2048 clock cycles to store the first sector of each note, closely cutting the FIFO's 2268 clock cycle output.

The bottleneck of our design is the sample rate of the .wav files stored on the SD card. The sample rate causes the FIFO queue module to take more clock cycles to output than any other component of the system, buffering for 2268 clock cycles. We could use a higher sample rate which would increase the FIFO's output rate. However, a higher sample rate would not necessarily improve the user experience—while it would allow extremely higher frequency notes to be played without aliasing, those notes are outside the notes contained in standard omnichord and piano catalogs and would be included in our instrument. Their extremely high pitch would also make them unpleasant to hear.

Another bottleneck is the speed of communication between the MCP3008/ribbon sensor and the FPGA. Currently, the values from the ribbon sensor are being decoded at 10 kHz. However, the maximum possible rate is 200kHz/15 = 13.3 kHz assuming that the readings from the ribbon sensor arrive on the clock cycle after communication from the FPGA is made. We chose to wait for an extra 5 clock cycles to provide some buffer room if readings are delayed. Then, the ribbon stabilizer module outputs a stable reading every 4096 values, which further limits the sensitivity of the sensor to detecting 2.44 readings per second. However, these limitations do not significantly bottleneck our device since users are unlikely to move their fingers and switch notes at a faster rate than the sensor can interpret them.

VIII. Design Evaluation

Our goal for this project was to build a working synthesizer that takes inputs from a keyboard and a touch sensor to generate audio. We wanted to use the SD card to save audio data and withdraw specific notes, as well as ensure that note switches happen in real-time without significant delay. We believe we accomplished our original goals: our FPGA omnichord decodes information from the keyboard and the ribbon sensors to select a note in real-time. Our FPGA omnichord allows note changes at every 0.25-second interval and can produce continuous, in-phase notes. Furthermore, we reached two of our stretch goals as we synthesized our own sine wave audio in python and created an alternating SD reader module to play two notes at a time.

Our implementation only used 1 BRAM (0.74% utilization) and 2 DSPs (0.83% utilization). We also implemented a 2-step pipeline to optimize the SD reader and obtained a positive slack value of 2.609 ns. The alternating SD reader method already allowed us to avoid the use of unnecessary buffers and minimize memory usage, which may explain the low memory usage values. We discussed earlier that one of the limitations of our design is that we can only play up to four notes as is. Since we still have the majority of our FPGA's memory available, we could create more harmonies in the future by creating additional BRAMs to store note information.

IX. Future Work

One additional functionality we could add with minimal changes to the overall design is to display the note being

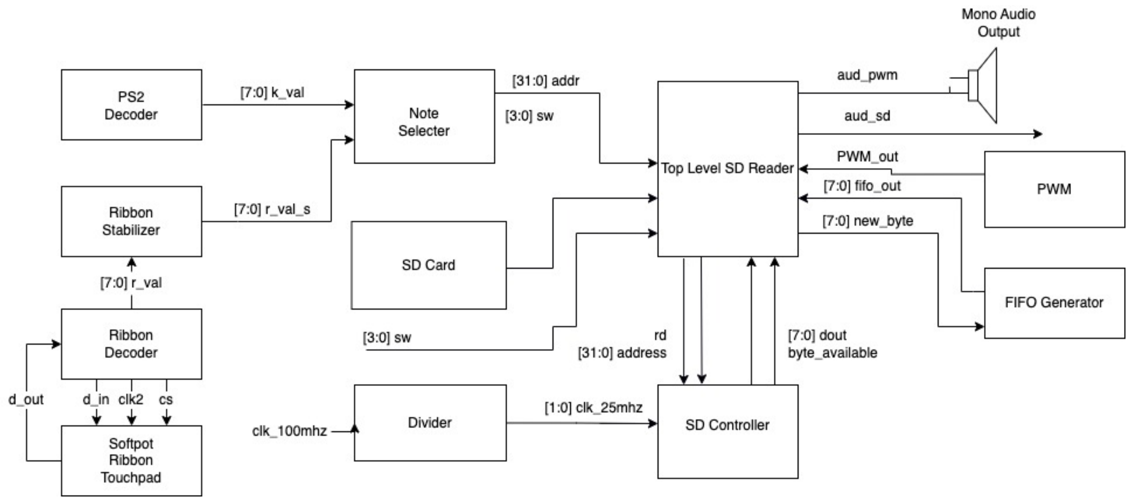

Fig. 4. system block diagram

played on the seven-segment controller. Additionally, we could add more harmonies by playing 3 or 4 notes at a time. For instance, a 4-note harmony would be implemented by alternating between 4 addresses on the SD card and instantiating 4 different FIFOs to store audio data. Given more time, we could add multiple instrument sounds by synthesizing them in Python, storing them on the SD card, and linking appropriate addresses in the Note selector module.

X. Distribution of Work

Rachel worked on incorporating the ribbon touch sensor and the keyboard to the FPGA, including the Ribbon Decoder, Ribbon Stabilizer, and the Note Stabilizer modules. She also synthesized the sine wave audio in Python, which were eventually stored in the SD Card. Additionally, she worked on alternating the reading of the SD card between two addresses so that more than one note could be played.

Keilee worked on reading from the SD card and outputting the audio at appropriate rates, which included the Top Level SD Reader Module, the FIFO module, and the PWM module. She also worked on the switching between FIFOs in order to mix notes together.

XI. URL to Code

https://github.com/rchae01/rachel_keilee_finalproject.git

References

[1] "2.7V 4-Channel/8-Channel 10-Bit A/D Converters with SPI Serial Interface". https://ww1.microchip.com/downloads/aemDocuments/documents/APID/ProductDocuments/DataSheets/21295d.pdf.
[2] "Online Omnichord." Online Omnichord, https://onlineomnichord.com/.
[3] "Spectra Symbol Softpot — Soft Membrane Potentiometer". https://www.spectrasymbol.com/product/softpot/.
[4] "SD Card Tutorial" 6.111 Starter Designs
[5] Jonathan Matthews, "sd_controller" https://github.mit.edu/6205/starter_designs.git
[6] "Pulse Width Modulation". Alchitry. https://alchitry.com/pulse-width-modulation-verilog.