6.111 Final Report — MemSynth: Exploring Digital Memory As A Multimedia Instrument

Jesus Diaz
Massachusetts Institute of Technology
jrdiaz@mit.edu

Artem Laptiev
Massachusetts Institute of Technology
laptiev@mit.edu

***Abstract*—We present MemSynth, an instrument for digital memory exploration. Departing from conventional file-based interactions, MemSynth invites users to explore raw memory data from an SD card by selecting arbitrary memory regions with switches and buttons. Powered by an FPGA, the device oscillates the selected data streams, outputting them as auditory soundscapes and visual representations. Creating new relationships between memory and perception, MemSynth challenges traditional paradigms of synthesis and offers a unique platform for exploring digital memory as a multidimensional, creative medium.**

I. Block Diagram

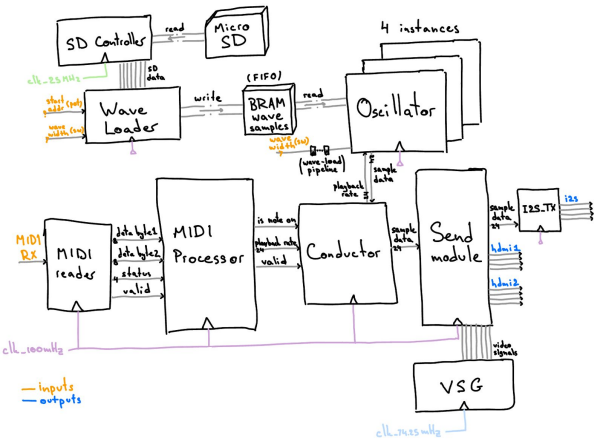

Fig. 1. **Updated block diagram for the final version of the device.**

II. Physical Construction

- K194725ASJH MIDI Shield to convert MIDI messages to a UART stream of data we can feed to our FPGA.
- CJMCU-1334 DAC Module to convert the output from our FPGA to line-level audio.
- Arturia KeyStep 37-Key Midi Keyboard Controller
- Urbana Board with Spartan 7 Series FPGA, by AMD/Xilinx.

III. Audio and MIDI

MemSynth implements classic features found in a conventional digital synthesizer:

1) MIDI reading and processing
2) Polyphony (with volume stabilization)
3) I2S audio output

A. Block Diagram

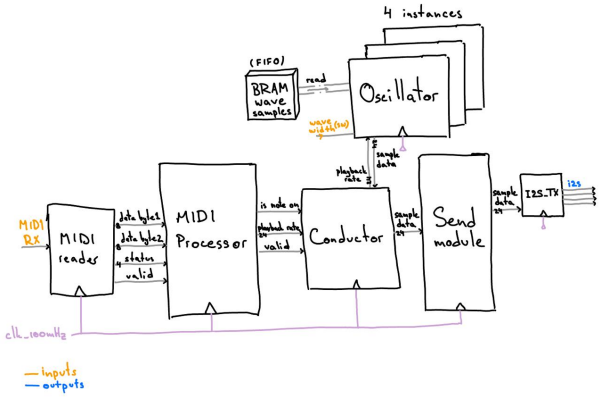

Fig. 2. **Updated block diagram for V0.**

B. MIDI Processor

To begin, we quickly highlight a design change in our MIDI Processor from the initial proposed block diagram for version 0 of this project.

Our original design featured a single MIDI module that would be fed UART data from our MIDI breakout board, and would output note on/off messages with their corresponding pitches. The MIDI protocol however, is capable of transmitting a variety of other information for events including pitch modulation and key press velocity. Because of this, we decided to split this module into two modules: (1) a general MIDI reader that is responsible for taking in raw UART data and outputting status and data bytes for MIDI events, and (2) a MIDI processor that takes in this data from the reader, ignores all data unrelated to note on/off events, and just uses the relevant events to output noteOn and pitch data. In this way, we can generalize the reading of MIDI data in case we decide to expand functionality to more MIDI events in the future.

C. MIDI Receiver

MIDI (Musical Instrument Digital Interface) is a technical standard that allows electronic musical instruments to communicate with each other. Each MIDI event sends three packages via UART, with the first package describing the ”status” of the event (identifiable by a 1 in the MSB), and two subsequent

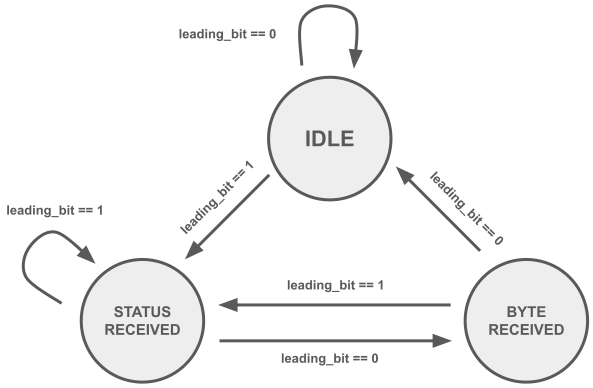

Fig. 3. **The MIDI state machine found in our MIDI Reader module receives incoming UART data from the MIDI instrument. Data is determined to be valid whenever there is a transition between the BYTE RECEIVED and IDLE states.**

data packages providing relevant information for the event. To make our system resilient to dropped MIDI packets, we implemented a state machine that serves as the intermediary between the coordinator for our oscillators and the incoming stream of UART data:

After receiving all required data, the state machine will set its valid data out bit accordingly.

D. Polyphony

For our version 0 of this project, we focused on implementing the two primary events found in MIDI: Note On and Note Off. When a key is pressed on our MIDI instrument, a Note On message is sent with the corresponding note number to our synth. The note continues to be played until the key is released and a corresponding Note Off event is sent.

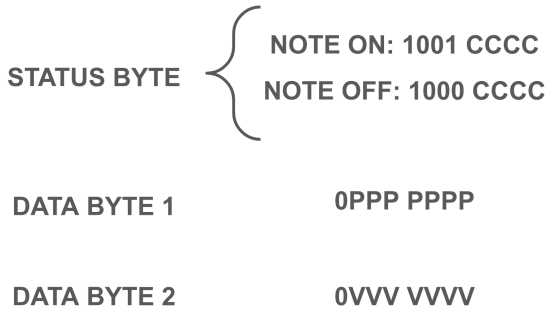

Fig. 4. **MIDI Note On and Note Off events with corresponding data bytes. The 7 lower bits of Data Byte 1 specify the pitch value and the lower 7 bits of Data Byte 2 specify the velocity value (which will be unused for now).**

Our original proposed design for this system utilized a monophonic synthesizer. Instead of receiving a Note On message and sustaining that note indefinitely, a monophonic synthesizer would have to overwrite the current note whenever a new Note On message was received. Such a design would have prevented users from being able to play multiple notes at the same time, and given the capabilities found on the FPGA, we decided it was worth redesigning our system to support this feature.

Our redesigned polyphonic synthesizer initializes a predetermined number of oscillator modules (adjustable via a parameter in our module) and coordinates between these oscillators and the incoming stream of messages from our MIDI receiver—mixing everything into the final stream of data that we feed into our I2S module.

To achieve proper coordination between modules we implemented a Least-Recently Used (LRU) approach to determine which oscillator should be overwritten in the case that we have more Note On messages than available oscillators. In addition to keeping track of each oscillator's active status and target pitch, we keep track of the "age" of each active module as a 32-bit integer. By incrementing this age for each active oscillator every clock cycle, we can then use this information to accurately determine which oscillator to overwrite with incoming Note On events.

With our current design, we have to be wary of the effects of overflow given that we are updating this value every clock cycle. Given our 32-bit age value, it would take $2^{32} = 2,147,483,648$ cycles for the age variable to overflow. Incrementing this value every clock cycle for a 100 MHz clock would equate to an update every 10 ns, so in total it would take $2,147,483,648 * 10^{-9} = 21.47483648$ seconds.

For now, we have determined that this is a reasonable time range for accurate LRU oscillator replacement. Given the fact that it is not necessary for this age value to be accurate down to the last clock cycle, we could very easily get away with instead updating this age every couple hundred clock cycles. Though it will be impossible for the user to determine a difference of less than a millisecond in key presses, it would present a noticeable hundredfold increase in our ability to accurately identify the earliest notes played on our synth.

Lastly, we highlight the need for a volume stabilization module, which was not included in our original V0 synthesizer. Our implementation of polyphony with multiple synthesizers created issues with drastic variations in volume and overflow due to the way we were directly adding the output from our individual synthesizers to get the final audio output.

To address this, we implemented the volume stabilization module, which serves as the middleman between our oscillator coordinator module and the output audio stream we feed to our I2S module. We receive a 18-bit audio stream from our oscillator module (scaled to accommodate the maximum possible value for the addition of 4 oscillators with 16-bit audio), and continuously feed this into a divider module, dividing by the combinationally-determined number of oscillators that are currently on.

E. I2S

To maximize the output quality of our audio, we have implemented an I2S protocol, passing the sample data to our DAC at 44.1 kHZ. To achieve this, we calculated the clock

(BCLK) we needed to drive the I2S at given our two channels (left and right) and our 16-bit audio:

$$\text{BCLK} =$$

$$(\text{number of channels}) \times (\text{sampling rate}) \times (\text{bits per channel})$$

Substituting the values:

$$\text{BCLK} = 2 \times 44.1\,\text{kHz} \times 16\,\text{bits}$$

$$\text{BCLK} = 1.4112\,\text{MHz}$$

In our implementation, we opted for a slightly higher clock of 2 MHz, given that it can be easily drawn from our 100 MHz clock.

IV. Memory Management

MemSynth's memory is divided into 2 parts:

1) "Main Memory" is the large pool of digital memory available for the user to explore. We implement it with an SD Card.
2) "Temporary Memory" is a BRAM that is used by our audio and visual oscillators, as well as our UART bytes explorer.

In order to achieve the desired effect of "playing" all the media files as audio as well as "displaying" the data as some imagery, we had to decide some constraints on what we consider to be a "sample" of data. Since the audio playback has been our focus for the project, we decided that each sample of data will have to be 16 bits in resolution. Therefore, we interpret each 2 bytes in our Temporary Memory BRAM to be a single sample, played out as audio and a single pixel shown on the screen of 720x360 pixel resolution in greyscale.

A. Memory Write

Every interaction of the user with the UI (changing the width of the selected data or the starting index) is followed by a rewrite of the corresponding section from the Main Memory into the Temporary Memory.

We use an SD controller to read 512-bits-long chunks of data byte by byte and update the corresponding positions in the Temporary Memory BRAM.

B. Memory Read

For quick access, our system stores data samples in BRAM, which is then continuously read by our oscillators. In our original implementation of the oscillator module, we had a dedicated BRAM for each oscillator. This was much simpler implementation-wise than coordinating with a single oscillator, but this greatly restricted our ability to scale the number of oscillators given the limited number of BRAMs on the FPGA.

To address this, we redesigned our oscillators to fetch data from a single BRAM, with a coordinator that would assign cycles based on a round-robin approach. We kept an array of oscillator addresses that would be updated by each respective oscillator module depending on its requested pitch frequency and a data array where samples from the BRAM would be assigned for each oscillator. We use an event counter that loops through our oscillator indices, and, with a one-cycle memory module, every cycle we update the data array for the previous oscillator and make a request for the current oscillator. Data is continuously read for the given values in the oscillator address array, so we use a separate isNoteOn array to indicate whether a sample should be included or ignored in our audio output.

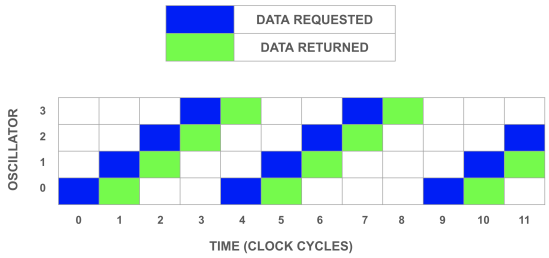

Fig. 5. **Round Robin coordination for oscillator BRAM usage. Data is returned one cycle after a request is made.**

V. Visual Representation

MemSynth outputs the sections of data, selected by the user, through 2 modes of visual representation.

A. UI Memory Exploration

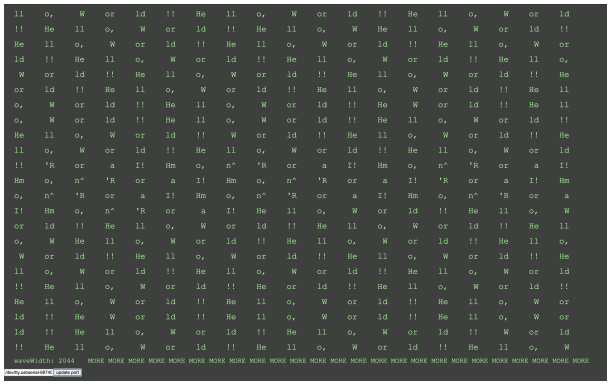

Fig. 6. **Raw bytes representation of selected data, using p5.js. 7-bit values decoded as ASCII chars.**

In order to provide an experience of closer familiarity with the digital memory for the user, we decided to implement a feature of displaying the raw data selection as text. Since we could only present a small fraction of the data selected as human-readable text, implementation of the graphics on the FPGA would be unnecessary. Therefore, we decided to pass part of the data from the FPGA over the UART to a PC, which implemented visualization using p5.js, a Javascript graphics library.

Since a number of variables had to be passed via UART (including the selected data wave length, playback indices of

the audio oscillators, and the data itself) all over a single port, we had to design a communication protocol between the FPGA and PC. A simple working solution was to append headers ”WAVLEN”, ”OSCIND”, or ”WAVDAT” before the corresponding data was sent.

B. Visual Representation

Among the stretch goals for this project was the design of a visual representation of the audio being outputted by our synthesizer. A simple way we were able to do this was by iterating through the audio data on our oscillator and feeding this directly to the red, green, and blue values being displayed on the screen. Making use of the existing HDMI framework we built earlier in the semester, we were able to generate some interesting output to complement the audio output of the synthesizer.

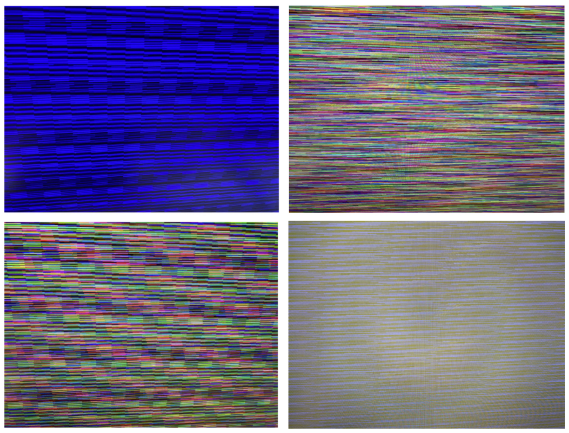

Fig. 7. **HMDI output of sine wave sample at various different sample widths.**

VI. Evaluation

We first evaluate this project by revisiting the goals from our initial block diagram presentation:

- The Commitment: For this low-score MVP we aimed to have a MIDI decoder and oscillator outputting I2S data with a frequency of 44.1 kHz. All of these goals were met with our V0.
- The Goal. This goal focused on having a way to mix the audio samples from our oscillators. In our V0, this was implemented with our oscillator coordinator, which is also responsible for coordinating with incoming MIDI events, and the volume stabilizer modules. This goal was also met in our V0.
- The stretch goal. At the time, our idea for a stretch goal was ”Real-time wave modification.” We were able to achieve this also in a relatively simple way by simply adjusting the number of samples we would oscillate in a given waveform.

Outside of these goals, the primary challenge that we came across in our system came down to memory; that is, both the constraints of memory on the FPGA and the way we manage it. Our vision for this synthesizer was to be able to take in any kind of data: MP3 files, PDFs, a picture of a cat, etc, and be able to output whatever that would sound like.

Without SD, we attempted to cut down on our BRAM usage in the V0 design. We initially had BRAMs for each oscillator, a BRAM for our HDMI output, and a BRAM for the UART data we were sending over to a laptop. Our solution to this was to integrate all oscillators into a single shared BRAM, and although that change certainly helped with BRAM usage, we were still constantly running into memory issues. Though we were only using around 30 percent of the available memory on the FPGA, we realized that the issue likely came down to the way we structured our data. Our sound samples were 16 bits wide but 512 levels deep, which we believe created very narrow strips of memory in our BRAMs, and could have been improved by efficiently redesigning our memory storage to store multiple 16-bit samples in a single line.

VII. Retrospective

Looking back on this project, we have a number of key takeaways:

- Take advantage of office hours and staff when you get stuck. Prior to our final-stretch meeting with Joe, we spent much of the time in V0 debugging our MIDI modules, which prevented us from making progress on our V1. We looked at the output of our MIDI breakout board, and assumed that it was fine because it was outputting what looked to be a proper UART message, so we looked over our MIDI modules over and over to try and find a bug. During our meeting however, Joe quickly pointed out that the voltage difference in these messages should be on the order of volts, not the millivolts we were seeing. Half an hour after our meeting, we realized we had forgotten to hook up the 5V pin in our breakout board, which had been the source of our issues.
- Use logic analyzers! At least for most things. A lot of our initial debugging was done with an oscilloscope, as we tested the various outputted frequencies that our synthesizer sent via I2S. Once our aforementioned MIDI problem was fixed, we stayed using the oscilloscope with triggers to analyze all the digital data that was coming out of the FPGA. Oscilloscope triggers are a wonderful thing but it also restricted us to analyzing four signals at a time and it meant we had to deal with a rat’s nest of wires and probes. Needless to say, we wish we had used PulseView with the Logic Analyzer earlier on in our project.
- Start early or prioritize early. Though we were able to implement and integrate all the features we laid out for our V0, we weren’t able to get to the full integration stage for all the features we wanted in V1, most notably with the SD card feature. In addition to being tricky to implement in isolation, it was even tricker to integrate these into our existing system, as bugs inevitably popped up that we had not anticipated for in our prior testing. As the deadline approached, we did not prioritize having one

feature over another, which led to a number of features that were close to working with our system, but not fully integrated.

VIII. Acknowledgments

We'd like to thank Joe for his guidance throughout this entire project, and the entire teaching staff for getting us through the debugging the final version of our synthesizer.