Oud Synthesizer Final Report

Adan Abu Naaj
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
adann@mit.edu

Ahmad Durra
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
durra@mit.edu

***Abstract*—We present a design for an FPGA-powered music synthesizer designed to replicate the sounds of the Arabic classical instrument, the Oud. The synthesizer features 24 capacitive sensors representing 3 music scales, with each scale containing 8 distinct notes. These sensors simulate the pressing of strings, allowing musicians to play the instrument. When a sensor is tapped, the FPGA detects the input and generates a corresponding note. This note is then processed through a PDM module and delivered to the speaker output, producing an Oud melody. As an MVP, our synthesizer will process individual notes, and then we will utilize FPGA parallel processing to handle multiple notes played at the same time.**

Fig. 1. Oud Instrument.

I. Physical Design (Adan)

The physical design of the synthesizer consists of a laser-cut, staircase-shaped enclosure featuring three distinct levels. Each level has a row of eight capacitive sensors, each of which corresponds to a unique waveform. The enclosure is engraved with Arabic calligraphy and olive tree branches, reflecting the cultural heritage of the Oud instrument.

We designed the capacitive sensors using computer-aided design (CAD) software and then milled the capacitive pads onto a copper PCB.

This multi-level design is inspired from multi-tiered pianos, highlighting the ability to play different waveforms in one instrument.

II. System Overview (Adan)

The synthesizer system comprises several interconnected components and modules, each responsible for a specific function in the audio synthesis process. The main features consists of:

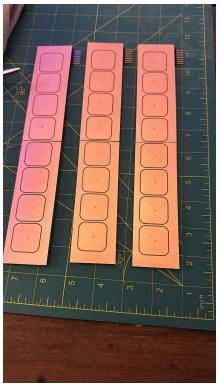

Fig. 2. Capacitor Pads.

Fig. 3. Laser Cut Enclosure.

- Capacitive Sensing Interface.
- Sine Waveform Synthesizer.
- Oud Audio Playback.
- Polyphonic Synthesizer.
- ADSR Envelope.
- Audio Output Module.

III. Design and Implementation

A. Capacitive Sensing Interface (Durra)

The capacitive sensing interface forms the user input mechanism for the Oud Synthesizer. Rather than the traditional keyboard interface, the Oud synth utilizes an array of capacitive

pads. It detects touch events on the capacitive pads and triggers corresponding notes to be played.

1) Physical Interface Design: The physical interface consists of an array of capacitive pads milled onto a copper PCB. Each pad represents a specific note on the Oud; A user touching the pad triggers that note to be played. To continuously monitor pad touch events, an MPR121 capacitive touch sensor IC is used. The MPR121 is capable of handling 12 touch inputs and we are using 3 mpr121 to handle 24 notes. The remainder of the Capacitive Sensing Interface handles communication with the MPR121.

2) I²C Controller Module: The MPR121 uses an I²C communication interface. To facilitate data exchange with this interface, an I²C controller module is implemented on the FPGA. The controller manages the low-level I²C protocol. The format of a read and write transaction to the MPR121 is shown in Figures 4 and 5, respectively.

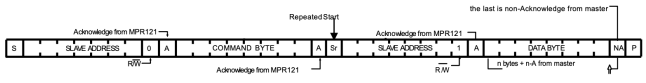

Fig. 4. I²C read transaction format for MPR121.

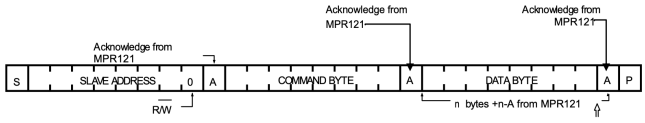

Fig. 5. I²C write transaction format for MPR121.

The module's key inputs and outputs are defined as follows:

- **Inputs:**
  - `command_byte_in`: 8-bit input specifying the target register address.
  - `data_byte_in`: 8-bit input specifying the data to be written to the target register.
  - `rw`: A control signal indicating the operation type (0 for write, 1 for read).
  - `start`: A signal to initiate an I²C transaction.
- **Outputs:**
  - `scl_out`: Clock signal for the I²C bus.
  - `data_byte_out`: 8-bit data received during a read transaction.
  - `ack_out`: A signal indicating acknowledgment from the MPR121.
- **Bi-directional:**
  - `sda`: Bi-directional data line for the I²C bus.

The I²C controller operates using a finite state machine (FSM) to manage the sequence of operations required for I²C communication. The FSM ensures that all timing constraints are met and that the MPR121's setup and hold times are respected. The key states of the FSM are as follows:

- **IDLE:** The FSM remains in this state until the `start` signal is asserted, indicating the initiation of a transaction.
- **START:** Generates the start condition by pulling `SDA` low while `SCL` is high. Once established, the FSM transitions to the **ADDR** state.
- **ADDR (Address Byte):** Sends the 7-bit MPR121 address (`peripheral_addr_in`) followed by the `rw` bit. The rw bit is only 1 after a repeated start for a read. The FSM then waits for an acknowledgment (ACK) from the MPR121.
  If an acknowledgment is received, the FSM transitions to either DATA, READ, START, or STOP state depending on what the last transmitted byte was. A transition to a START state occurs if it is a read transaction and the last transmitted byte is a command byte indicating that the next step is a repeated start. Transitions to other states follow the format for an MPR121 read/write shown above.
- **ACK:** Ensures `SDA` is in a high-impedance state to read the acknowledgment signal from the MPR121. If an acknowledgment is received, the FSM transitions to either **DATA**, **READ**, **START**, or **STOP** state depending on what the last transmitted byte was. A transition to a START state occurs if it is a read transaction and the last transmitted byte is a command byte indicating that the next step is a repeated start. Transitions to other states follow the format for an MPR121 read/write shown above.
- **DATA (Data Transmission):** Sends the command or data byte, depending on the context. If the last transmitted byte was the address, the FSM transmits the command byte. If the command byte was last sent, it transmits the data byte. The FSM then transitions to **ACK**.
- **READ (Data Reception):** For read operations, the FSM transitions `SDA` to a high-impedance state to allow the MPR121 to transmit data. The FSM reads this data byte and then transitions to **ACK** followed by **STOP** state.
- **STOP:** Generates the stop condition by pulling `SDA` high while `SCL` is high. This indicates the end of the transaction. The FSM then transitions back to **IDLE**.

3) MPR121 Controller Module: The MPR121 controller module handles initialization, configuration and continuous reading of touch. It implemented as a FSM that sends a sequence of transactions to the MPR121 to disable it before starting configuration, set touch and release thresholds for each electrode, enable it, followed by continuously reading its touch status registers. The touch status is outputted and taken in by the node decoder module.

B. Sine Waveform Synthesizer (Durra and Adan)

*1) **Note Decoder Module**:* The Note Decoder Module translates the touch status input from the capacitive interface to determine which note the user is playing. It then uses a lookup table (LUT) to output the selected note to the `phase_accumulator` module.

*2) **Generate Sine Waveform** :* To synthesize sine waveforms of the pitches being played, we store a 256 sample sine wave in a BRAM. Given the note to be played, we traverse

through the 256 samples at the frequency of the note using the Phase Accumulator Module

3) ***Phase Accumulator Module****:* Given note_out from the Note Decoder. The phase accumulator calculates a phase increment value corresponding to the desired note frequency. Given a phase increment, every clock cycle, it accumulates a 32-bit phase value steadily by the phase increment, and wraps around at the desired frequency.

To calculate the phase increment, the formula $\frac{\text{Clock Frequency}}{\text{Frequency}}$ gives us the number of clock cycle that one period of that note spans. We would like the phase value to sum up to 2^{32} in that number of cycles which means that the phase increment is 2^{32} divided by the number of clock cycles in a single period giving us the equation below for the phase increment

$$\text{Phase Increment} = \left(\frac{\text{Frequency} \times 2^{32}}{\text{Clock Frequency}}\right)$$

C. Oud Waveform Implementation (Adan)

To accurately replicate the Oud notes, we utilized the Python library librosa for audio analysis and processing. We load the authentic analog Oud recording and resample the audio to 16 ksps sampling rate where each audio sample is quantized to 16-bit integer format preserving the resolution and quality of the Oud note.

Additionally, the processed audio samples are encoded into a hexadecimal format to create a hex file for each distinct Oud note that is compatible to be used by the FPGA. These files are generated in advance and are later stored in BRAMs on the FPGA for audio playback.

To verify the accuracy of the hex files, another Python script using librosa and numpy converts the hex files back into .wav audio format. This process allows us to compare between the original and the resampled audio generated. The testing process includes playing both .wav files to make sure we capture the authentic sound of the note, and visualizing the note by plotting the note before and after the resampling.

As shown in the plots below, the regenerated waveform closely matches the original, demonstrating the accuracy of our hex file generation process. Minor discrepancies, if present, are due to the resampling and quantization processes but remain within acceptable limits for playback quality.

D. Single Note Playback (Adan)

When a capacitive sensor is triggered, the system identifies the corresponding note and retrieves its audio data from memory for playback. The playback process involves multiple modules including the `note_decoder`, BRAM storage, `sample_address_counter`, and `sample_rate_counter`.

a) Note Decoder: The `note_decoder` translates input signals from the capacitive sensors into corresponding memory addresses for the note data. Each capacitive sensor is mapped to a unique note, and the `note_decoder` uses a lookup table to output a corresponding address or signal that directs the system to the correct hex file in BRAM.

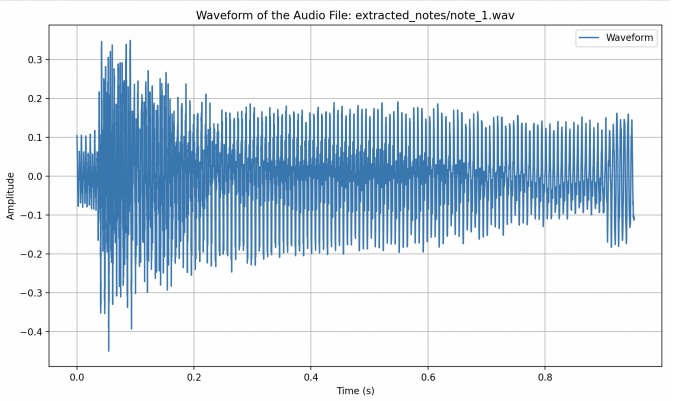

Fig. 6. Original Oud Note Waveform.

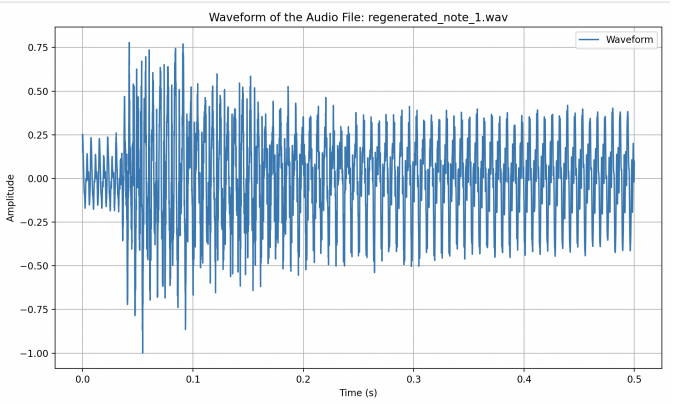

Fig. 7. Regenerated Waveform (From Hex File).

b) BRAM Storage: The hex files generated for each Oud note are stored in Block RAM (BRAM) on the FPGA. Each note is stored as 8192 samples, where each sample is a sequence of 16 bits.

$$\text{Bits per Note} = 8192\,\text{samples} \times 16\,\text{bits/sample} = 131,072\,\text{bits}.$$

Using the Xilinx Spartan-7 XC7S50 FPGA, which has 120 BRAM blocks each with 36,000 bits of storage, the number of BRAM blocks required per note is:

$$\frac{131,072\,\text{bits}}{36,000\,\text{bits/BRAM}} \approx 3.64\,\text{BRAM blocks}.$$

This rounds up to approximately 4 BRAM blocks per note. Therefore, the FPGA can efficiently store up to 24 notes (two full octaves) in BRAM.

Evaluation: By using this technique, we trade off memory to achieve more accurate and authentic Oud sounds. However, to scale the project and generate additional notes, we continued with the sine wave synthesis approach and added different waveforms and ADSR module to get more accurate instrument sounds.

c) Address and Sample Counters: This system consists of two modules: `sample_address_counter` and `sample_rate_counter`. The `sample_address_counter` determines the current memory address being read from BRAM during playback by sequentially incrementing through memory addresses containing the audio sample.

The `sample_rate_counter` divides the FPGA's system clock to generate the required sampling clock for audio playback, ensuring that audio samples are read at the correct playback rate of 16 kSps.

E. Multiple Waveforms

We generated sine, sawtooth, and square waveforms using Python's NumPy library, a powerful tool for numerical computation. These waveforms were stored in hexadecimal format and loaded into the FPGA's BRAM for synthesis. Each waveform was sampled 256 times per cycle, with a 16-bit unsigned data width, ensuring sufficient resolution for audio playback.

The formulas used to compute the values for each waveform are as follows:

Sine Wave

$$\text{sine\_wave}[n] = \text{amplitude} \cdot \sin\left(\frac{2\pi n}{\text{num\_samples}}\right) + \text{offset}$$

Sawtooth Wave

$$\text{sawtooth}[n] = 2 \cdot \text{amplitude} \cdot \left(\frac{n}{\text{num\_samples}}\right) - \text{amplitude} + \text{offset}$$

Square Wave

$$\text{square\_wave}[n] = \text{amplitude} \cdot \text{sign}\left(\sin\left(\frac{2\pi n}{\text{num\_samples}}\right)\right) + \text{offset}$$

These three waveforms produce distinct sounds for the synthesizer:

- **Sine Wave:** The sine wave generates a smooth and continuous oscillation, resulting in a clean and pure tone.
- **Sawtooth Wave:** The sawtooth wave has a linearly rising slope followed by a sharp drop, creating a bright and harsh sound.
- **Square Wave:** The square wave alternates between high and low levels, producing a hollow and buzzy tone.

F. ADSR Envelope (Adan)

The ADSR (Attack-Decay-Sustain-Release) module models the amplitude envelope of a sound signal, replicating the dynamic and expressive qualities of real instrument sounds. The design logic is based on the observation that when a real instrument generates a note, the volume of the musical note changes over time—rising rapidly from zero and then steadily decaying. To produce this effect, we multiply the note waveform by an ADSR envelope that contains attack, decay, sustain, and release phases. The graph below illustrates the amplitude envelope.

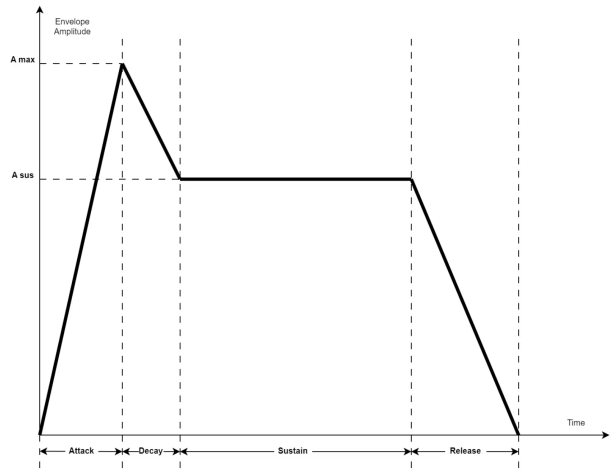

Fig. 8. ADSR Envelope Amplitude Values

*1) **Inputs & Parameters**:* The ADSR is a sequential module with two primary inputs:

- **Start:** A trigger to start the envelope.
- **Hold:** Keeps the amplitude at the sustain level. Typically, the note decays to silence after the sustain phase, but we made a design choice to stay at a low sustain level.

Additionally, the ADSR takes two amplitude parameters, A_{max} and A_{sus}, as well as four time parameters: *attack_time*, *decay_time*, *sustain_time*, and *release_time*.

- $A_{\textbf{max}}$**:** The maximum amplitude the envelope can reach.
- $A_{\textbf{sus}}$**:** The amplitude of the sustain segment.
- **Attack time:** The duration of the attack segment.
- **Decay time:** The duration of the decay segment.
- **Sustain time:** The duration of the sustain segment.
- **Release time:** The duration of the release segment.

The module uses the above parameters to calculate these local parameters:

$$\text{attack\_step} = \frac{A_{\text{max}} - 0}{\text{attack\_cycles}}$$

$$\text{decay\_step} = \frac{A_{\text{max}} - A_{\text{sus}}}{\text{decay\_cycles}}$$

$$\text{release\_step} = \frac{A_{\text{sus}} - 0}{\text{release\_cycles}}$$

*2) **FSMD Design** :* The ADSR module uses a Finite State Machine with Data Path (FSMD), combining the FSM control logic with a datapath for managing the amplitude level, which is tracked in the *amplitude_counter*.

The FSM transitions through the following states:

- **Idle:** Default state; this state waits for a *start* signal to transition to the *Launch* state and initiate the envelope generation process.
- **Launch:** Initializes *amplitude_counter* to 0 and transitions to the *Attack* phase.

- **Attack:** Increments *amplitude_counter* by *attack_step* in each clock cycle until reaching the maximum amplitude level A_{max}. When *amplitude_counter* equals A_{max}, the system transitions to the *Decay* state.
- **Decay:** Decrements *amplitude_counter* by *decay_step* every clock cycle until reaching the sustain level A_{sus}. When *amplitude_counter* equals A_{sus}, the system transitions to the *Sustain* state.
- **Sustain:** Maintains the amplitude at the sustain level A_{sus} for a specified duration (*sustain_time*) or as long as *hold* is active. *sustain_time_counter* is incremented in each clock cycle, and the system transitions to the *Release* state when *hold* is low and *sustain_time_counter* equals *sustain_time*.
- **Release:** Gradually decreases the amplitude to zero after the sustain phase by decrementing *amplitude_counter* in each clock cycle until reaching zero, then transitions to the *Idle* state.

At any point, if *start* is triggered again (indicating that the user played the same note or a different note), the envelope resets, and the state transitions to the *Launch* state. The picture below illustrates the ADSR FSM.

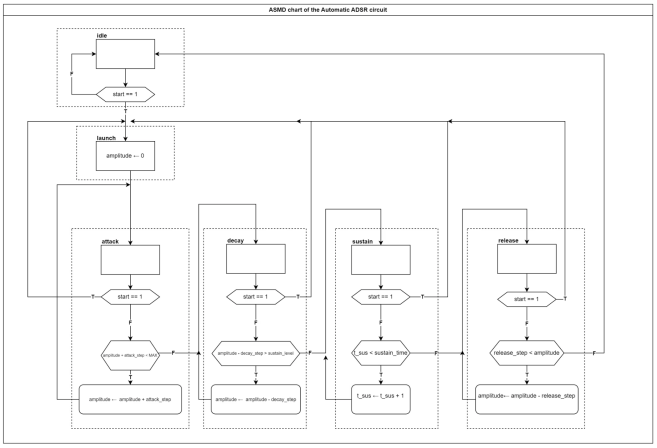

Fig. 9. ADSR FSM Transitions

*3) **Output** :* The ADSR module outputs a 16-bit envelope signal representing the amplitude at any given time. This signal modulates the note waveform generated by the synthesizer. To enable polyphony, each note has its own instance of the ADSR module, allowing multiple notes to play simultaneously.

G. Audio Output Module (Adan)

The sound output system is designed to convert digital audio samples into analog signals that can drive a speaker or headphones through a 3.5mm TRS (Tip-Ring-Sleeve) audio cable. To achieve that, we use Pulse-density modulation (PDM) module which takes in the 16-bit digital samples and converts them into PDM signals. The duty cycle of the PDM signal is proportional to the amplitude of the input audio sample and the PDM carrier frequency is set significantly higher than the audio signal frequency to ensure smooth analog output after filtering.

H. Polyphonic (Adan)

Polyphony, the ability to play multiple notes simultaneously, is a key feature of our project. Our system can handle up to 8 notes in parallel, each processed independently and then combined to produce a harmonious output.

***ADSR Modules**:* First, each note is assigned its own ADSR (Attack-Decay-Sustain-Release) instance to dynamically shape its amplitude over time. Each ADSR module outputs an envelope (`adsr_envelope[i]`) for the respective note, enabling simultaneous and independent amplitude control for up to 24 notes.

***Phase Accumulators & Address Generator**:* As explained above, the phase accumulators generate phase values corresponding to the frequency of each note. The phase values are then processed by the address generator, which translates them into memory addresses to retrieve waveform samples stored in BRAM.

In addition, the `address_generator` keeps track of how many and which notes are played by updating and outputting `active_voices` and `num_voices`:

- **`active_voices`**: Identifies which notes are currently being played. It is an 8-bit value where each bit represents the state of a corresponding note. If a note is being played, its bit is set to 1; otherwise, it is 0. For example, `active_voices = 8'b00001010` indicates that the 2nd and 4th notes are currently active.
- **`num_voices`**: A counter that keeps track of how many notes are currently being played.

***Waveform Storage in BRAM**:* Waveforms for each note are preloaded into Block RAM (BRAM) modules. Each BRAM is configured with dual-port access, allowing simultaneous reads for up to 2 notes per module. The address generator provides memory addresses to retrieve waveform samples corresponding to the current phase of each note.

***Note Summation and Averaging**:* The waveform samples retrieved from BRAM are multiplied by their respective ADSR envelopes if the corresponding note is active. The waveform for note i is output as `sine_spk_data_out[i]`. If `active_voices[i]` is high, the sample is multiplied by the ADSR envelope, resulting in:

$$\text{voice\_values[i]} = (\text{sine\_spk\_data\_out[i]} \times \text{adsr\_envelope[i]}) \gg 16$$

This involves a 16-bit by 16-bit multiplication, and the result is shifted right by 16 to fit within 16 bits.

The values from all active notes in `voice_values` are then summed. The total is normalized by dividing it by the number of active voices (`num_voices`), as determined by the address generator.

***Division Optimization**:* To efficiently handle division by numbers that are not a power of 2, we applied a method based on the logic that dividing a number by 2 can be achieved by first multiplying it by 16 and then dividing it by 8. Extending this logic, as long as the divisor is a power of 2, we can

perform efficient division through multiplication and a single shift operation. This approach avoids direct division, which is computationally more expensive.

The implementation uses a case statement to determine the appropriate scaling factor based on the value of `num_voices`. The scaling factor is calculated as:

$$\left\lfloor \frac{256}{\texttt{num\_voices}} \right\rfloor$$

The result is divided by 256 by performing a right shift of 8 bits after applying the scaling factor. For example, to divide by 3, the sum is first multiplied by:

$$\left\lfloor \frac{256}{3} \right\rfloor = 85$$

and the result is then shifted right by 8, effectively dividing it by 256.

This logic ensures that the final combined and normalized waveform is prepared for playback through the Pulse Density Modulation (PDM) module.

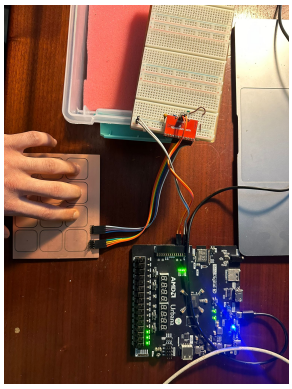

Fig. 10. Capacitive Sensors and Synthesizer.

IV. Evaluation

Latency and Throughput

We analyzed the Vivado Post-Route Timing Summary and Post-Place Utilities to evaluate the performance and efficiency of our design.

Our design meets all timing requirements. The Worst Negative Slack (WNS) is 2.168 ns, and the Total Negative Slack (TNS) is 0 ns, indicating no timing violations. The longest path in the design extends from `address_generator_inst/active_voices_reg[22]/C` to `multiplied_sum_reg[10]/D`, with a total delay of 7.677 ns.

This longest path is primarily impacted by multiplying all note waveforms with ADSR envelopes, introducing computational overhead due to the 16-bit by 16-bit multiplications. However, these computational steps ensure each note has a dynamically shaped amplitude, significantly enhancing the musical quality and realism of the synthesized sound.

Resource Utilization

The design makes efficient use of FPGA resources, as indicated by the post-place utilization report. The utilization figures include:

- **Slice LUTs:** 3231/32600 (9.91 %)
- **Slice Registers:** 2213/65200 (3.39 %)
- **DSP Blocks:** 21/120 (17.50 %)
- **Block RAM Tiles:** 0/75 (0.00 %)

The DSP slices are utilized at 17.50 %, efficiently handling critical arithmetic operations such as the multiplications required for ADSR envelope shaping and waveform summation.

V. Goals Achieved and Scalibility (Adan)

We successfully implemented our commitment that consists of a synthesizer with switches that play individual notes using a sine wave and a PWM module.

Later, we achieved our goal of creating an Oud synthesizer using playback audio. However, this approach consumed a significant amount of memory, so we replaced it with an ADSR module and generated multiple waveforms to produce more dynamic sounds and better replicate real instruments. Additionally, we upgraded the PWM module to a PDM module.

Finally, we achieved part of our stretched goal and scaled our project to handle 24 capacitors instead of the initial 8.

To further scale the project, a straightforward change could be to control the ADSR parameters using potentiometers instead of module inputs. This will give users the flexibility to modify the sounds, enabling a single synthesizer to emulate multiple instruments.

Another simple addition could be to change the phase increments to support more octaves.

Furthermore, we aim to explore alternative capacitor designs, such as arranging them in a row or a circular layout, allowing the synthesizer to play different amplitudes by detecting specific locations on the capacitors.

Fig. 11. Final Synthesizer

VI. Block Diagram

Refer to Figure 7 and 8 attached at the end of the report.

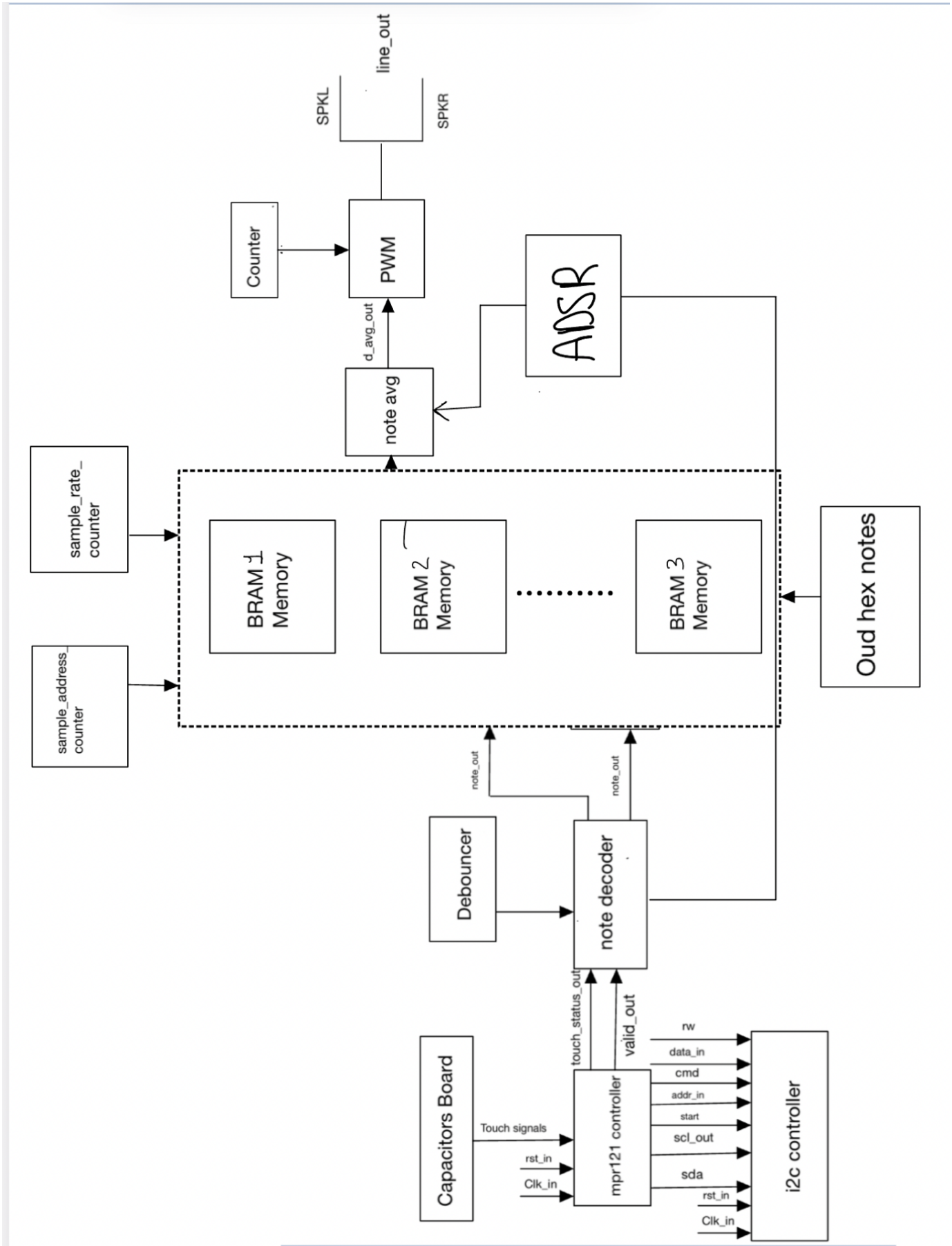

Fig. 12. Block Diagram for Sine Synthesizer using Phase Increment Approach

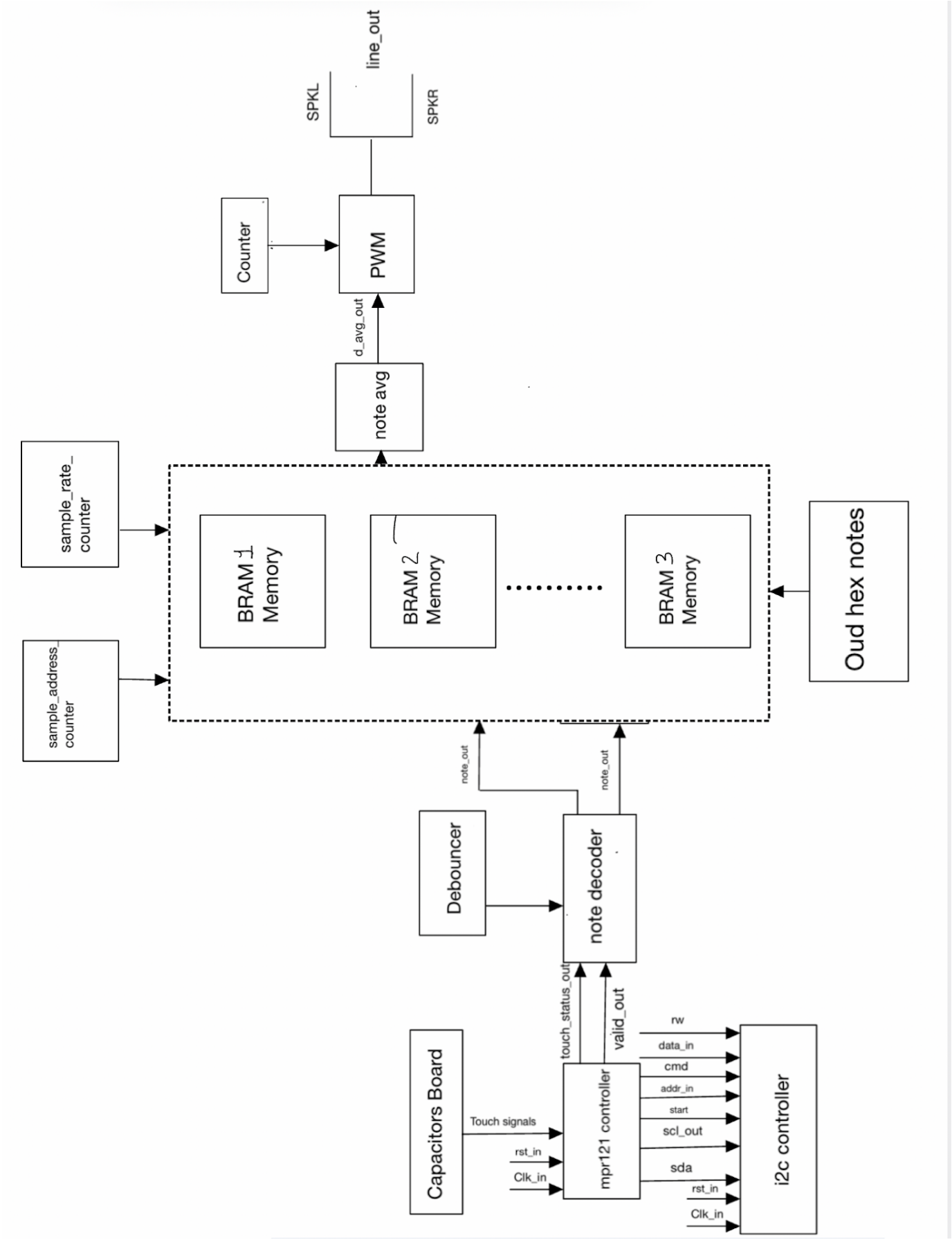

Fig. 13. Block Diagram for Oud Audio Playback Synthesizer using Multiple BRAMs

ACKNOWLEDGMENT

We would like to thank Professor Joe Steinmeyer, Kiran Vuksanaj, Stephen Kandeh, Kailas Kahler, and all other Teacher Assistants and Lab Assistants for their help and guidance throughout this semester. We would also like to thank maker space mentors who helped us with the physcial design of the project.

REFERENCES

[1] Librosa Documentation, *Online*. Available: https://librosa.org.

[2] J. Valdez and J. Becker, "Texas Instruments: Understanding the I2C Bus," SLVA704, June 2015. *Online*. Available: https://www.ti.com/lit/an/slva704/slva704.pdf?ts=1732171821168.

[3] SparkFun Electronics, "MPR121 Capacitive Touch Sensor," *Online*. Available: https://www.sparkfun.com/datasheets/Components/MPR121.pdf.

[4] Element14 Community, "System Verilog ADSR," *Online*. Available: https://community.element14.com/technologies/fpga-group/b/blog/posts/systemverilog-study-notes-adsr-envelope-generator-for-sound-synthesis.