

FPGA Spinthesizer Final Report

1st Than Thanpanit

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
thanpa28@mit.edu

2nd Ben Lammers

Department of Aeronautics and Astronautics
Massachusetts Institute of Technology
Cambridge, MA, USA
blammers@mit.edu

Abstract—We present a hardware-based design for the FPGA Spinthesizer, a real-time musical system that converts MIDI input into audible tones generated by stepper motors. The system is implemented entirely on an FPGA fabric and integrates a modular MIDI processing pipeline with a scalable array of stepper-motor output generators. Incoming MIDI packets are decoded, routed, and optionally transformed through a pseudo-random improvisation engine that enables call-and-response musical interaction. These processed notes are mapped to motor-driving waveforms using a precomputed note-to-period conversion module and a dedicated hardware signal generator, allowing stepper motors to produce musically accurate pitches. Our implementation harnesses the RealDigital Urbana Board, an FPGA-based prototyping platform that uses the AMD Spartan-7 XC7S50-CSGA324 FPGA.

Index Terms—Digital systems, Field-programmable gate arrays, MIDI, Stepper motors, Embedded music systems

I. INTRODUCTION (BEN)

Stepper motors are designed to actuate precise mechanical movements in 3D printers, CNC machines, and robotic systems. As a byproduct of precise mechanical movements, they often create pitches that sound like music. These pitches are not intentional, and thus not in tune, and there are currently very few frameworks to control stepper motors for artistic purposes. Thus, there was a unique opportunity for specialized hardware: we designed an FPGA Spinthesizer that can be used to convert user input to stepper motor controls, allowing stepper motors to play familiar melodies with harmony.

We present an FPGA-based interface that enables a MIDI input to be translated into stepper-motor music. The system plays up to 6 notes at a time at up to 4 different volumes based on live input from a MIDI keyboard. In addition, the system demonstrates pseudo-random improvisation, harmonizing with the user as they provide their input, and responding user’s melodic improvisation in a call-and-response fashion.

II. HIGH-LEVEL EXPLANATION OF THE DESIGN (THAN)

At the high level, the `midi_processor` module of our Spinthesizer takes in MIDI data, optionally adds its own improvisation to the data, and parses everything into the frequencies and volumes of the notes to be synthesized. The parsed data is then used by the `stepper_out_gen` module to generate control signals for the stepper drivers. The enable signals are serialized and passed to an external 24-bit shift register as the PMOD header does not contain enough pins to directly output all the required signals.

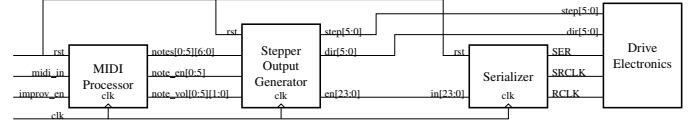


Fig. 1: High-level system architecture diagram showing the how the MIDI Processor block integrates with the four Stepper Output Generator blocks to play multiple notes at the same time.

The Spinthesizer itself consists of the following components:

- 25 Key MIDI Keyboard
- MIDI to UART Adapter Board
- 24 NEMA 17 Stepper Motors with Driver Electronics
- RealDigital Urbana FPGA Board
- Custom PCB carrying all stepper driver electronics
- 24 3D-printed “flags” attached to motor shafts to show rotation
- A laser-cut acrylic board, on which the motors are mounted, and an accompanying 3D-printed stand

III. MIDI PROCESSING (BEN)

A. UART Receiver

The UART Receiver takes in the 100 MHz clock, as well as a `midi_in` signal. This signal passes through a commercial off-the-shelf adapter according to the official MIDI electrical specification [1], which converts the signal into a UART waveform compatible with our FPGA. We have implemented a UART receiver based on the design from Lab 3, and configured it for the MIDI baud rate of 31250. The receiver converts the asynchronous serial stream into 8-bit data packets, which are forwarded to the MIDI Processor to keep track of which notes are on and off.

B. MIDI State Machine

The MIDI Processor parses each incoming 8-bit packet. If the incoming byte is a status byte, it records the MIDI channel of the byte, and listens to the next two packets to determine the note and velocity. It maintains an internal state that tracks active notes, storing the pitch of each active note in registers. When a note is signaled as active, the note number and a note enable signal are passed to a Stepper Output Generator.

The MIDI Processor Core is responsible for tracking which notes are active and assigning them to different Stepper Output Generators.

This module has been implemented and testbenched, and is now being adapted to pass its note and note enable signals to the Pseudo-random Improvisation module, which will toggle on/off real-time improvisation. This module will be toggled on/off by a switch on the FPGA board, which is passed to the `improv_en` input.

C. Pseudo-random Improvisation

The pseudo-random improvisation module generates responses to user input that are derived from the current harmonic and rhythmic context. We have now clarified both the role and scope of this module and developed an initial implementation and testbench. Our design uses a Linear Feedback Shift Register (LFSR) to provide pseudo-random pitch variation, while repeating the same rhythmic patterns played by the user.

The system is loaded with a 24-second (12 measures at 120 bpm) chord progression, containing chords sampled at 24 samples per second, with 3-5 notes per sample. This progression is preloaded into the BRAM of the device using 8 bits per note, requiring at most 23,040 bits. The system continuously plays and loops these stored notes.

To support call-and-response interaction, the system alternates between improvising and listening to the user. During the listening phase, the system records single-note user input at a rate of 192 samples per measure (8 bits per sample) for 12 measures, storing up to 18,432 bits in a dedicated BRAM module.

After the listening phase concludes, the system replays the same chord progression for the improvisation phase. During this phase, the cached user notes are played back in time-synchronized fashion. While replaying the cached rhythms, the module adds LFSR-generated offsets to modify the pitch of each note while preserving the user's rhythmic pattern. Each pitch is then quantized to the nearest note in a provided scale to ensure harmonic consistency with the underlying chords.

At present, the chord and scale logic supports only a 12-measure C blues form, using the C blues scale for improvisation. We are currently extending the system to allow users to preload arbitrary scales and chord progressions. In addition, we are exploring improved mechanisms for signaling the transitions between listening and improvisation phases, providing clearer feedback to the performer.

IV. PLAYING NOTES ON STEPPER MOTORS (THAN)

A. Conversion of Note Number to Period

From the MIDI specifications [1], the note middle C is assigned the note number of 60, and incrementing the note number by one raises the note by a half tone. Therefore, the note A4, which is tuned to 440 Hz in twelve-tone equal temperament (12-TET), corresponds to note number 69. Hence, the frequency of the note with the note number n is

$$f_n = (440\text{Hz}) \times 2^{\frac{n-69}{12}}.$$

Therefore, the period of the note with the note number n is

$$T_n = \frac{1}{f_n} = \frac{2^{\frac{69-n}{12}}}{440} \text{s}$$

Using the 100 MHz clock, one period corresponds to N_n clock cycles, where

$$N_n = T_n \times (100\text{MHz}) = (12\,231\,220.59) 2^{-\frac{n}{12}}$$

rounded to the nearest integer.

Because n ranges from 0 to 127, inclusive, the maximum value of N_n is $N_0 = 12\,231\,221$. Therefore, $\lceil \log_2 N_0 \rceil = 24$ bits are needed for each note's cycle count. To store the precomputed values of all 128 notes, $24 \times 128 = 3072$ bits of distributed RAM is needed. This is only 0.5% of the maximum 600 Kb possible on our FPGA [2]. Thanks to the distributed RAM's asynchronous read capability, this conversion does not require any additional clock cycles.

In practice, this module is seen by Vivado as regular logic and synthesized as lookup tables.

The SystemVerilog module `nn_to_period` implements this functionality.

B. Driving stepper motors

To produce a note on a stepper motor, we need to step the motor at the frequency equal to f_n . Since we already have the number of clock cycles per period N_n , we simply need to step the motor every N_n cycles.

According to the A4988 datasheet [3], stepping the motor is done by pulling the STEP pin of the A4988 driver high for at least 1 μs and then low for at least 1 μs .

To achieve a visual effect, the direction of rotation reverses every time the note changes. This is done by pulling the DIR pin of the A4988 driver high or low, depending on the desired direction, 200 ns before the rising edge of the STEP signal and hold it until 200 ns after the rising edge of the STEP signal. In other words, the setup time and hold time are both 200 ns [3].

For simplicity, we operate the stepper motors in the full-step mode. This is realized by tying the MS1, MS2, and MS3 of the driver module to ground [3].

The SystemVerilog module `stepper_drv_sig_gen` implements these functionalities.

C. Playing Multiple Notes with Multiple Volumes

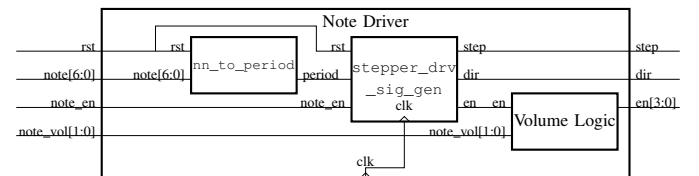


Fig. 2: Block diagram for the Note Driver module (`note_driver.sv`)

To achieve multiple volumes, each note is assigned 4 stepper motors for 5 different volumes: not playing, 0, 1, 2,

and 3, corresponding to 0, 1, 2, 3, and 4 enabled motors, respectively. The SystemVerilog module `note_driver` (Fig. 2) implements this functionality.

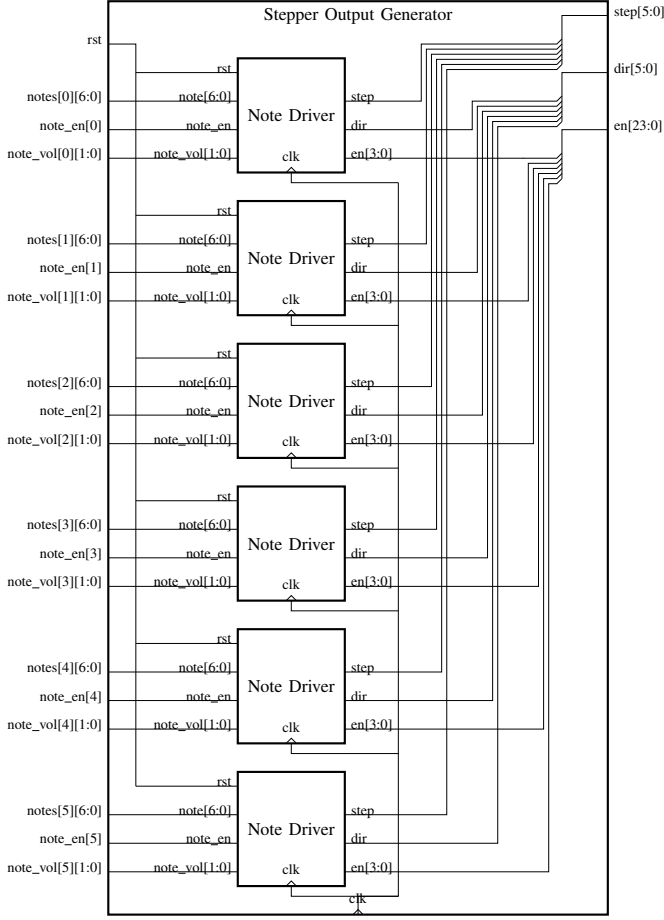


Fig. 3: Block diagram for the Stepper Output Generator module (`stepper_out_gen.sv`)

To play multiple notes at once, we instantiate one `note_driver` per note and route the output to the driver electronics. This is implemented in the `stepper_out_gen` SystemVerilog module (Fig. 3).

D. Serializing Enable Signals

Outputting the signals directly requires 12 PMOD pins for `step` and `dir` signals and 24 pins for the `en` signals. Our FPGA board does not have enough pins. To mitigate this, we serialize the `en` signals, feed the serialized data to 3 daisy-chained SN74HCS59 shift registers which act as a single 24-bit shift register, and connect the shift registers' outputs to the enable pins of the A4988 drivers. We only use shift registers for the `en` signals because they are not timing sensitive, and this is enough to overcome the pin number constraint.

The SystemVerilog module `serializer` implements this functionality according to the state machine in Fig. 4. In the IDLE state, the module saves the input data for future reference and pulls both RCLK and SRCLK low. In the SHIFT_OUT state, the module set the SER output (the input

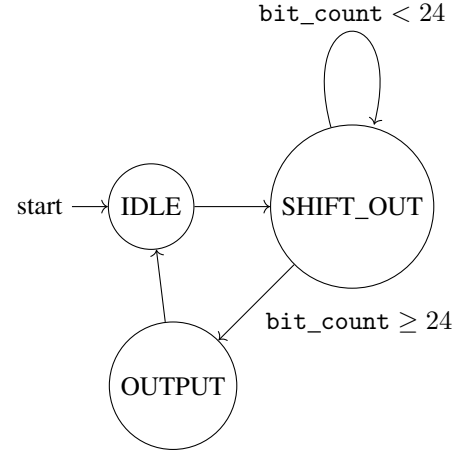


Fig. 4: Serializer finite state machine

to the shift register) according to the saved data and pulses SRCLK, the shift register's clock, while keeping track of the number of transmitted bits. In the OUTPUT state, it pulses RCLK to make the shift register's output flip-flops save the data.

V. OVERALL SYSTEM EVALUATION

A. Throughput, Latency, and Timing Requirements

Because the focus of our system is on user interaction, it suffices to keep the latency low and throughput high enough for the user to not experience any lags. Based on our tests, the system does not have any lag that is appreciable by humans.

We were able to comfortably satisfy the timing requirement with the worst negative slack (WNS) of +2.721 ns. We did not encounter any issue related to timing and did not need additional pipelining.

B. Memory

Three different components of our system require memory. The first is our chord memory, for pseudo-random improvisation. This consists of 24 seconds of chords (each 3-5 notes that use 40 bits of memory per sample). Sampled at 24 times a second, this produces 576 BRAM entries, each at a width of 40 bits, using 23040 bits in total, fitting inside a single BRAM module on our FPGA. The second component that requires memory is our user input memory, also for pseudo-random improvisation. This consists of 24 seconds of user input notes. Since the system samples at 96 times per second, and each sample takes 8 bits (only storing a single note), this takes 18,432 bits of BRAM, also fitting inside a single BRAM module. The third component that requires memory is conversion of note number to period, which stores precomputed values of the period that stepper motors need to be stepped for a given MIDI note number. At 24 bits per note, all 128 MIDI notes take up 3072 bits in total in our DRAM.

All three of these components can be optimized for greater memory efficiency. To start, our BRAM containing chords can be optimized in a multitude of ways. The first is that instead

of taking a sampling based approach, we could store notes paired with the amount of time that they should be played. Thus, instead of storing 24 samples of the same note, we could store a note once with a duration value equal to 24. In addition, instead of adding blank lines into our BRAM for periods where no notes are played, we could add a "rest" chord with the same duration value.

These same optimizations could be applied to our user input memory. In particular, because of the nature of the user input being MIDI, we could move our user input memory further upstream in our pipeline, to log MIDI messages and the time they were received. This would allow us to simply replay these messages during the improvisation phase, and have the same effect of reducing blank or repetitive lines in our BRAM.

If needed, we could eliminate the use of a MIDI note number to step periods lookup table, but this would result in additional latency in our design.

C. Checklist

Across the full scope of our checklist, we successfully met the Commitment, Goal, and Stretch Goal objectives. At the commitment level, we were able to receive MIDI input over UART, parse the MIDI note-on and note-off messages, and accurately control a stepper motor to play a specific pitch. Building on this foundation, we achieved our goal by showing polyphony, with multiple stepper motors simultaneously rendering harmonies given by our MIDI input. We were even able to meet our stretch goal, implementing a pseudo-random improvisation and call-and-response module driven by live MIDI input and demonstrating scalable, clean orchestration across 24 different motors, including effective volume control by varying between 1-4 motors playing a note at a time.

D. Implementation Insights

One issue that arose during testing was that when trying to play a note in the range E3 to G#3, the small stepper motors (but not larger ones) jittered. We suspected that this could be a resonance-related issue which only happens for a specific range of frequencies as the same RTL design did not cause the larger stepper motors to jitter.

Currently, our system only supports playing notes on stepper motors. It should be possible to additionally support percussion with relays by adding the capability to parse such MIDI messages to the `midi_processor` module and switching the relay on or off accordingly. A simpler but related addition is to use one or more relays as a metronome, which should allow the user to synchronize better with the system's improvisation.

E. Additional Use Cases

While beyond the scope of this project, this system could be easily modified to play pre-written music. Because the chords are preloaded, support up to 5 notes at a time, and are sampled at 24 chords per second, you can actually create melodies with harmony and load them onto our FPGA. In it's

current implementation, we can modify the notes in our python script (to create the .mem file) to play any 24-second song, as long as the rhythms in the song can discretize well into our sample rate (in other words, the tempo might need to be changed). In addition, while our system currently only uses a 25-note keyboard, it can be expanded to support a larger MIDI keyboard. We tested this by switching octaves on our smaller keyboard, and we saw the system could support both an octave above and below our current keyboard capability. Thus, a larger, piano-sized keyboard could be used, which would be easier to play on.

VI. INDIVIDUAL CONTRIBUTIONS

A. Code

Than implemented the `stepper_out_gen` and `serializer` modules, as well as designed the PCB and mechanical parts (acrylic mounting board, stand, and motor flags).

Ben implemented the `midi_processor` module, which includes UART receiver, MIDI processor, and improvisation.

We both collaborated on testing and debugging the system. Our code can be viewed at <https://github.mit.edu/6205F25/fa25-6205-team60>.

B. Report

Than wrote the sections II, IV, and appendix. Ben wrote the sections I and III. We collaborated on the sections V and VI.

APPENDIX

DRIVER ELECTRONICS (THAN)

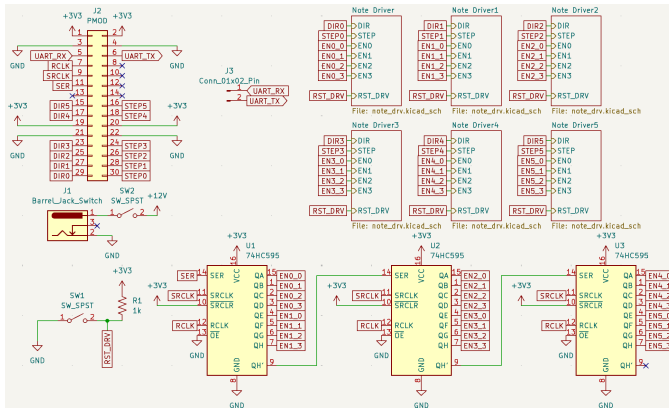
The schematic diagram of the driver electronics is shown in Fig. 5. Note that decoupling capacitors are omitted to prevent clutter.

ACKNOWLEDGMENT

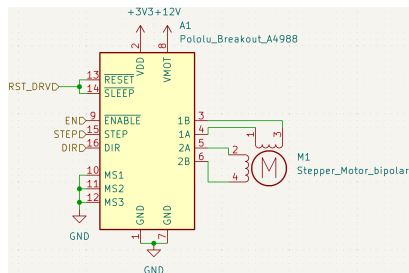
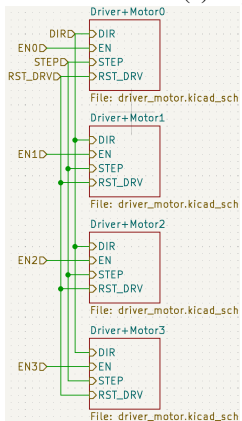
This work would not be possible without the financial support of the 6.205 class budget, which allowed us to purchase the necessary equipment, including our keyboard, adapters, stepper motors, and custom PCBs. In addition, we are grateful to the support from our mentor, Yolanda Gao, who has guided us through this project. Finally, we would like to express our gratitude to Prof. Joe Steinmeyer, for not only teaching a great class, but putting up with our numerous PCB revisions throughout our design process.

REFERENCES

- [1] MIDI Manufacturers Association Incorporated, *MIDI 1.0 detailed specification*, version 1.0, Los Angeles, CA, 1996.
- [2] Xilinx, Inc., "7 series FPGAs data sheet: Overview (DS180)," Xilinx, Inc., 2020.
- [3] Allegro MicroSystems, "DMOS microstepping driver with translator and overcurrent protection," Allegro MicroSystems, Manual A4988, 2022.



(a) Top-level schematic diagram



(b) Schematic diagram (c) Schematic diagram for one motor and for one note driver driver

Fig. 5: Schematic diagram for the driver electronics