EE108a Final Project Fall 2012-2013
Group 18
Clarence Chio, Dominic Delgado, Matthew Cooper
10 December 2012

"MIDI real-time keyboard synthesizer and multi-voice music player"

Specifications

Human Interface: We are able to connect, through an rs232 communication link to receive real-time MIDI instrument data. We are using a MIDI synthesizer (in testing, we used Matlab) to send MIDI signals over the serial input. We are using dip switches to switch between a lab4 version of the music player

Dynamics: Besides attack and decay on both MIDI and music player, on the MIDI player, the speed that we press down the key (velocity) also determines the volume of the note played.

Chords: Our MIDI player can play 4 notes at once, and our music player can play chords of up to 3 notes at once.

Harmonics: We have implemented harmonics in the music player, and implemented a piano voice and trumpet voice (to implement more voices would simply be an issue of modifying harmonics and duplicating the same modules, depending on dynamic & decay characteristics of the instrument)

Multiple Voices: We can play multiple voices at a time in our music player. We have the ability to change real-time through buttons on the FPGA how many of each voice we want. For instance, if we want 3 pianos 0 trumpets, 2 pianos 1 trumpet, 2 pianos 1 trumpet, 0 pianos 3 trumpet etc.

Midi Player Design and Implementation

The aim of the midi player was to be able to receive MIDI instrument messages and to be able to play the corresponding notes with piano like dynamics. The first step was to implement the serial communication within the project's top module. The module called RS232 accomplishes this task. (For the purposes of this project, the baudrate is 115200 because messages are being forwarded by a MIDI interface to serial converter -- such as Matlab, by reading from a connected MIDI interface, in our case.) Moving inside of the midi_player module, the first task was to collect serial bytes into one, two, or three byte messages depending on the command byte received (handled by midi_msg_capture). Full messages are then sent to midi_msg_handler for processing.

The midi_msg_handler module waits for new messages, and, upon receiving one, proceeds to handle it based on the command byte and any data bytes. The current implementation of the project can handle note on, note off, and MIDI controller messages. Controller messages are handled by simply outputting the latest controller data, and, upon receipt of new data, setting the update_all output high to notify the system of the update. Note on messages are handled by writing to an external register (midi_note_reg) with the note and velocity data at the address of the first midi_note that is not currently playing. Additionally, to make note off messages work, a RAM inside midi_msg_handler is written with the note to which the data was written at the address of the musical note being played. This RAM makes it easy to turn of the correct midi_note instantiation when note off messages are received -- the velocity data for the note is written to 0 in the midi_note_reg. Once the latest audio sample is produced, triggered by ready_to_update for the midi_msg_handler, the corresponding bit of the update output of midi_msg_player is set high for a clock cycle to update only the midi_note which has received a note no or note off request. Finally, based on the number of notes playing, midi_msg_handler outputs a multiplier values, which can only change each time a new sample is about to be generated, to be used by the mixer.

The polyphony_mixer simply finds the sum of the current samples and factors based on the number of notes playing. The sample from this module is the final sample for sending to the codec.

Finally, the midi_note module proved to be the most problematic. Ultimately, this was mostly a resource issue, considering that one midi_note instantiation exists for each note to be played. (If time had permitted, an improved implementation would involve a time-division multiplexing approach to reduce this issue.) It was hoped to obtain a full 32 note polyphony, but resources limited this significantly. In order to minimize resources as much as possible, the creation of multiple sine waves (for harmonic creation) was pipelined in midi_wave_reader. Addtionally, the calculations to achieve dynamics controll on each harmonic was pipelined to a single multiplier for each midi_note instantiation.

Of these components inside midi_note, the midi_dynamics module is the most noteworthy. In order to achieve a high fidelity dynamics system, a gain value is tracked for each harmonic's dynamics (done in midi_harmonic_dynamics). The values starts at zero, counts to 128 for the attack, and then counts down for the decay. The decay rate is determined by the state of the note (on, off, or sustain controller pressed) and the note being played (like a piano, lower notes decay slower while than higher notes).

The most unique approach used in midi_dynamics was the use of a multiplier, and the linear gain value, to achieve an exponential decay. This is done by scaling the audio sample by x^4 (where x ranges from 0 to 1). In short, th pipelined_scaler1_128 module shifts the sample to divide by 128 and then multiplies by the input multiplier. Dynamics_calc uses a single pipelined scaler to run the calculations. For example, the sample for the first harmonic is first multiplied by the harmonic's gain value. If the harmonic is in the exponential decay stage, the output of the multiplier is placed to its input and multiplied by the gainvalues for three more multiplications. (This achieves the x^4 model. Note that the linear stage causes the multiplier for the second, third, and fourth calculations to be 128.) Finally, a fifth scaling is done with the note's velocity as the multiplier. By cycling through the harmonic samples and gain values, the dynamics_calc module achieves far superior efficiency compared to using multipliers for each harmonic.

Music Player Design and Implementation

The music player that we have created is very much an extension of the lab four music player, and it was created with a focus on compatibility and sound quality. In fact, the interface is similar enough to the original that, with only a few additional ports, it can be played using the lab4_top module.

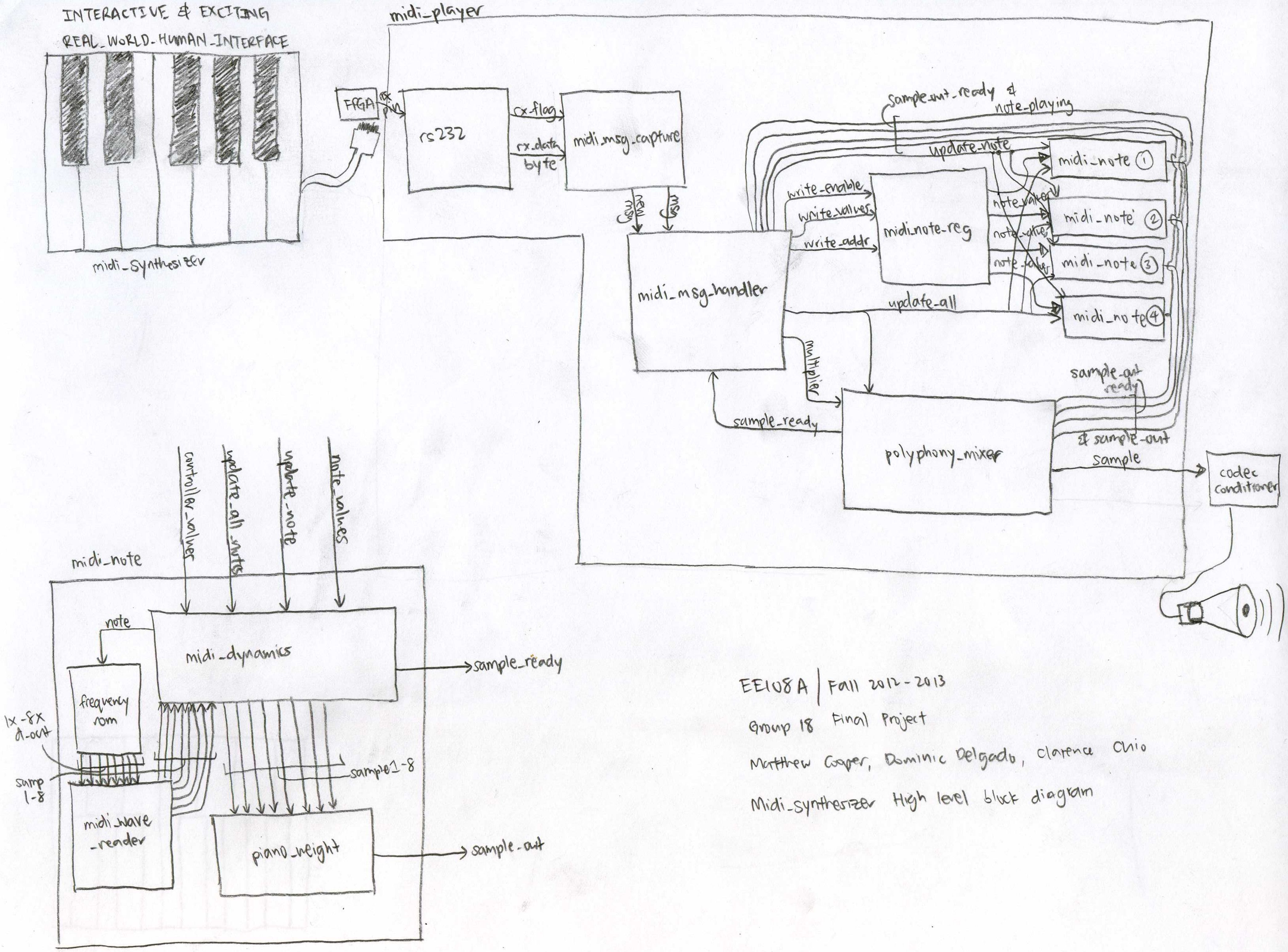
Music_player_ext (the new music player) contains MCU(unchanged), a song reader, and a chords module, within which the samples are generated. The songs are stored in 512-entry song_rom, which is managed by song_reader, and each entry consists of a bit that indicates whether the entry is a note or a time-advance, six note bits and six duration bits. The entries are fed one at a time into the chords module, except for the first bit, which helps to determine whether song_reader should send a note or wait.

The chords module houses three note players (note_player_ext, to distinguish from lab 4's note player) that are assigned values based on a select signal. These note players can create the sound of either a piano or a trumpet based on the state of three DIP switches, and each one can be toggled individually at any point when the music player is on. The note player module contains a piano voice and trumpet voice module, both of which are identical except for the weighting module, and they "instantiate" per-harmonic dynamics controllers of the same sort as are described above, into which are fed the output of eight sine readers (one per harmonic) and out of which come eight samples which are fed into the weighting modules.

The weights that we chose for each harmonic are based on graphs of Fourier analyses of trumpet and piano sounds. The reason that each voice has its own module is that, for the sake of efficiency, each weighting module was given unique bit-shifting algorithms that would have been difficult to parameterize/generalize. Multiplications by constants other than powers of two were decomposed into additions of multiplications by powers of two (i.e. x * 5 => (x * 1) + (x * 4)).

Design Challenges

This project surprisingly didn't suffer from many of the typical challenges like timing constraints, etc. Instead we found ourselves fighting resource usage. Originally, it was hoped to have 32 note polyphony, but the number of ROMs, multipliers, and general-purpose logic was far too much for the devices. Thus, the number of notes was reduced and the pipeling of sine_roms and multipliers was done. (Time did not allow for thes kinds of improvements on the music_player side.) Other than these resource issues, the primary problems were the typical challenges of implementing, testing, and synthesizing our design.

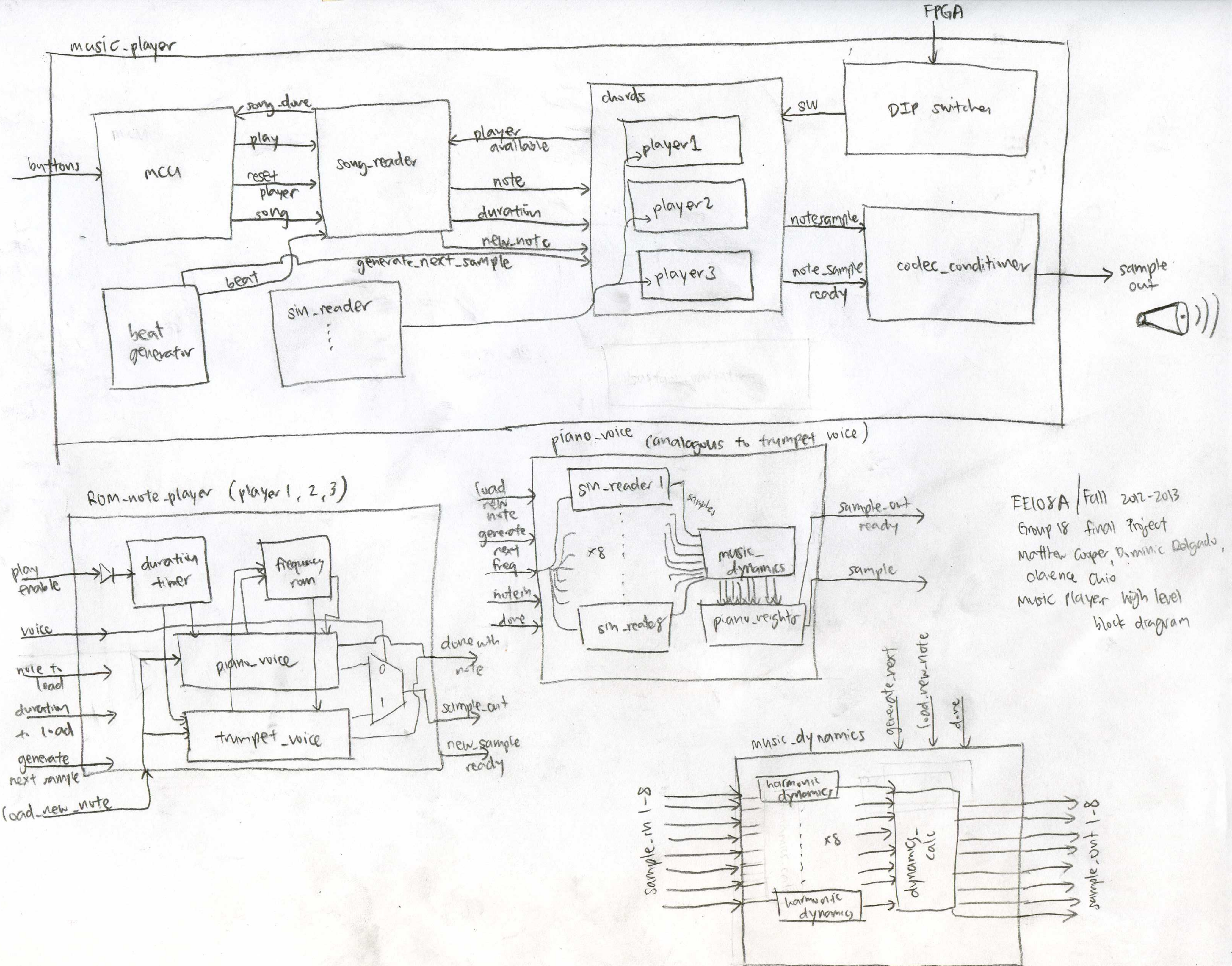


EE108A | Fall 2012-2013

Group 18 Final Project

Matthew Cooper, Dominic Delgado, Clarence Chio

Midi_synthesizer High level block diagram



EE108A | Fall 2012-2013

Group 18 final Project

Matthew Cooper, Dominic Delgado, Clarence Chio

Music Player high level block diagram