Eliriana Lleshi
Josh Stone
Ethan Oliver

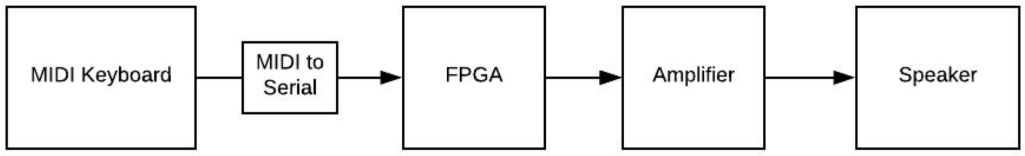
Spring 2019
Prof. Steven Bell
5/1/19

ES 4 Final Project Report: MIDI Keyboard Synthesizer

OVERVIEW

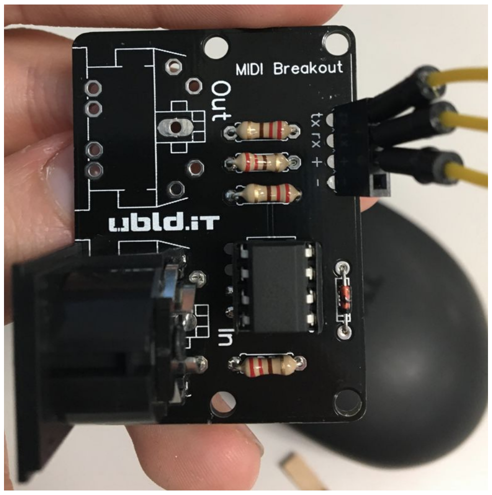
The basic idea behind our project was to create a synthesizer for the Alesis Q25 25-Key MIDI Keyboard Controller. Users interact with our project simply by pressing and releasing keys, one at a time. Octaves can be changed by pressing the UP and DOWN buttons to the left of the keyboard, up to two times each. Traditionally, MIDI keyboards are interacted with using USB connection to a personal computer, at which point software downloaded onto the computer is able to understand the signals the keyboard emits. Our intention was to bypass this by inputting MIDI signal using pins on the FPGA and interpreting them using VHDL, according to the MIDI message protocol. Because the keyboard itself does not generate sounds standalone, our goal was also to pipe these signals into a speaker such that when someone presses a key, the speaker emits the corresponding tone, just as someone would expect from a piano.

Our project consists of several hardware components, including the MIDI Keyboard itself, a MIDI to Serial breakout board, the Lattice iCE40UP5K FPGA, an amplifier, and a speaker. Closer depictions and instructions regarding the breakout board and amplifier will appear in the following section. The hardware components interact as shown in the following high-level diagram. Below it, we have included the final physical setup of our components, where all but the MIDI keyboard appear in the circuit to the left.

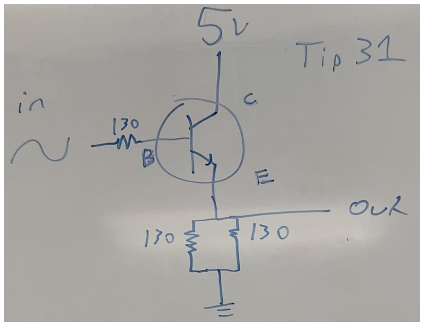


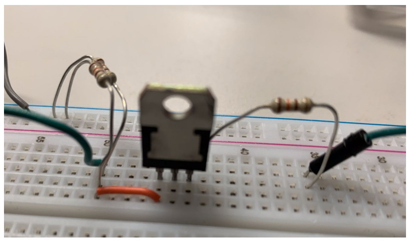
TECHNICAL DESCRIPTION

Of the above hardware components, the MIDI breakout board and amplifier were not pre-built. The MIDI breakout board takes in a MIDI cable connection and has pins *rx, tx, +, -*. These signify receiver, transmitter, power, and ground, respectively. Because we only wish to receive MIDI input, we used the *rx* pin to convert our signals from MIDI to serial input. In short, the MIDI cord connects to the breakout board, and the *rx* pin connects to a pin on the FPGA which will read in a single bit of serial data according to the MIDI protocol. It was necessary to soldier the breakout board together safely, as in the referenced instructions[1]. It can also be noted that it is not imperative to soldier the output MIDI jack, because we are only reading MIDI input.



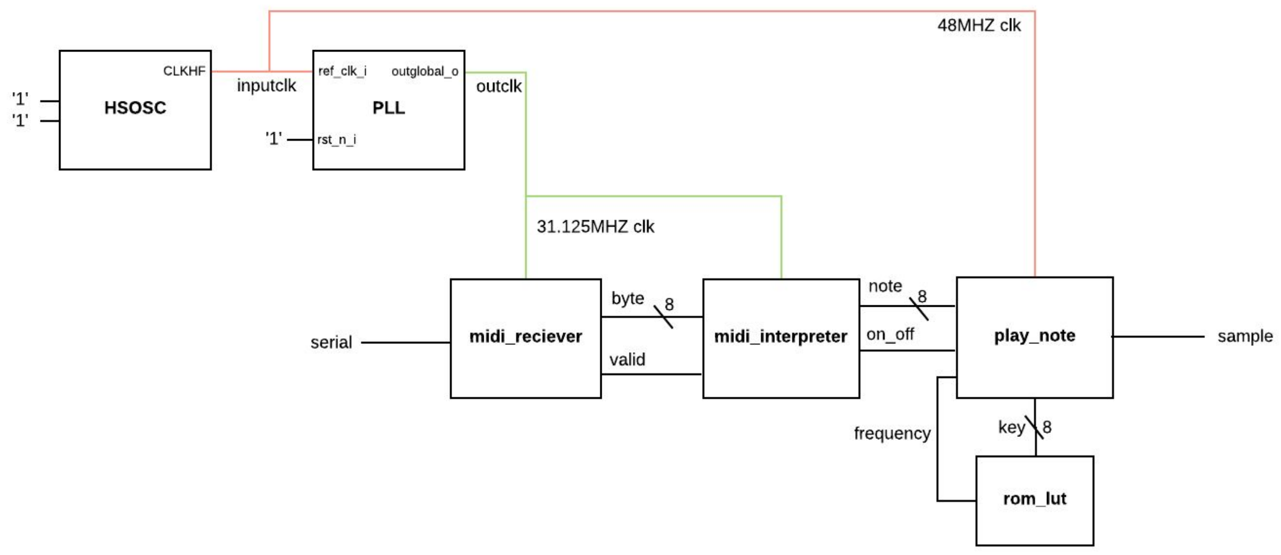
The amplifier was also constructed by hand according to the diagram below. Our amplifier is composed of three 130-ohm resistors and an ON Semiconductor (TO-220AB Case 221A Style 1)[2]. It can be replicated from the datasheet, circuit diagram, and physical picture.





The block diagram on the previous page laid out the hardware overview, with one box representing all the modules on the FPGA. Naturally, this box can be broken down into several modules. First, we will introduce the architecture by depicting module interactions, and then we will delve more deeply into what each module consists of and how it contributes to our overall goals. On the following page is the overall block diagram, which reflects the Netlist Analyzer.

[1] http://ubld.it/wp-content/uploads/2015/03/MIDI_BO_Assembly_1.0.pdf
[2] https://www.onsemi.com/pub/Collateral/221A-09.PDF



This block diagram represents everything within the `top` module, for which there is one `std_logic` input, `serial`, and one `std_logic` output, `sample`. The `serial` input comes directly from the pin on the FPGA which is connected to *rx* on the breakout board. The `sample` output is connected directly to the amplifier and speaker. Every component in between was tied together using intermediate signals and port maps in the `top` module.

Farthest left, we have the module `HSOSC`, which generates a 48MHz clock, and `PLL`, which takes this clock rate as input and outputs a clock of a different frequency, which in our case was 31.125MHz. Because we have worked closely with these in previous laboratories, we will not delve any deeper into these modules. It should be noted that the `midi_reciever` and `midi_interpreter` modules run on the 31.125MHz clock, whereas the `play_note` module runs on the 48MHz clock, for reasons soon to be explained in the module descriptions.

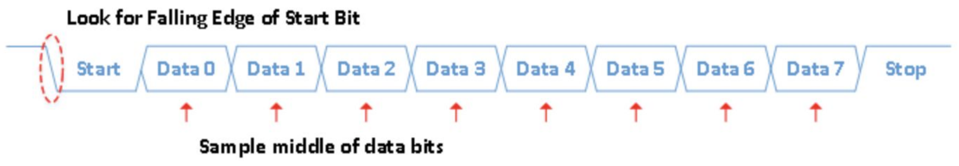
Before describing the `midi_reciever` and `midi_interpreter` modules, it is important to describe the details of the MIDI protocol. The MIDI specification[3] describes it concisely as follows:

“MIDI is an asynchronous serial interface. The baud rate is 31.25 Kbaud (+/- 1%). There is 1 start bit, 8 data bits, and 1 stop bit (ie, 10 bits total), for a period of 320 microseconds per serial byte. ... The MIDI protocol is made up of messages. A message consists of a string (ie, series) of 8-bit bytes.”

[3] http://www.gweep.net/~prefect/eng/reference/protocol/midispec.html

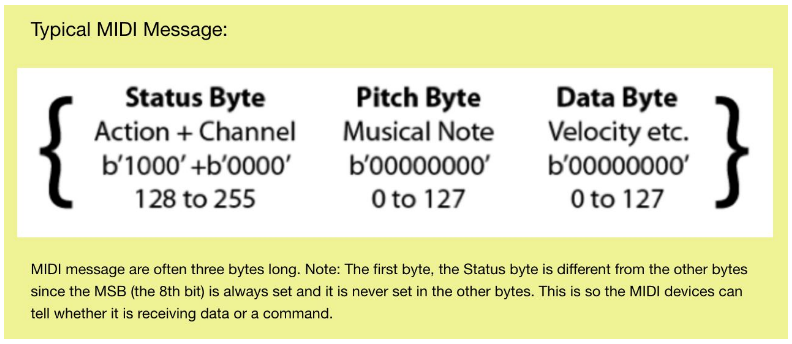
So, our MIDI keyboard emits serial data every 320 microseconds, in a message that consists of three 8-bit bytes in succession, where each byte holds different information about the actions on the keyboard. Our implementation relies on the consistency of these messages. By parsing these messages as they are input, we are able to extract desired information such as key, and manipulate it for our goal.

As mentioned in the specification, each byte is organized in the same way: 1 start bit, 8 data bits, and 1 stop bit. This is also known as the UART serial data stream, for which a visualization is included below. Relevant information lies in the data bits, and must be parsed out of the stream. This is precisely what the `midi_reciever` module is responsible for.


Look for Falling Edge of Start Bit
Start
Data 0
Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
Data 7
Stop
Sample middle of data bits

UART Serial Data Stream [4]

While the `midi_receiver` interprets every byte that is read in, it is also necessary to discern these bytes from one another, since every message consists of 3 distinct bytes. It is the responsibility of the `midi_interpreter` module to interpret every 8-bit byte extracted by the previous module. The 3 bytes are organized as follows: a status byte, data byte 1, and data byte 2, in that order, reliably in succession. Each byte contains the following information:



Typical MIDI Message:

Status Byte	Pitch Byte	Data Byte
Action + Channel	Musical Note	Velocity etc.
b'1000' +b'0000'	b'00000000'	b'00000000'
128 to 255	0 to 127	0 to 127

MIDI message are often three bytes long. Note: The first byte, the Status byte is different from the other bytes since the MSB (the 8th bit) is always set and it is never set in the other bytes. This is so the MIDI devices can tell whether it is receiving data or a command. [5]

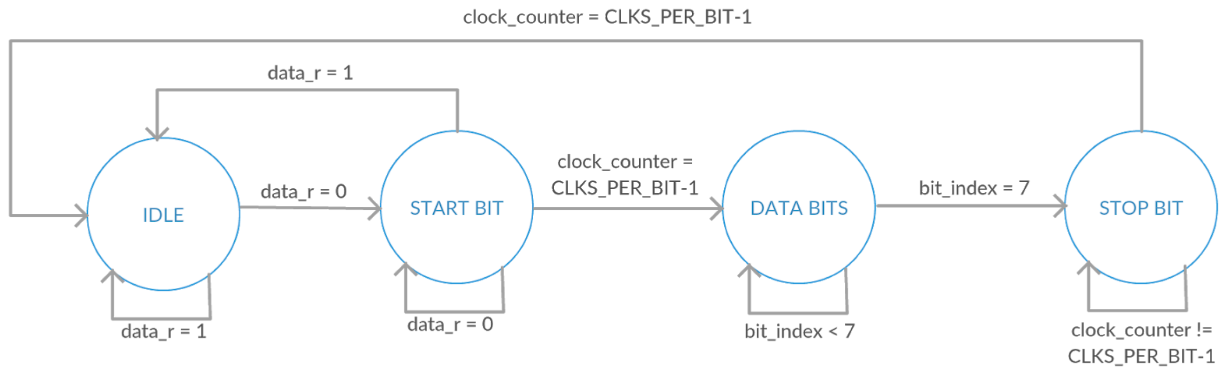
[4] https://www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html

[5] https://spikenzielabs.com/learn/serial_midi.html

As can be seen from this visual, it is possible to discern the status from data bytes because the first bit of the status byte is always high. Now that it is more clear how these messages are formatted, we can look more closely at how to implement the `midi_reciever` and `midi_interpreter` modules.

- **`midi_reciever`**

The asynchronous MIDI interface requires that data sampling occurs at the right time. This timing is directly related to the baud rate, which is 31.25 KHz. Using this value, we can find how many clock cycles are in one bit, or how many clock cycles occur while one bit is being read. Having this value allows us to create a counter that can signify the end of one bit and the beginning of another. Reaching this counter by incrementing on every clock cycle is how we know which bit we are looking at within one byte. This counter was calculated by dividing the clock frequency by the baud rate. Because we wanted this counter to be an integer, we decided to use the PLL module to generate a frequency that could easily be divided by 31.25KHz.



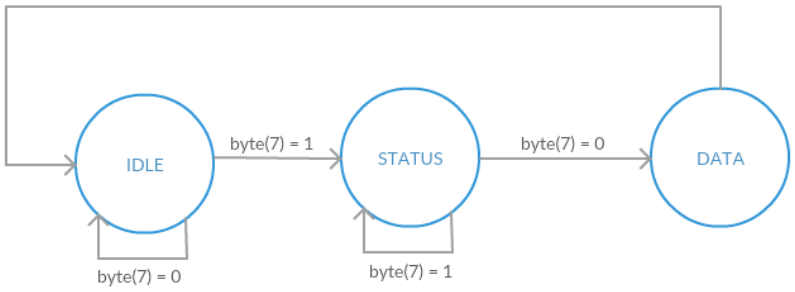
As the stream is sampled, we look for transitions that signify the state of the stream. The states in this FSM are `idle`, `start bit`, `data bits`, and `stop bit`. A FSM diagram is provided above, where `data_r` represents the current bit being read, and `CLKS_PER_BIT` represents the counter (31.125MHz/31.25kHz = 996). The state machine begins in the `idle` state. Referencing the UART serial data stream on the previous page, the line begins high, and when it drops low, this means we are beginning to sample the start bit. First, we want to make sure that we are still sampling the start bit, which is done by incrementing halfway up to the counter and ensuring `data_r` is still low. If so, we proceed in incrementing all the way up to the counter, at which point we move into the `data bits` state. We would like to sample from the middle of each data bit, so we increment halfway to the counter and save the bit to the corresponding index within the `byte` output, where the first bit read in should be sent to `byte(0)`, and so on. The index is incremented every time the counter reaches its maximum (every time a new bit is read in), until all 8 have been read in, at which point the `byte(7 downto 0)` is full of data and we transition into the `stop bit` state. The `stop bit` state must set the valid signal to high when the counter reaches its maximum, to signify that an entire byte has been read in. The state machine will then transition back to `idle` for cleanup.

- **`midi_interpreter`**

This module works on a byte level within the architecture. It is designed to interpret one full MIDI message. The MIDI message is broken down into 3 bytes. Every byte parsed by the `midi_receiever` is input into the `midi_interpreter`. Because our implementation only handles Note On/Note Off features, we only need to parse the status byte for these values (which appear in the high 4 bits) and to keep track of the entirety of data byte 1. We have dubbed data byte 2 the “discard” byte, because we do not use it. This module receives a `byte(7 downto 0)` and a `valid` bit, and must differentiate the bytes each time the `valid` bit is high, signifying that an entire byte has been successfully read in. The critical pieces of information from the message are shown in the table below.

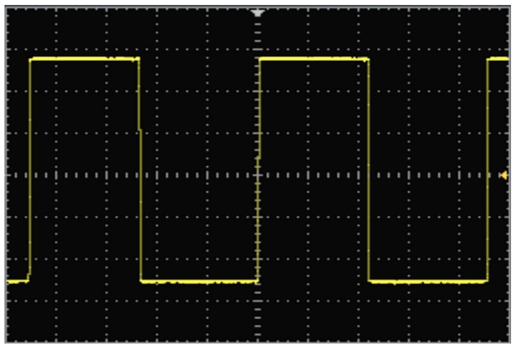
Voice Message	*Status Byte*	*Data Byte 1*
Note Off	0x80	Key number
Note On	0x90	Key number

This module is based of of 3 states: `idle`, `status`, and `data`. The FSM begins at `idle`, enters `status` when looking at the status byte, and enters `data` when looking at data byte 1. Any other conditions return the FSM to `idle`. The `idle` state waits for a `valid` bit and for `byte(7)` to be high. This signifies that a status byte has been read in and can be evaluated. The Note On/Note Off status we are interested in occurs in the four most significant bits, where a value of 0x80 sets the `on_off` signal low and 0x90 sets it low. This module outputs the `on_off` signal to prompt the `play_note` module to begin or stop emitting a signal. When `byte(7)` becomes low, this signifies the beginning of data byte 1, and thus shifts the FSM into `data` state. The entire byte delivered during this state is saved to `note(7 downto 0)` and output to `play_note` in order to play or cease playing that note, ranging from 0 to 127. After this occurs the state changes to `idle` again, completely ignoring the following "discard" byte.



- **`play_note` and `rom_lut`**

Sound is created from pressurized air vibrating a certain frequency. Regardless of the shape wave (square, sine, triangle), if it is output at the frequency corresponding to the desired note, the resulting sound will still ring at the desired pitch, but the tone of the sound may be different. In our project, we used square waves because of the pins of the FPGA are limited to a high or low voltage.



Our `play_note` module creates an oscillation by counting clock cycles up to a specified number calculated based on the 48MHz clock rate and the note's frequency. To calculate this number, let us call it *N*, we divided the clock frequency by the note frequency[6],

[6] http://pages.mtu.edu/~suits/notefreqs.html

which produces the number of clock cycles in the period of that note. We then divided this number by 2 to get the exact point at which the square wave goes from high to low. This was repeated for all 75 notes. These counters were stored in a ROM lookup-table. Each note on the keyboard corresponds to a distinct 8-bit combination. By using the UART signal functionality on Waveforms, we were able to determine the exact combinations that corresponded to a given key. All these 8-bit combinations are stored in another module called `rom_lut` with their corresponding values of *N*.

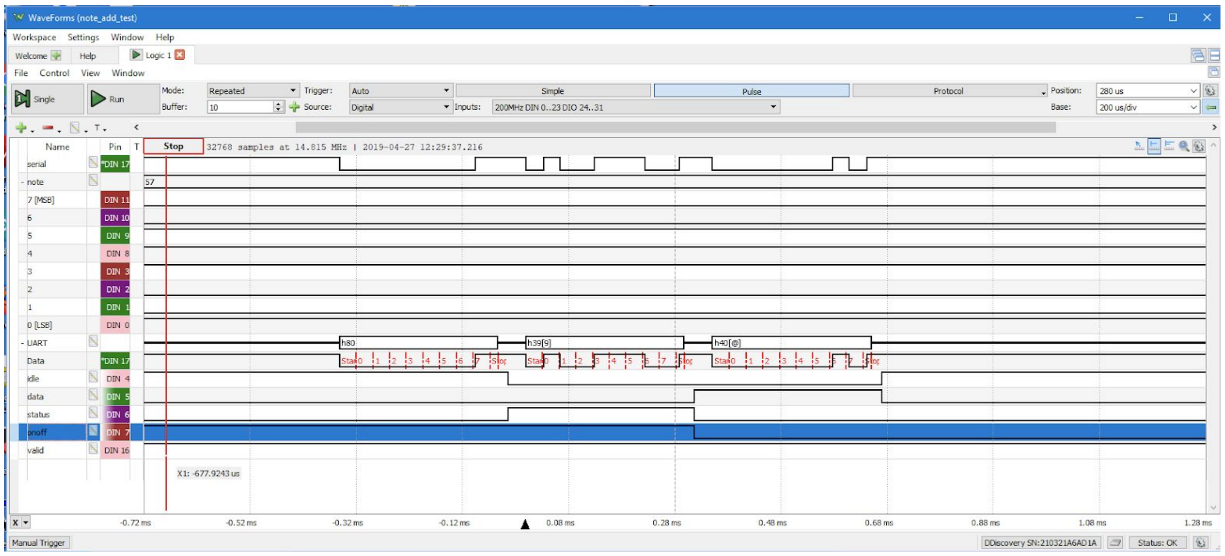
When the `play_note` module receives `key(7 downto 0)`, it passes the combination to `rom_lut`, which returns the desired counter. If the `on_off` signal input from `midi_interpreter` is high, a counter then begins counting up to the value of *N* received from the ROM, incrementing by one each clock cycle. When the counter reaches its maximum value, it switches the output from either 0 to 1 or vice versa. It continues to do this until the note `on_off` signal is low. By oscillating between 1 and 0 on every clock cycle and routing the output `sample` to an amplifier and speaker, a sound can be played.

- **Testing and Debugging**

The main debugging tool used as we were building was the Digital Discovery and Waveforms software. Timing was critical for our project to be successful. We were happy to find that Waveforms allows that you add a UART signal directly mapping to the serial input pin, and displays the entire three bytes with their values and timing. This gave us a valuable reference for the rest of our data. Each time we pressed a note, we could compare the correct bytes to the ones our FSM had parsed out.

First, it was necessary to confirm that the `midi_reciever` was working smoothly. This involved ensuring that the `byte(7 downto 0)` output contained data, and that and the `valid` bit was being set high at the right times. This was confirmed by adding `valid` as a signal and `byte` as a bus in Waveforms. Unfortunately, our group has lost track of the Waveforms screenshot for this module, however we will describe what should be looked for. Using the UART signal as reference, the `valid` bit should be set high just as one byte has been read it, and until another begins, and should begin and end low. For the `byte`, we just looked to see that the square waves were changing and flashing as we played notes, to prove that we were extracting data. We made the choice to proceed to the following module to confirm that the data as accurate. Hangups occured because of small errors in our code which made it such that the state machine would get stuck at one of the states. This was overcome by setting a signal for each state, and making it high whenever the state machine was there. This was, we were able to track down which states didn't have leavable conditions, debug that state, and check the Waveforms until the states `idle`, `start bit`, `data bits`, and `stop bit` follow each other in that order.

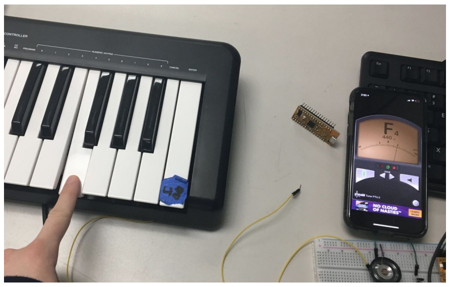
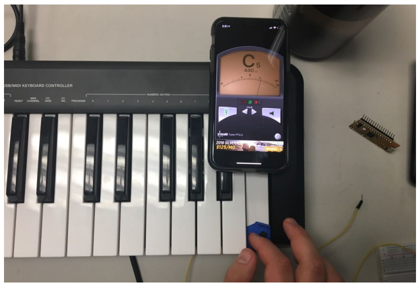
Next, we wanted to confirm that our `midi_interpreter` was functional, using similar techniques. We confirmed this by adding `on_off` as a signal and `note` as a bus in Waveforms. In this case, we have provided a Waveforms screenshot with the correct timing, after debugging. It can be seen that each state goes high just as the corresponding byte has been read in, which is a result of our implementation. This called for a similar debugging style in order to get out of states we were stuck in. The highlighted bar for the `on_off` signal can either start high or low, but should flip to the opposite after having read in the data byte (0x80 is Note Off, 0x90 is Note On, can be checked on the UART signal). Finally, we wanted to check that the note itself had been accurately extracted from data byte 1. At the top of the `note` bus, the number 57 appears, which is no coincidence. The 8-bits read in correspond to both 57 and to the h39 in our reference UART signal. Thus, our note is accurate.



```
Mode: Repeated   Trigger: Auto      Position: 280 us
Buffer: 10       Source: Digital    Inputs: 200MHz DIN 0..23 DIO 24..31   Base: 200 us/div
Stop   32768 samples at 14.815 MHz | 2019-04-27 12:29:37.216
```

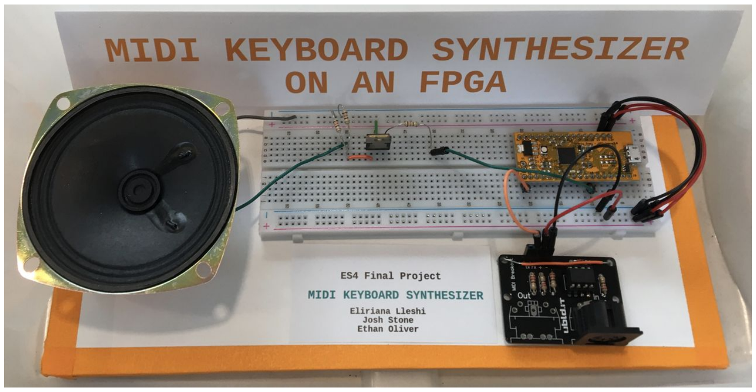
Our `play_note` and `rom_lut` modules were very simple and their functionality relied directly upon receiving accurate data from the previous modules. We visually checked the code to see that our counter increments as we wish, until the looked up frequency, and that a signal was being output on the Netlist Analyzer.

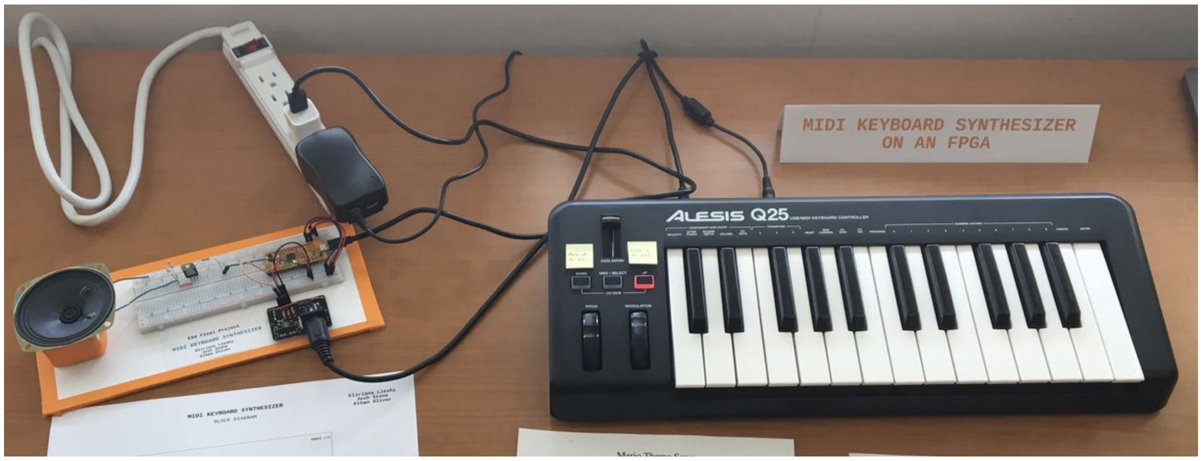
We decided to try out the entire setup on a limb, and were pleased to find that it worked on the first attempt. While the speaker emitted sound, we wanted to confirm that the pitch was true to the key pressed. We knew that the highest key on the MIDI corresponded to h48 on the UART data byte 1, which is also known as C5. We confirmed its pitch with an iPhone app, and repeated the process for other notes and for new notes when the keyboard was extended several octaves.

RESULTS

We are happy to report that the state of our project is working and complete! Because our project relies upon sound to confirm functionality, it isn't possible to provide proof of this in a report. There are videos of people interacting with the synthesizer that will be uploaded to Dropbox. Some pictures of our final product are included below.





Some metrics related to overall performance involve our ability to play 75 notes in total. This was made possible by the UP and DOWN features on the MIDI keyboard, which allow the relatively small 25-key MIDI to be extended several more octaves above and below what is visible. The maximum bytes processed within a certain timeframe relates directly to the baud rate of the MIDI, which we have mentioned delivers a serial byte to input every 320 microseconds. This kind of timing makes it so that the human ear believes the sound is emitted instantaneously.

There aren't any explicit bugs in our project, considering that our goal for implementation was to play one note at a time, and this was delivered. However, when switching between notes, it is necessary to cleanly lift off of one note before selecting another, or else the following note cannot transmit data when the other note is still transmitting data, and thus makes no sound. While this makes a lot of sense to us given our implementation, from the perspective of the user experience, our solution is might be limited.

REFLECTION

As a team, we feel that our project went quite well. We were very pleased to see our peers and some faculty interact with our design, some even playing short tunes. If we had to do it all over again, we would start writing our VHDL code earlier, and solidifying the `midi_reciever` and `midi_interpreter` modules earlier in the game. These modules comprise most of the difficulty of this project, and if they are working, it is possible to add on more features. If allowed another iteration of this project, we would have added features such as velocity, which can be parsed from data byte 2, and reflects the strength with which the key is pressed. Most of all, we think it would have been impressive to be able to play multiple notes at a time, which could be done by taking advantage of the multiple channels in the lower 4 bits of the status byte.