MIDI Synthesizer

Engs 31 Final Project

Matthew Timofeev and Kevin Lin

06 June 2022

Abstract

A midi synthesizer was designed and implemented from ground-up on an FPGA in VHDL. The synthesizer consists of a MIDI keyboard signal receiver, a waveform generator which interfaces with a sine wave lookup table, a SPI transmitter, and a digital-to-analog converter for the waveform output. The synthesizer mimics a pipe organ and supports playing multiple notes on the keyboard.

Contents

Introduction

In this project we designed a Music Synthesizer that generates the sound of a pipe organ from a keyboard using the Musical Instrument Digital Interface (MIDI) protocol. MIDI is a serial communication protocol that outputs a series of bits that convey status, velocity, and value information about a musical note on a digital instrument. We developed a MIDI receiver, pipe organ waveform generator, and SPI transmitter. The project was designed in Vivado and implemented on a Basys 3 FPGA.

Design Solutions

Specifications

The MIDI protocol sends bits serially at a bit rate of 31.25kbit/s. Data bytes are framed by a start bit ('0') and a stop bit ('1'), which differentiate between the three data bytes that makeup a MIDI message. The first byte is a status byte, which contains information on the note status (on/off) and the MIDI channel being used by the instrument. The second byte is a value byte, which indicates the current note pitch. The third byte is a velocity byte, which indicates the speed at which the note is pressed.

The hardware implementation uses the onboard 100MHz oscillator divided down to a 1MHz system clock.

Inputs

The synthesizer has a single input, the bitstream following the MIDI protocol being output from the digital instrument. Note values from C4 to C6.

Outputs

The synthesizer has three outputs. A chip select digital signal, a system clock signal, and a data bitstream that outputs at 1MHz and contains sound data in the form of a sine wave. In the hardware implementation, these signals are output to a Digilent PmodDA2 DAC, then a PmodAMP2 AMP which sends a voltage signal to a mono-speaker.

Operating Instructions

Ensure the circuit is wired according to the below tables and image. The Basys 3 should be programmed with the bitstream file and turned on. To operate, use channel 1 on the MIDI instrument and then input through the MIDI instrument according to the instrument's operation instructions.

Port Connections

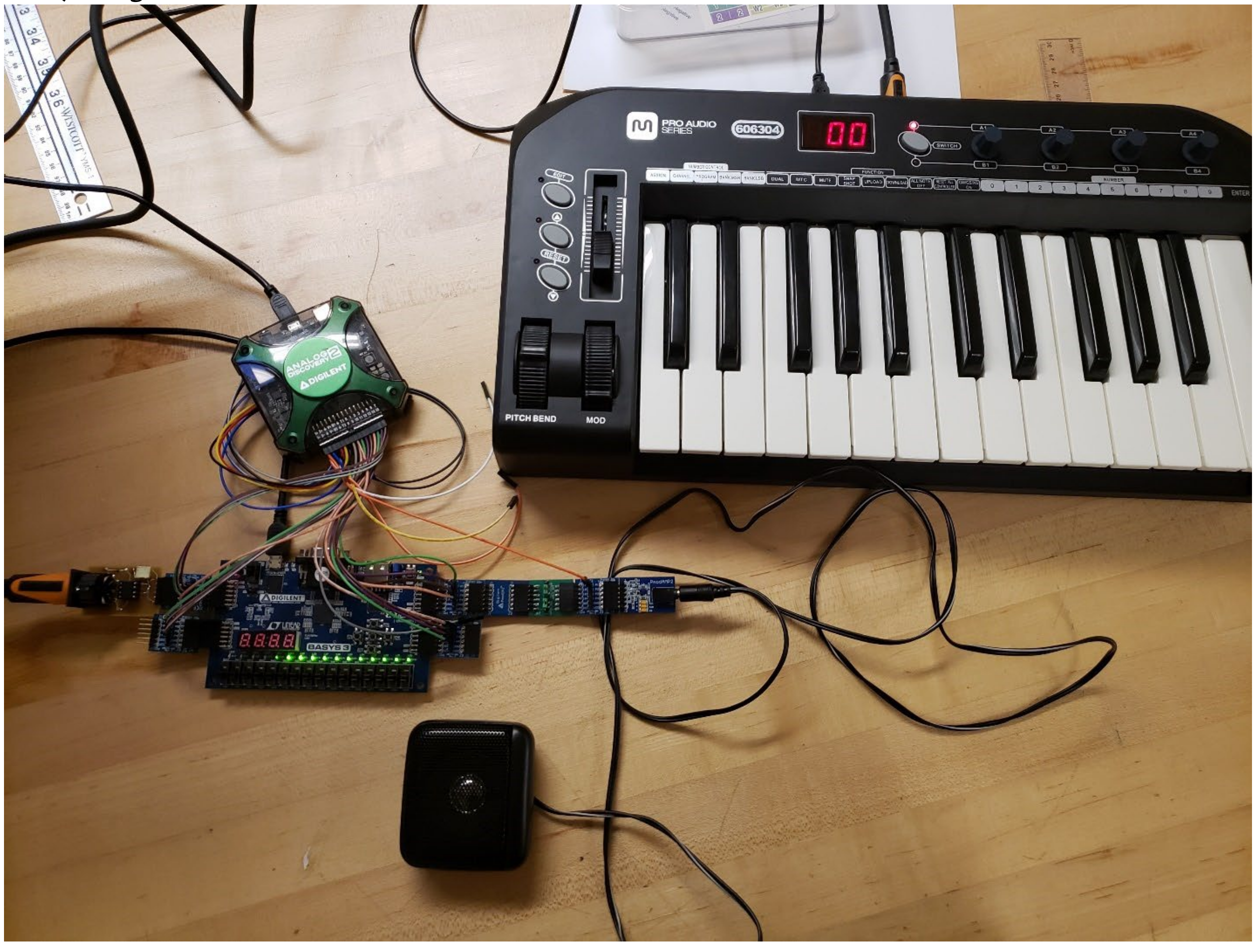
Port	**Signal**
JA1-6	PmodAD1

JB1-6	PmodDA2 (cs, data, xx, clk, GND, 3.3v)

Chip Connections

Output	**Input**
MIDI Instrument	PmodAD1
PmodDA2	Thayer Gain Chip
Thayer Gain Chip	PmodAMP2
PmodAMP2	Mono Speaker

Setup Usage



The usage setup is shown here. The FPGA is connected to the MIDI keyboard on the left using a custom port (MIDI signal on JA1). The data_valid port is on JB1, and the serial dac_data_out is on JB2 (right side of FPGA), which goes to the PMOD Da2 DAC. This then outputs to the amplifier, and finally the speaker. The user presses keys on the MIDI keyboard, including chords, and the sound is outputted by the speaker. The oscilloscope is hooked up for debugging purposes in this picture. For the purposes of this project, other keyboard functions such as pitch bend are not implemented, and are treated as irrelevant signals since they do not contain the “note on” or “note off” status signals.

Theory of Operation

The Mathematics of Sound

All musical notes, harmonies, and timbres (textures of sound) can be created by superpositions of sine waves of varying frequencies and amplitudes, as seen in the Figure 1. Mathematically, a superposition is a simple sum of displacement values at every point in time. Thus, simple sine waves can combine to create complex waveforms, which can sound like various musical instruments like violins or organs, or sound like harmonies, like major and minor chords. A synthesizer exploits this principle of superposition of sine waves to create notes.

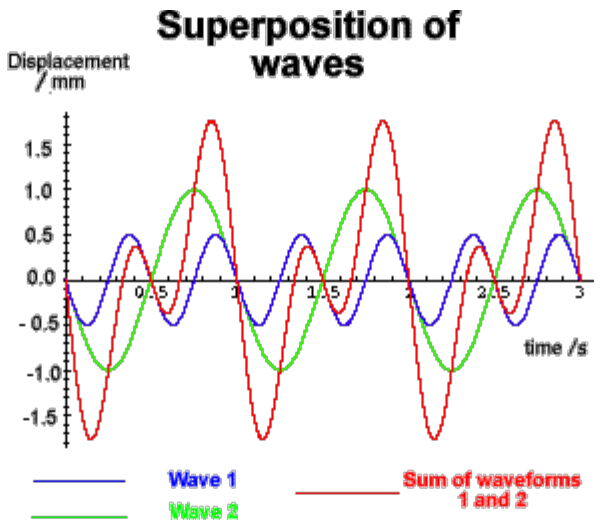


Figure 1. Image credit: Wikipedia

$$y(t) = \sum_{n=1}^{N} \sin\,(w_n * t)$$

y(t) is the resultant displacement, and each w_n is the angular frequency of the component waves.

However, in practice, storing multiple sine waves of varying frequencies in memory is not necessary to achieve the rich variety of resulting waveforms. One well-sampled sine wave cycle (from 0 to 2pi radians, with enough points in between) is enough, since by skipping locations in time, the whole cycle can be traversed quicker—exactly the same as having a higher frequency cycle. With enough sample points, the resolution loss is minimal and not audible to the ear, which allows the synthesizer to save space in memory.

Using these two ideas, one arrives at a new formula for the output waveform.

$$y(t) = 1/N \sum_{n=1}^{N} wave\_values[skiprate_n * t]$$

Where wave_values is an array of displacements, and $skiprate_n * t$ is the calculation for the index in the array, and N is the number of notes being played. Thus, the skiprate mimics the angular frequency. The factor of 1/N is to scale the resulting waveform to the same amplitude as the individual notes.

Implementation

The synthesizer code is divided into components: a clock divider (from the 100MHz on the FPGA down to 1 MHz), a controller, a datapath which keeps a running register of note velocities (either 0 or not), a sine wave lookup table (a BROM named “blk_mem_gen_y”), a math component which calculates the waveform based on the note velocities and the BROM, and a SPI transmitter which sends serial data to the digital-to-analog converter (DAC).

MIDI Protocol

The synthesizer uses the bitstream generated according to the MIDI protocol from the digital instrument. MIDI protocol consists of asynchronous, SCI-like transmission of data at 32,150 bits per second, or one bit every 32 microseconds. MIDI protocol idles high (on 1). A signal consists of three sequential 10-bit data packets, each beginning with 0, and ending with 1. The 8 middle bits of the first packet contain 4 channel bits (which are ignored here) and 4 status bits—“1000” for “off”, “1001” for “on”. The second packet encodes which note is being played. The third packet encodes the velocity of the keypress. The time between the packets is not fixed, but always has signal “1” throughout. A sample signal is shown in figure 2.

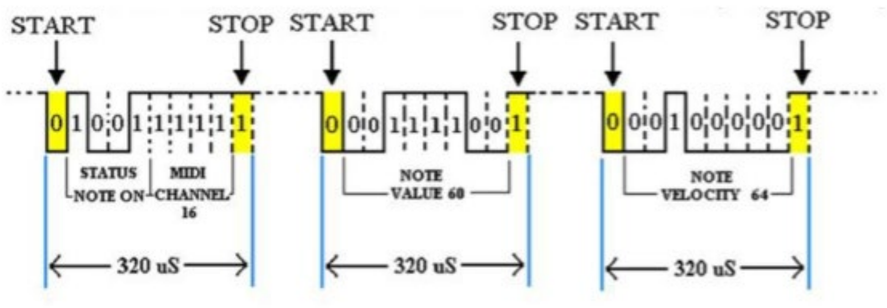


Figure 2. TIMING DIAGRAM FOR MIDI NOTE ON

Image credit: Prof. Hansen

The three data bytes are parsed and the status, note value and note velocity are bit shifted into each corresponding register.

The controller idles in the get_status state, with statshift_en high. The status packet is sent by the keyboard first, and begins with a falling 1 to 0. When the controller detects this, it waits half a baud period before sampling the zero and letting the serial data be shifted into the note_status register in the datapath. An internal counter keeps timing to the baud period, and another counter tracks the number of bits shifted in.

When 10 bits have been shifted, the datapath checks the note status, looking for either “x1001xxxxx” or “x1000xxxxx” in the status packet. Through combinational logic in the datapath error-checking logic, the signals “note_on”, and “note_off” go high in each of the cases, respectively, and indicate to the controller when to shift in data. If the status packet does not contain the note on or off bits, then neither “note_on” nor “note_off” go high for the controller, and the controller clears the status data it has shifted in, returning back to the first get_status state.

If the data does concern a note, the controller then enters the velocity_shift state, enabling note value to be shifted into a shift register. Once this has finished, the note velocity is shifted in. (The controller waits for the packets to arrive). The note value is mapped to a memory location in the velocities register (locationmem = noteval – 36). If note_on is high(note is pressed), the velocity is stored at that location in the velocities register. If the note _off is high (note is released), that location in the velocities register is reset to 0. See datapath and controller block diagrams, FSM.

Waveform generation algorithm

Concurrently, the entire velocity register is continuously looped through by the math component and a calculation is performed to determine a 16 bit signed amplitude of the final waveform. The result is determined by obtaining the value of a specific time point in the period of a sine wave. To create the frequencies of different notes, we utilized different skip rates when counting through the points of the sine wave to simulate a shortened period and higher note frequency. See Appendix for calculation of skip rates.

The math component keeps a running noteaddress (from 0 to 24), which increments every other clock cycle. (Every other clock cycle to accommodate a delay from the sine wave lookup table). Each note’s phase is stored in a phase_register, which is computed based on the global time (named “phasecount”) multiplied the skiprate for the note. (See Theory above and Appendix). This phase is sent as raddr_val_port to the LUT, and the LUT returns the y_value.

As the velocity register is looped through, if the velocity is non-zero, this y-value at the current phase in the wave is added to the running numerator. However, since this value obtained from the LUT is a signed number, performing simple addition would lead to positive number overflows caused by the MSB 1, even though the numbers should actually be negative. The resulting waveforms would be discontinuous and produce distorted noise. To solve this, the LUT values were shifted up by ½ amplitude, thereby making the sine wave range from 0 to the amplitude, instead of – 1/2 to 1/2 amplitude. By keeping numbers all positive, the behavior is more predictable, and overflows can be easily corrected by allocating more bits to the running sum.

To handle multiple notes at once, the running sum must be divided by the number of notes played. A bitshift strength reduction division operation (multiply by mult_const, shift right by bitshifts) is performed on the result to keep a constant max amplitude, avoiding slower integer or floating point division. The number of bitshifts and the multiplicative constants are calculated in the appendix. Finally, an additional right bitshift is performed to ensure no overflows in the result, and a smooth waveform. Every time the math finishes running through the notes, the final result is updated, and new_data goes monopulse high for the SPI transmitter.

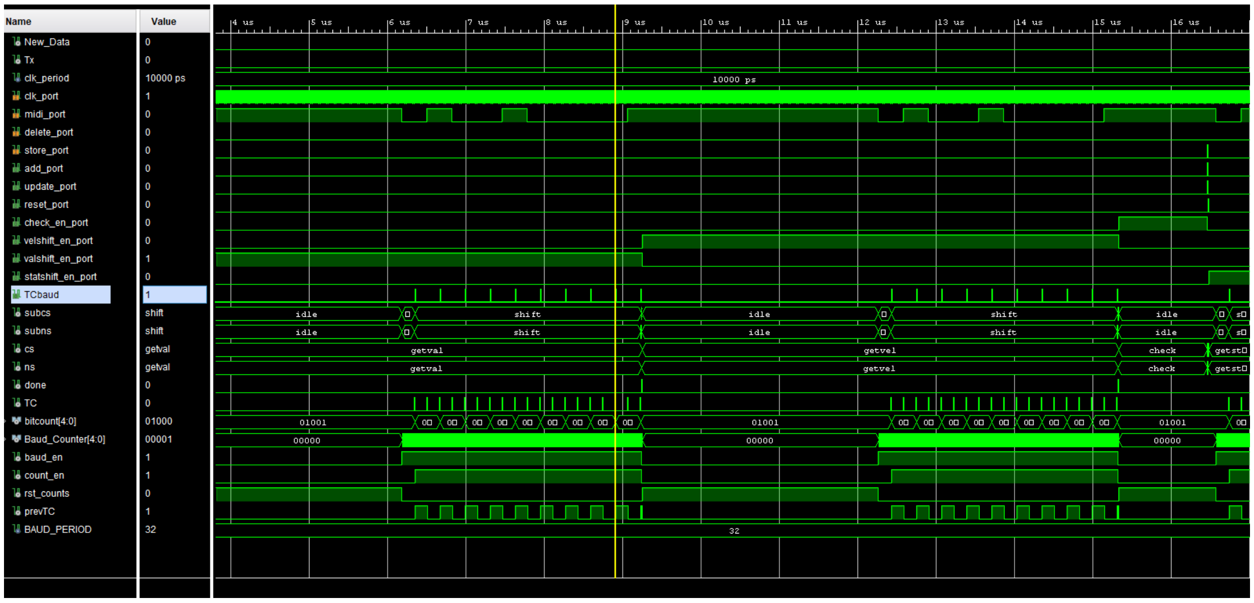
SPI Transmitter

When the SPI transmitter receives new_data high monopulse, it enters a load state and loads the result data from math into its shift register. It then enters the shift state. The SPI data_valid signal goes low and transmission begins. The parallel-input wave result is shifted left and the MSB is sent out in series by the SPI transmitter to the PmodDA2 DAC. When transmission ends, data_valid idles high at 1. The DAC converts the series in bits to an analog signal. The Thayer Gain Chip and PmodAMP2 then amplify the analog signal to output the voltage waveform to the mono speaker.

Construction and Debugging

Rather than testing all components of the synthesizer individually and painstakingly simulating all expected signals from one component to another, testing took a Russian-nesting-doll approach. The central component—the controller—was tested first. Once it worked as expected, the datapath was included in the testbench and hooked up to the controller to test the two together. Then, the math and LUT were included, and finally, the SPI transmitter.

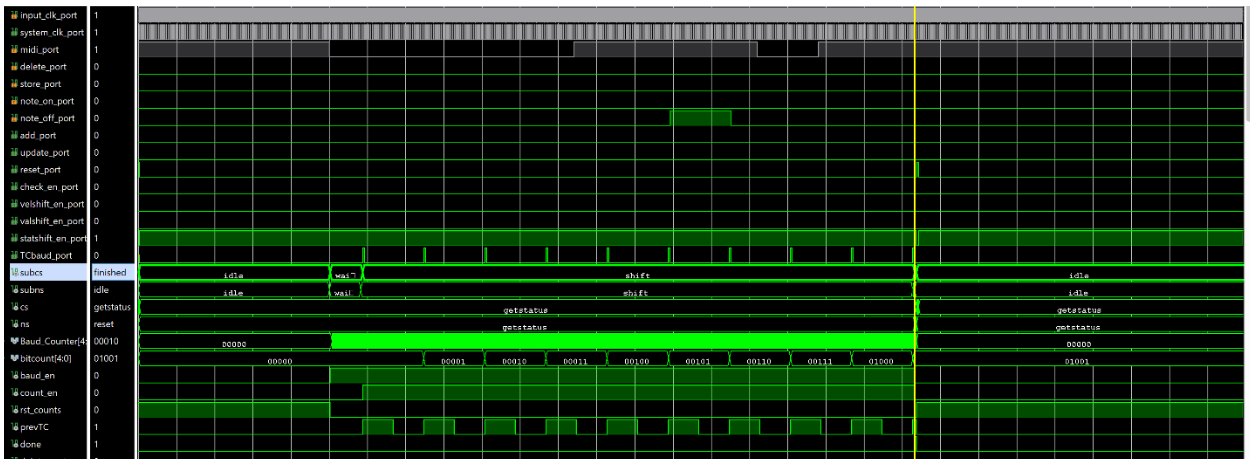
Controller FSM



Name	Value
New_Data	0
Tx	0
clk_period	10000 ps
clk_port	1
midi_port	0
delete_port	0
store_port	0
add_port	0
update_port	0
reset_port	0
check_en_port	0
velshift_en_port	0
valshift_en_port	1
statshift_en_port	0
TCbaud	1
subcs	shift
subns	shift
cs	getval
ns	getval
done	0
TC	0
bitcount[4:0]	01000
Baud_Counter[4:0]	00001
baud_en	1
count_en	1
rst_counts	0
prevTC	1
BAUD_PERIOD	32

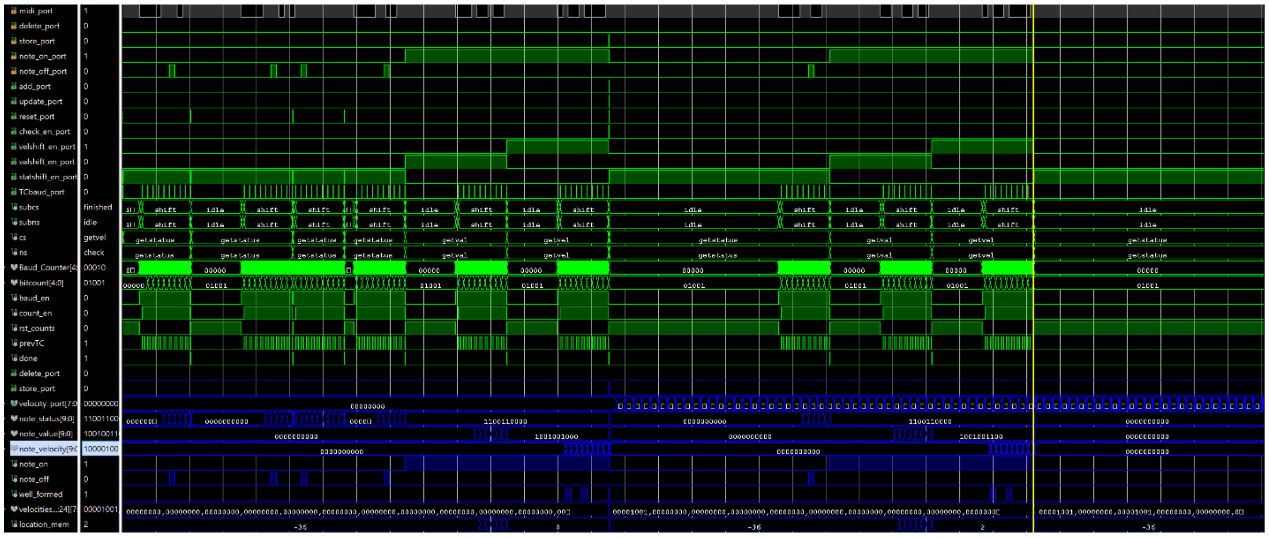
This test confirms that the FSM and sub-FSM are both operating correctly with the intended outputs to the datapath. The simulation starts with the FSM shifting the note value in and the sub-FSM in an idle state. Once midi_port starts transmitting and equal zero, it moves to the waithalf state where it waits half a baud period until TC goes high to move into the shift state. Once the 10 bits that represent the value have been shifted in, the valshift_en_port goes low and the velocity begins shifting in instead. After the FSM check state, the add_port and update_port go high as intended and then the reset_port goes high the following clock cycle.

Initially we had issues with ensuring that the FSM and sub-FSM worked in concert with one another. However, by testing on the testbench and simulating inputs from the midi_port we were able to confirm proper operation of both FSMs and outputs to the datapath.

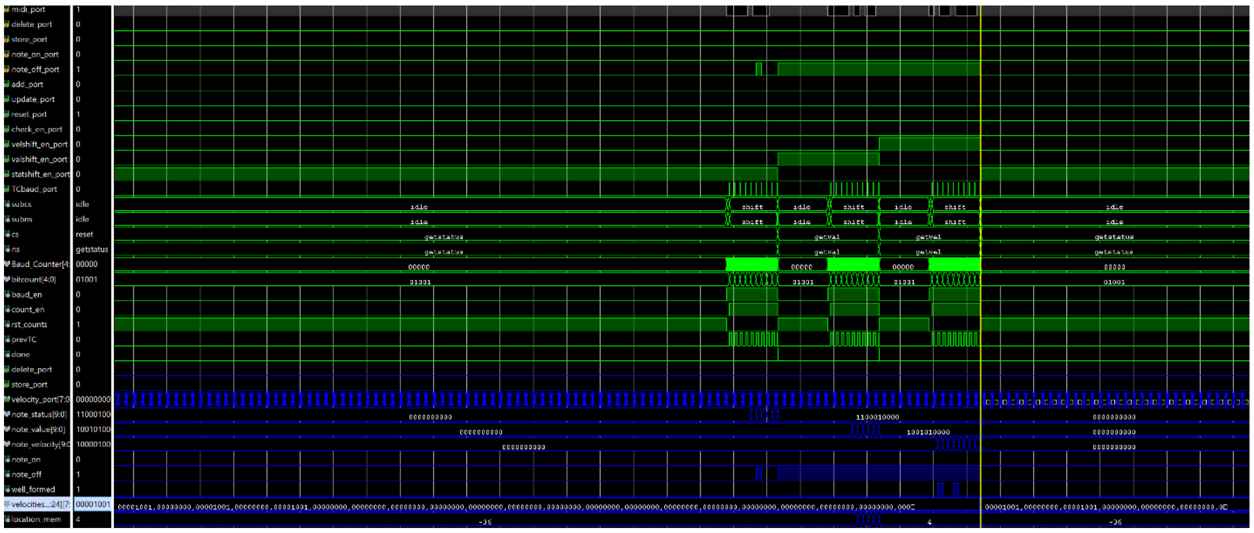


Here we test if the controller validates messages properly and will skip a simulated bad signal. For the status byte we send "0000000000" to the simulation. We see that neither the note_on_port nor the note_off_port go high. This causes the FSM to return to the reset state from the getstatus state when done goes high because the status signal was detected to be invalid.

Datapath

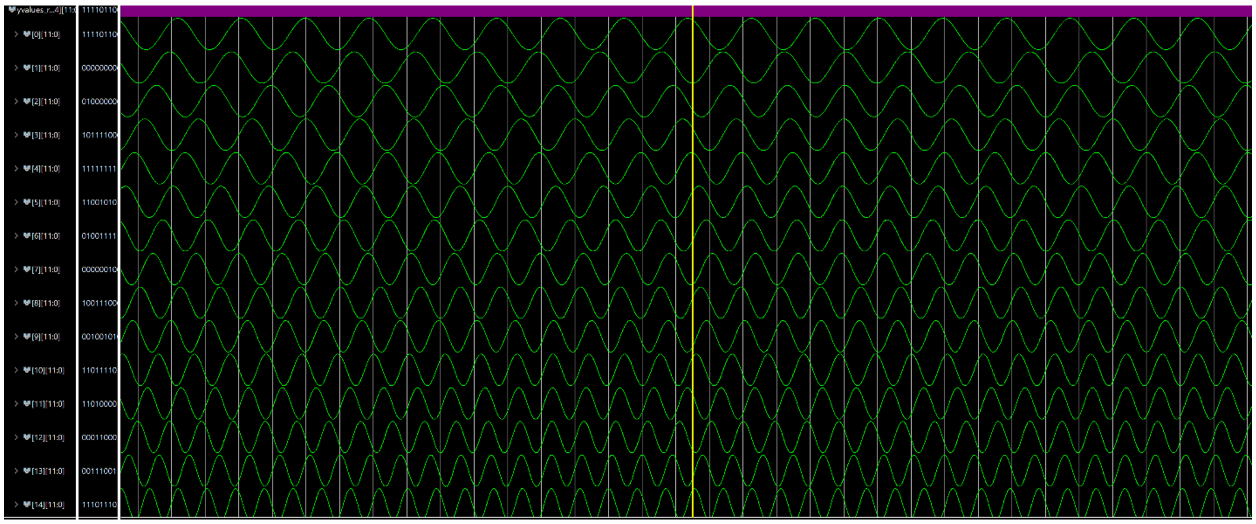


Here we test that the datapath can properly save a note. We expect that when add_port and update_port go high, the current velocity is loaded into the velocity register at the address location_mem. We can see that add_port and update_port go high twice in the simulation. The first time the ports go high, the location_mem is set to 0. We can see that the velocity register(0) gets set to "00001001" as expected the first time through. The second time tat the add_port and update_ports go high, location_mem = 2 and we can see index 2 of the register being set to "00001001" as expected as well.



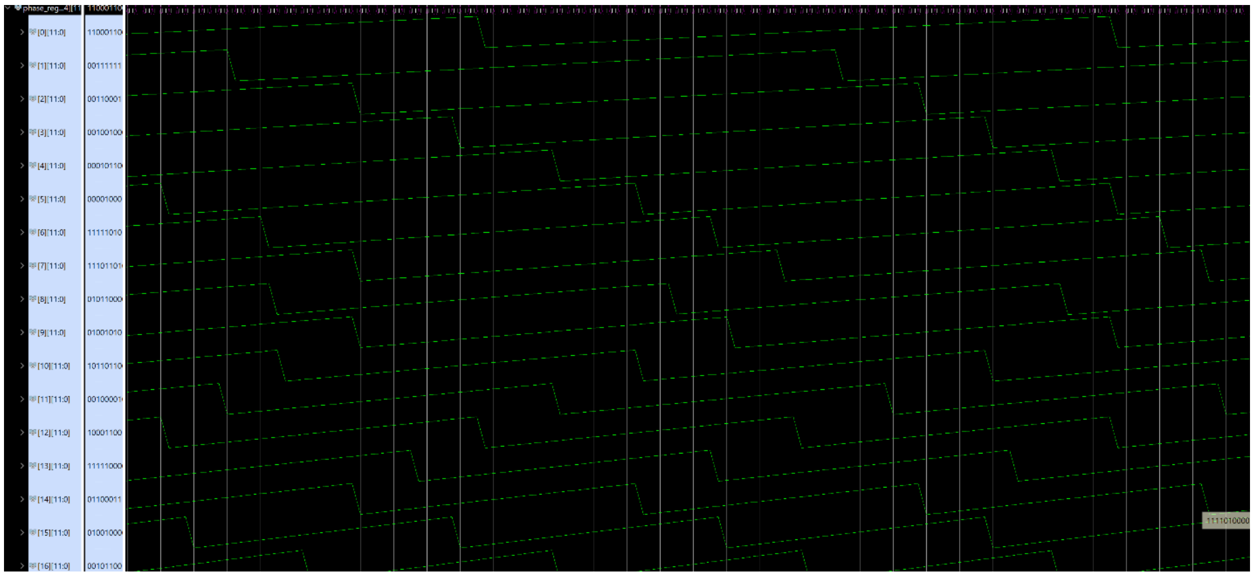
Here we test if the datapath delete note functionality works. We'd expect to see a value at address location_mem in the velocity register get reset to 0 when the output. By sending the "off" note status of "10001000" for note value "001010000", we simulate releasing the note at address 4 in the velocity register. As expected, we can see that update_port goes high and the value at address 4 gets set to 0 from "00001001".

Math

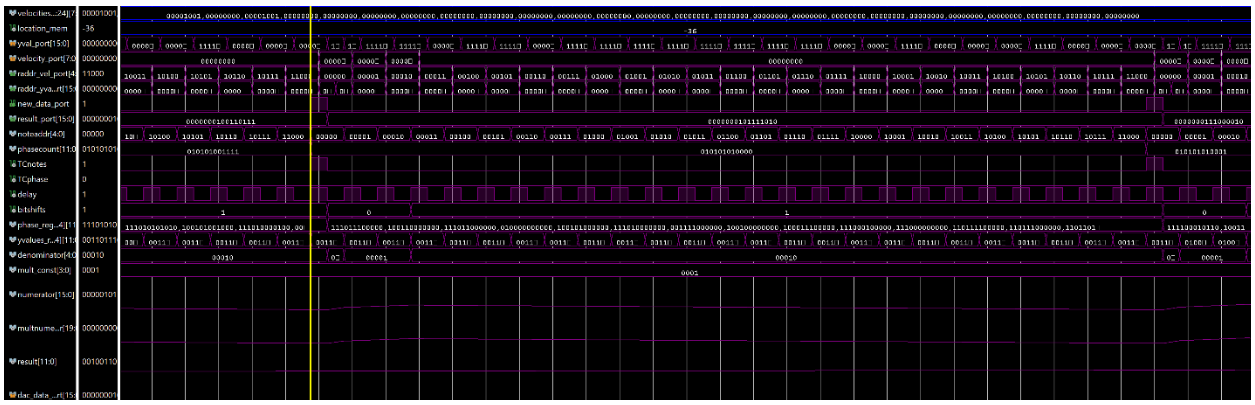


This simulation is to confirm that the yvalues in the yvalues_reg are being calculated properly from the yval_port and Sine LUT. We can already see that the yvalues are in the form of a sine wave as expected. From address 0 to 14, we can see that the frequency of the wave is increasing because those notes have higher skiprates and a frequency is being simulated. Additionally, address 12 has a period that is approximately ½ that of address 0.

This is expected because the frequencies of an octave (12 half-steps) should be in a ratio of 1:2.



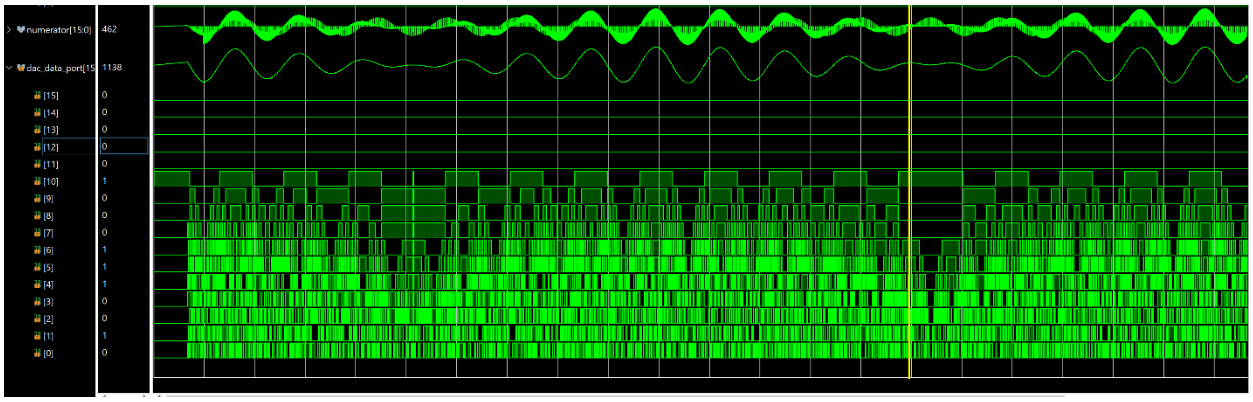
The simulation above is for the phase_register which stores the value of the phase (input time) of each note's sine wave. The phase value goes to the Sine LUT to obtain the yvalues in the yvalue_reg. The phase values are calculated through (skiprates*phasecount). We expect that lower notes (lower addresses) will have longer period because they have lower skiprates. The expected behavior is being seen. Additionally, since address 12 should correspond to the note that is exactly 1 octave above address 0, we should see that there are two whole sine phases pass at address 12 for each phase at address 0. The expected behavior is seen.


velocities...:24][7: 00001001
location_mem -36
yval_port[15:0] 00000000
velocity_port[7:0] 00000000
raddr_vel_port[4: 11000
raddr_yva...rt[15: 00000000
new_data_port 1
result_port[15:0] 00000001
noteaddr[4:0] 00000
phasecount[11:0] 01010101
TCnotes 1
TCphase 0
delay 1
bitshifts 1
phase_reg...4][11 11101010
yvalues_r...4][11:0 00110111
denominator[4:0] 00010
mult_const[3:0] 0001
numerator[15:0] 00000101
multnume...r[19: 00000000
result[11:0] 00100110
dac_data_...rt[15: 00000001

Above is a full math cycle which sums across all 25 notes. The cycle begins when TCnotes goes high. The note address (noteaddr) is set to 0, the numerator and denominator are also set to 0. The velocities register (from datapath) however, has a nonzero velocity in address 0, so on the next rising clock edge, the denominator is incremented by 1, and the y_value from address 0 of the y_values register gets added to the numerator. The noteaddr increments to 1, but the velocity of the note at velocities_reg(1) is 0, so nothing is added to the numerator and denominator. When noteaddr increments to 2, the velocity is once again nonzero ,so denominator is incremented, and the y_value from y_values(2) is added to the numerator. Additionally, the number of bitshifts, which was 0, now becomes 1 to scale for the higher numerator. Since all of the subsequent velocities for the notes are 0, the numerator and denominator are not updated further. When noteaddr reaches 24 (last note), TCnotes goes high, and the result becomes the strength-reduced numerator, while the numerator and denominator themselves are cleared, and the cycle repeats. Timing-wise, this occurs independently of any changes in the velocities register, which helps to simplify and modularize the overall design of the synthesizer.

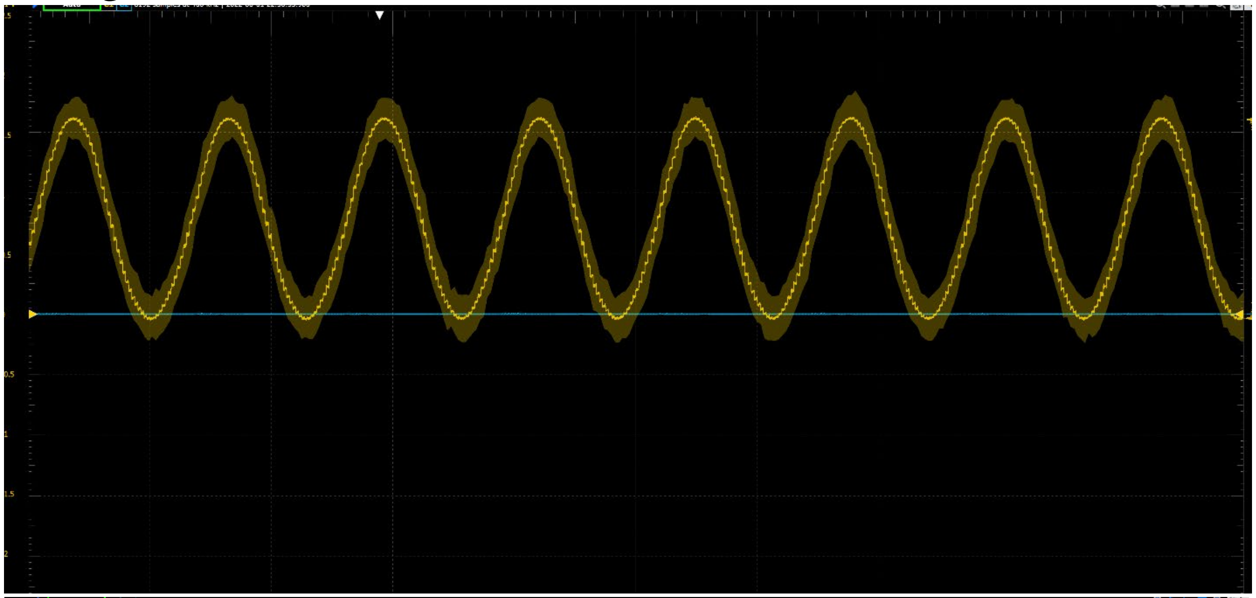
SPI Transmitter

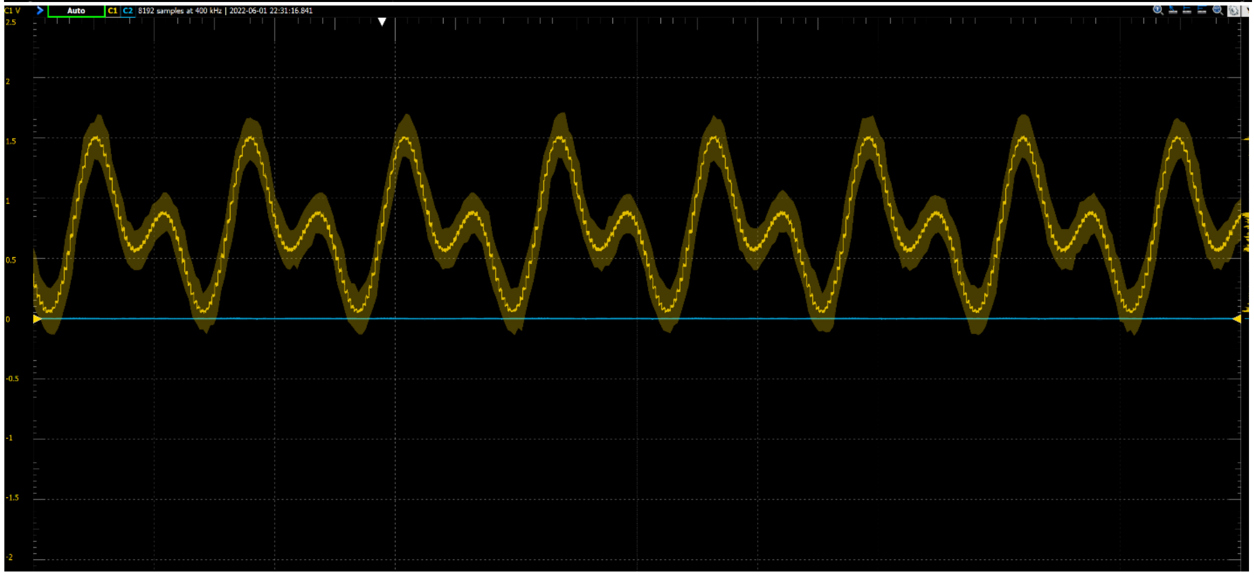
Whole Note Simulation



Shown above is a simulation for the waveform output, with two adjacent keys being played at once. The numerator's shaded appearance is due to its' constantly being recalculated when the math algorithm clears it, then adds to it the first and second notes' y_values. The result, since it only gets the finalized numerator (after all notes have been summed), is stable, and only depends on the actual notes waveforms' interference. A clear interference pattern is seen here—there is the average frequency of the two notes (rapid oscillation, or phase velocity), and the much slower oscillation in amplitude (the "beats" phenomenon caused by two notes with similar frequencies interfering). This slower oscillation is the group velocity. This behavior matches the expected results from physics.

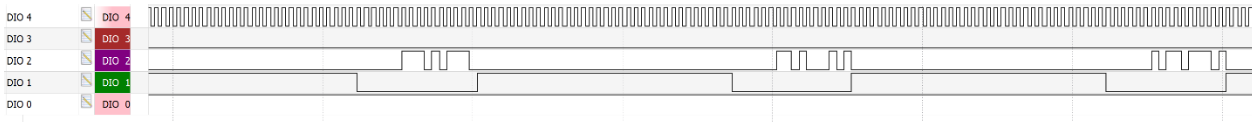
Analog Wave





The analog waves were generated by reading in the analog signal coming out of the DAC through an oscilloscope. The first wave is when a G4 is being pressed on the MIDI keyboard. The second wave shows a G4 and G5 being pressed at the same time. Both waves have amplitudes that are equal even though the second wave is the result of the sum of 2 different values. This confirms that the strength reducing functionality is working properly because the amplitude would be both larger and variable otherwise. The time division for one outlined square on the oscilloscope is 2 ms, and the expected period of G4 is roughly 2.5 ms, so this matches the expectation.

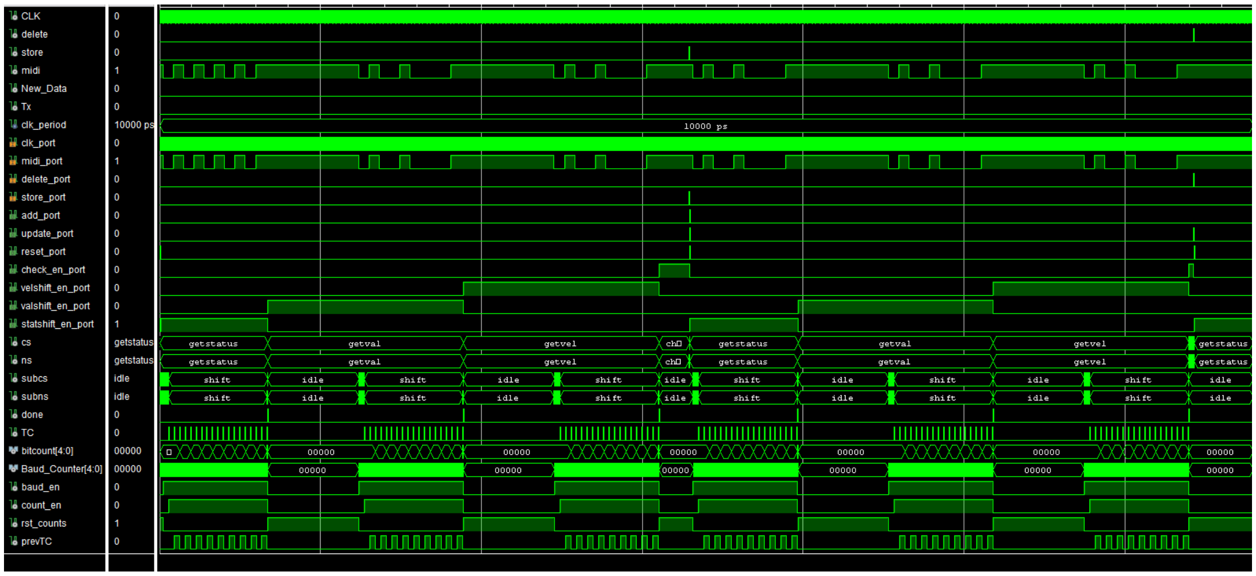
SPI Transmitter



The SPI output data is shown here. Data_valid (DIO 0) goes low when transmission begins, and the dac_data_out is sent out serially (DIO 1). When transmission ends, the data_valid goes back to 1. Transmission occurs every 50 clock cycles, so at a rate of 20 kHz, which is more than enough time to transmit all data.

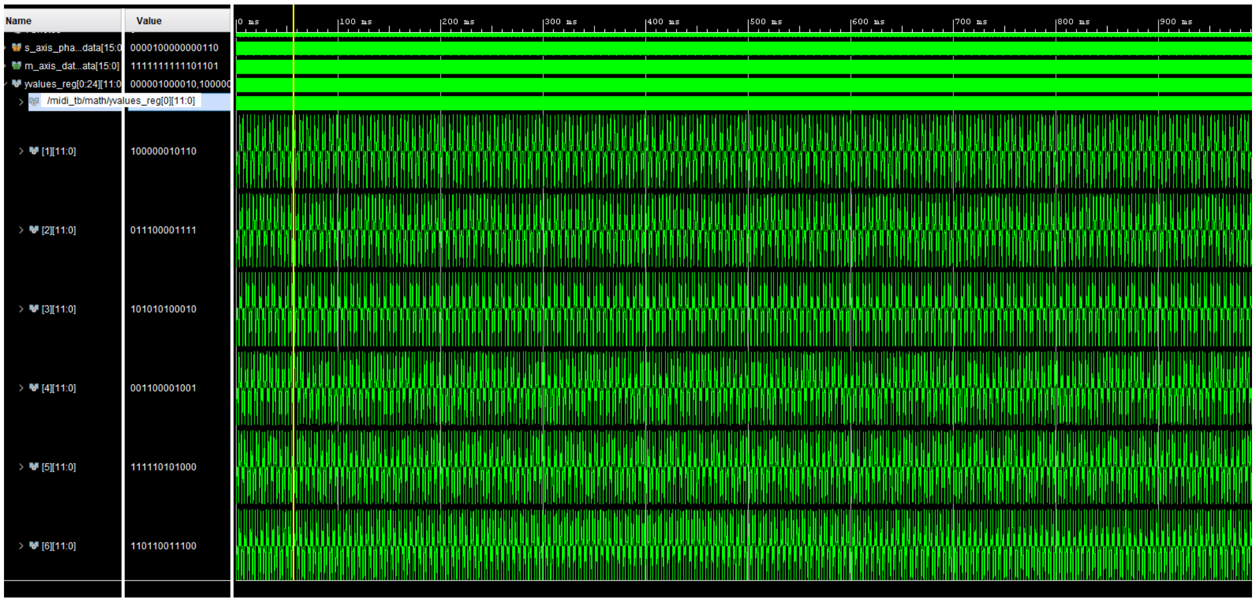
Selected Bugs Encountered (and Fixed)

Shown here are selected bugs encountered during the design process. The first problem was in the controller, where the TC was not offset by ½ a baud period and went high twice as often as it should have, not responding to the prevTC signal, which serves to create the offset.



This was resolved by defining a new TCBaud signal which depended on both TC and prevTC.

Another problem was the generation of waveforms for the notes, where the y_values overflowed and created discontinuous waves for the component notes, as shown below.



This was resulting from the fact that the values from the LUT were signed, but they were being summed like unsigned numbers . The issue was resolved by shifting the entire wave half an amplitude upward (changing the msb of the returned y values 1→ 0 and 0→1) before performing any kind of summation. This and dividing the final wave result by 2 (shifting right once) fixed overflow errors and generated smooth waves.

Design Evaluation

In our implementation we had two separate goals with different criteria to meet each. First, a minimum viable product (MVP). Second, a finished product.

The goal of the MVP was to have the product completed up to the point where the synthesizer had achieved the minimum possible amount of functionality as to where the product could still be considered a synthesizer. We decided that as long as the project was correctly reading in bits using the MIDI protocol and producing pitches after outputting a bitstream to an AMP and DAC it could be considered a MVP. To achieve an MVP, there were only 2 criteria that we were looking for.

I. Pressing a note on the MIDI keyboard produced a pitch through the mono speaker
II. Different notes on the MIDI keyboard produced different pitches

The fully flushed out project had many more criteria. A true synthesizer should not only be able to produce pitches but be able to act as a true musical instrument to meet a basic use case. These were the following criteria we evaluated for.

I. The wave output from the DAC must be nearly a pure wave
II. Up to 10 notes at once can be played for the 10 fingers of a user
III. Pitches are true to the correct frequencies of the played notes
IV. Releasing notes should stop those notes from being played
V. Multiple notes should not increase the volume of the sound
VI. Incorrectly formed MIDI notes should be ignored

In our final product we completed all six of the criteria we evaluated for.

Conclusion, Recommendations

In our final product we were able to fulfill all expectations and created a full fledged working music synthesizer. Given inputs following the MIDI protocol at 31.25kbit/s, our synthesizer can play up to 25 different notes and produce a sound wave for up to 10 notes concurrently. It takes our 1MHz clock 50 cycles to generate a single data point in our wave for all 25 notes. It generates a clean wave and ensures that the overall volume does not exceed a certain limit. Pitches are within __ of the true frequency value in equal temperament and releasing notes stops notes from being played.

We created a MIDI receiver and datapath that parsed through the input bits. We created a Math module that uses a Sine LUT and a register of hard-coded skip rates to generate notes of different frequencies from a single LUT. We created a system that divides down the total resulting amplitude based on the number of notes being played to ensure that the volume does not increase with each successive note. Lastly, we created an SPI transmitter that sends bits to a PmodDA2 DAC to translate our digital resultant wave into an analog signal capable of being output through a mono speaker.

Potential further work on this project would implement different volumes for the keys, and implement different instrumental timbres, such as strings and pianos, by using different wave lookup tables.

Acknowledgements

We would like to thank our Lab instructor Benjamin Livingston “Lifesaver” Dobbins and our Professor Geoffery Luke.

References

Sine LUT Generation

- https://canvas.dartmouth.edu/courses/51947/pages/direct-digital-synthesis-dds-updated

MIDI Data

- https://canvas.dartmouth.edu/courses/51947/pages/prof-hansens-guide-to-midi

PmodDA2 and PmodAD1 Timing

- https://canvas.dartmouth.edu/courses/51947/pages/21-spi-communication

Audio amplifier

- https://digilent.com/reference/pmod/pmodamp2/start

Digital to analog converter

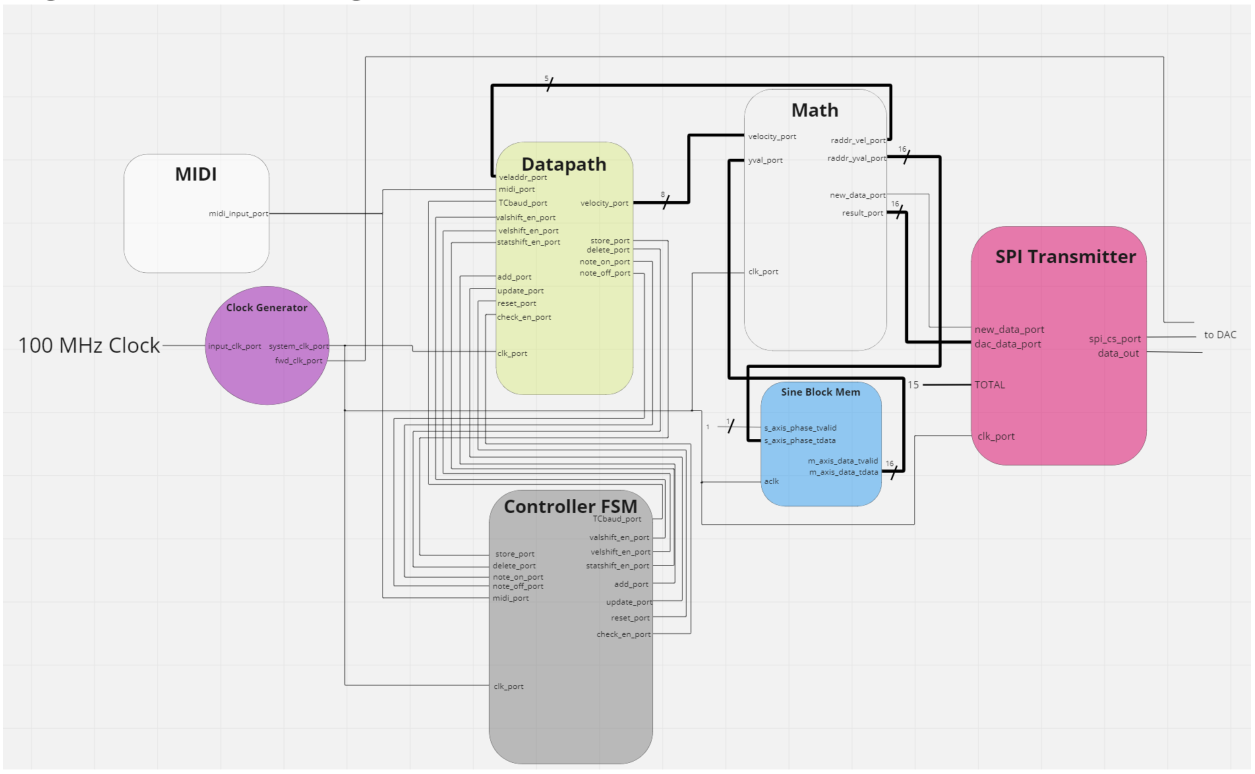
- https://digilent.com/shop/pmod-da2-two-12-bit-d-a-outputs/

Appendices

I. System Level Diagrams

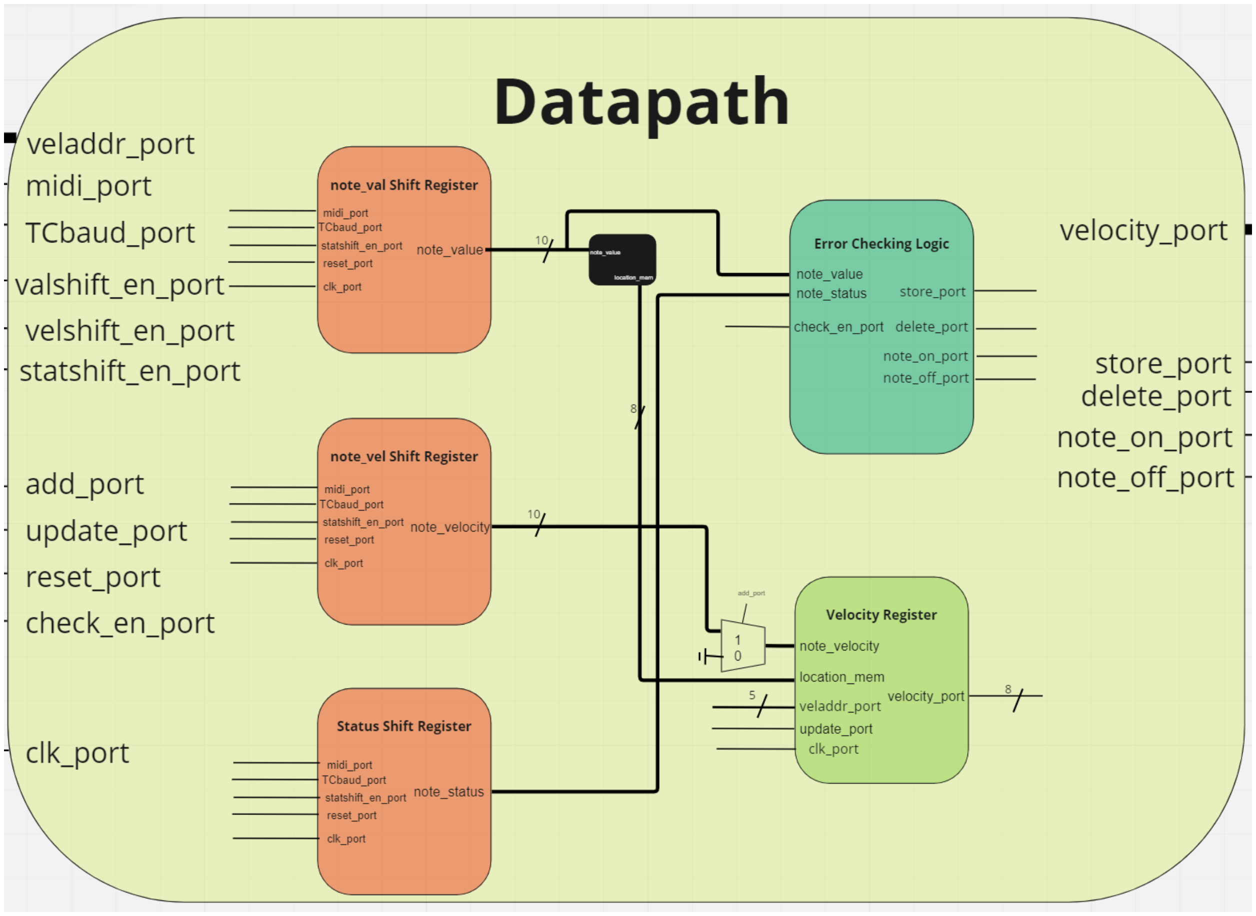
A. Block Diagram

High Level Block Diagram



See https://miro.com/app/board/uXjVOxIIgjo=/. The overall high level block diagram consists of two inputs. A 100MHz clock from the Basys 3, and a bitstream at 31.25kHz from the MIDI instrument. We have a top level synthesizer shell and six additional components. We have a clock divider that initially divides the 100MHz clock to a 1MHz one. The Datapath parses the inputted MIDI bitstream and confirms that the data is intact before storing note velocities in a note velocity register for the Math component. The Math component addresses through all the notes stored in the velocity register and uses the Sine Block Memory and internal skiprates register to calculate value of a sound wave. The resultant values are provided to the SPI transmitter which shifts the values out to the DAC.

Datapath



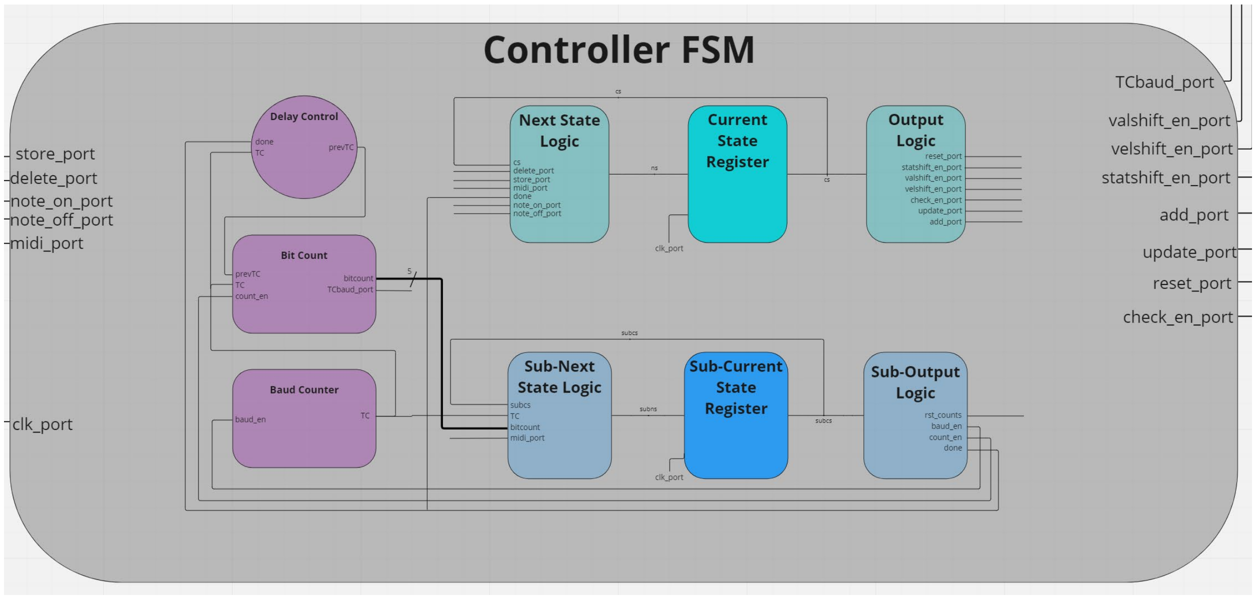
In the diagram above, all ports found on the left are input ports. Apart from *veladdr_port*, *midi_port*, and *clk_port*, other inputs are single bit inputs from the Controller FSM. Veladdr_port is a 5 bit address from the Math component, *midi_port* is a one bit data stream from the MIDI Instrument, and *clk_port* is the system clock generated by the Clock Generator.

Ports on the right are output ports. Apart from the *velocity_port*, all other outputs are single bit outputs to the Controller FSM, while *velocity_port* is a 8 bit velocity to the Math component.

The function of the Datapath is to update a Velocity Register that helps inform the Math component of the currently active notes and their respective velocities. The Math component accesses these values through the *veladdr_port*. The velocity register is filled by parsing the *midi_port* bitstream according to the MIDI Protocol and storing the respective note value, note velocity, and note status/channel data in respective shift registers. After the first packet of 10 bits, and later when the full message arrives through the *midi_port*, the Error

Checking Logic confirms that the message is relevant, and then that it is valid and intact. Depending on the note status, the note velocity is either loaded into the Velocity Register or is reset to zero.

Controller



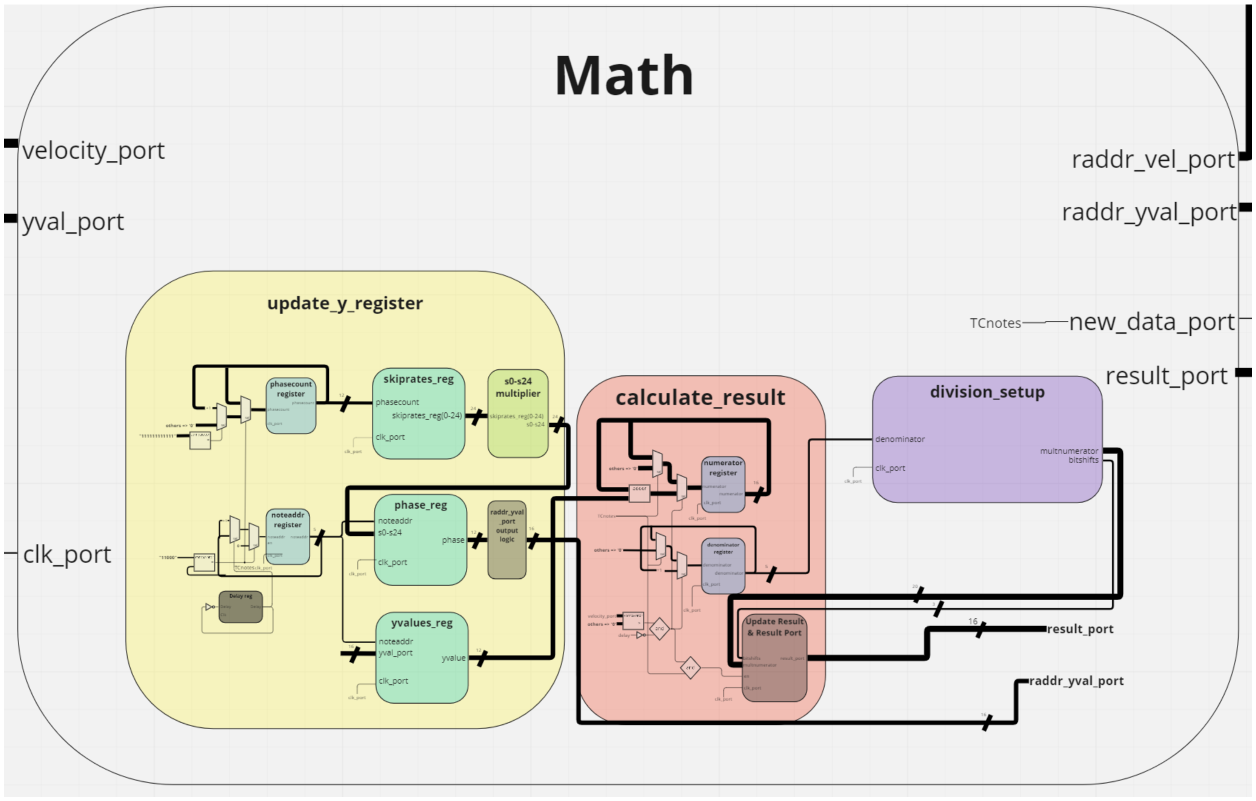
In the diagram above, all ports found on the left are input ports. Apart from *midi_port* and *clk_port*, other inputs are single bit inputs from the Datapath. The *midi_port* is a one bit bitstream output from the MIDI Instrument and *clk_port* is a system clock generated by the Clock Generator.

Ports found on the right are output ports. All output ports are single bit outputs to the Datapath and help control the data flow.

The function of the Controller is to wiggle the single bit output values to control the timing of actions in the Datapath component. The values being output from the Controller is determined by the current state of the Finite State Machine (*see Controller State Machine*). Every clock cycle, the state of the Controller is updated based on values being input from the Datapath. The FSM starts at a *reset* state that clears values stored in the shift registers of the Datapath. Afterwards, the FSM moves through a *getstatus* state, *getval* state, and *getvel* state where the respective bytes are loaded into shift registers. Once values are all loaded in, the Controller moves into a *check* state where the MIDI message is validated. Depending on the result of the validation, the FSM either moves into an *add* state, where the note velocity is loaded into the velocity register, or *clear* state where the velocity register is cleared.

The Controller also includes a Bit Counter, Baud Counter, and Delay Control. The Baud Counter is used to create Baud period of 32 clock cycles to match the MIDI Protocol bitrate. The Bit counter is necessary for the Sub-FSM to keep track of the number of bits that have been shifted in to the current register up to 10 bits ('1' + message byte + '0'). The Delay Control is used to create a half Baud period delay in the Bit Count in order to align the Shift Registers in the Datapath with the middle of the bit rather than at the bit change.

Math

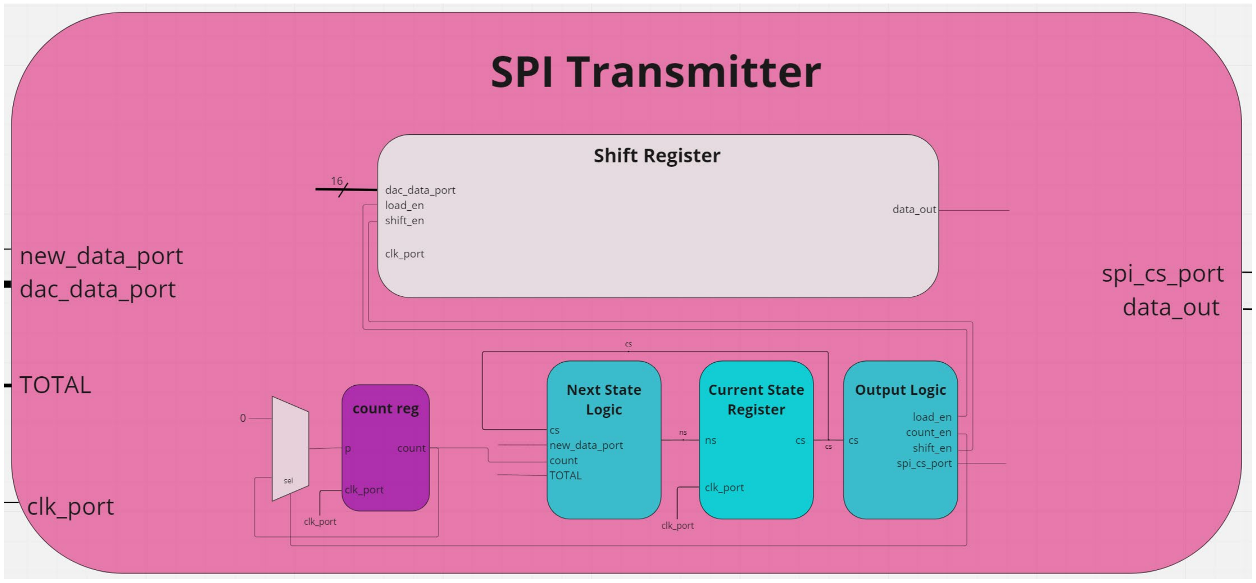


In the diagram above, all ports found on the left are input ports while ports on the right are output ports. The *velocity_port* is an 8 bit data port that contains the value stored at an address raddr_*vel*_port in the Datapath velocity register. The *yval_port* is a 16 bit input from the Sine Block Memory that contains the value at in the Sine LUT at address *raddr_yval_port. New_data_port* is a one bit output to the SPI Transmitter and the *result_port* is a 16 bit output to the SPI Transmitter.

There are three processes within the Math Component. The Update Y Register process keeps track of a current *noteaddr* that iterates from 0-24 and a *phasecount* that iterates from 0-4096. Based on values hardwired in a Skiprate Register and the current point in the period phase determined by *phasecount*, yvalues at the address *noteaddr* are updated in the YValues Register.

The Summing process (originally the Calculate Result Process) generates a numerator value based on the total sum of the yvalues and a denominator value based on the total number of yvalues being calculated.

SPI Transmitter

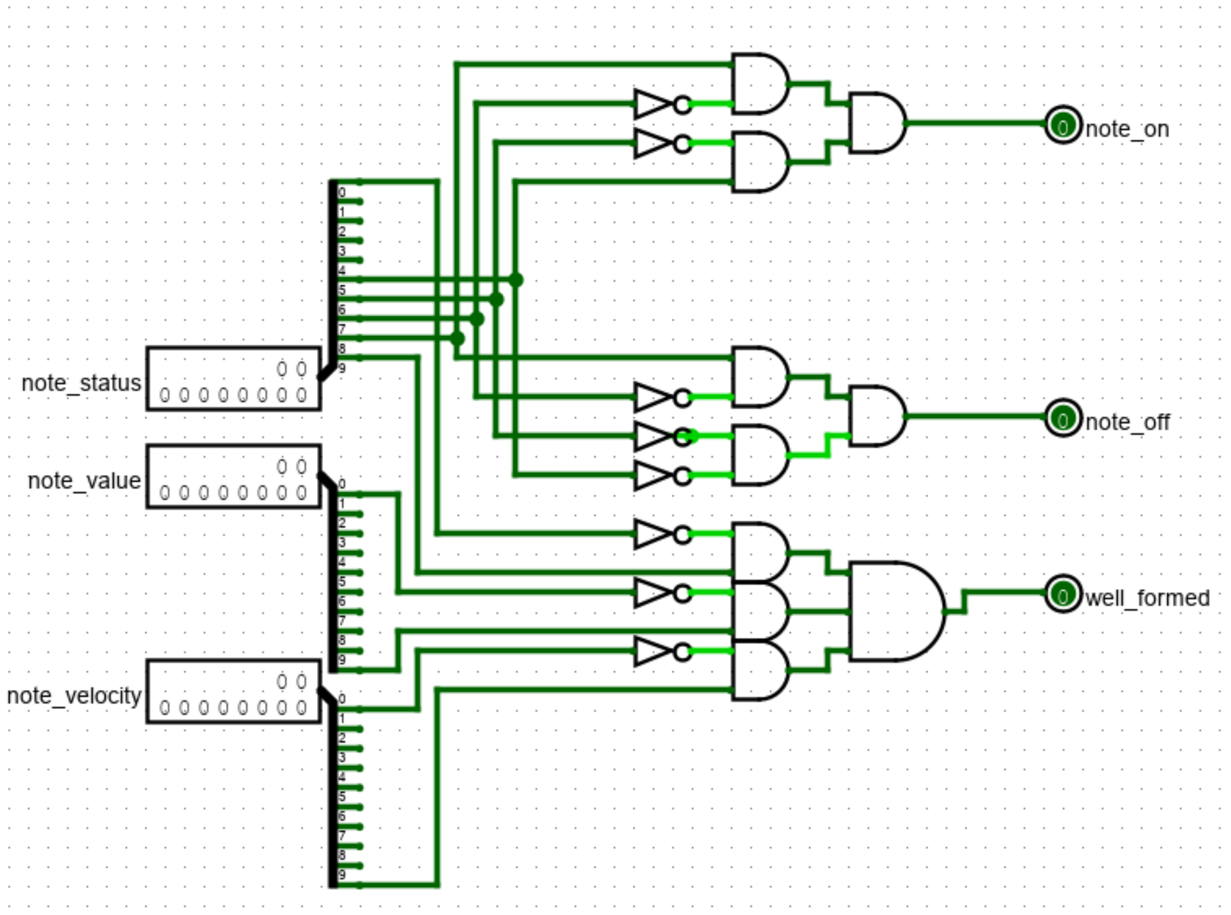


In the diagram above, all ports on the left are input ports. *New_data_port* is a single bit input from the Math Component, while *Dac_data_port* is a 16 bit input from the Math Component. *TOTAL* is a top-level shell generic value set to 15 and the clk_port is a clock generated by the Clock Generator.

Ports on the right are output ports to the DAC. *Spi_cs_port* is a single bit output and *data_out* is a one bit bitstream.

The SPI Transmitter outputs based on the current state of the SPI Transmitter FSM. The Transmitter is either in an *idle* state, *load* state where the *dac_data_port* loads in a value in the overall sound wave, or a *shift* state where the bits in the Shift Register are shifted out MSB first.

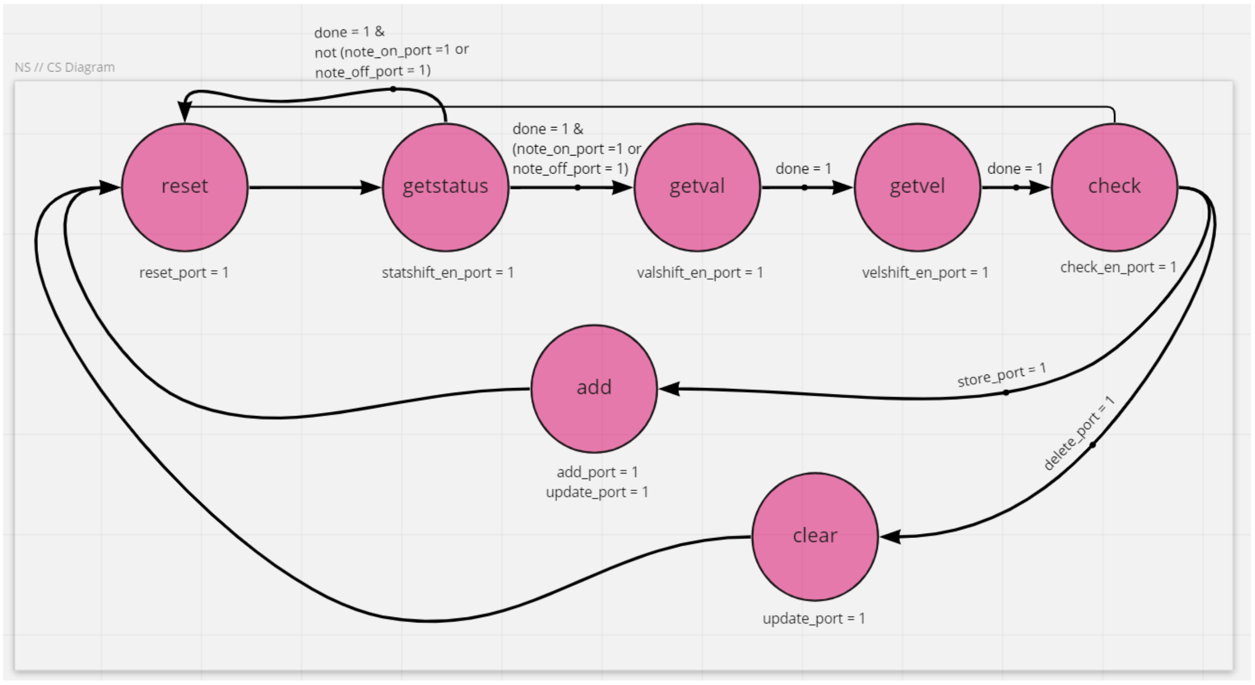
Well Formed Logic



The logic to determine the note_off, note_on, and well_formed signals hardwired in the Datapath. Note_on has a signal of "1001" while note off has a signal of "1000". A well formed signal is one that starts with a '1' bit and ends with a '0' bit and is indifferent to the intermediate bits.

B. Finite State Machines

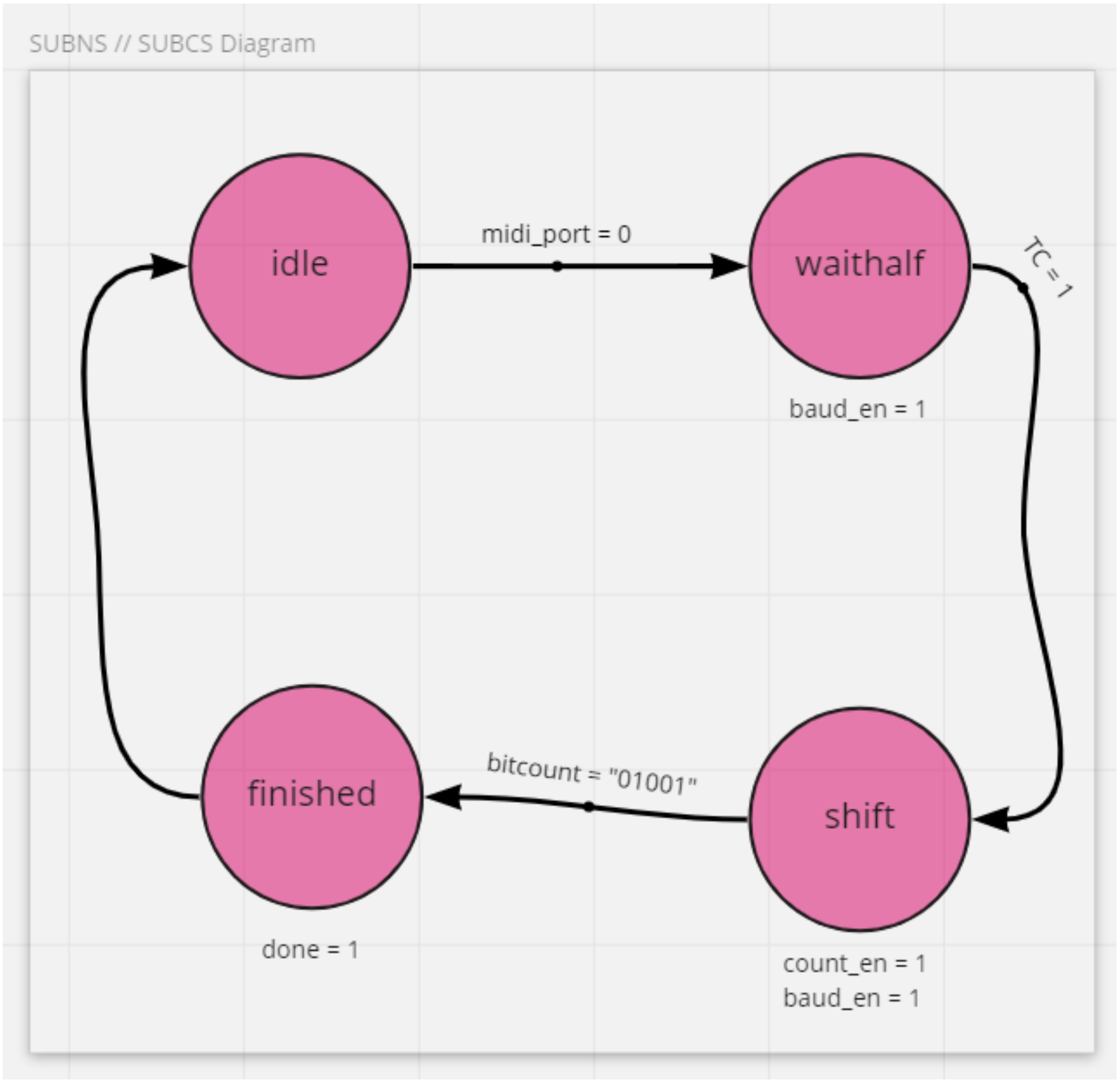
Controller State Machine



The Controller State machine above helps determine which action in the Datapath is taking place. In order, the FSM will load the status, value, then velocity shift registers. After the status is shifted in, the FSM awaits confirmation from the Datapath that the note is either turning on or off. If neither value is high and the whole status has already been shifted in, then the FSM returns to the reset state.

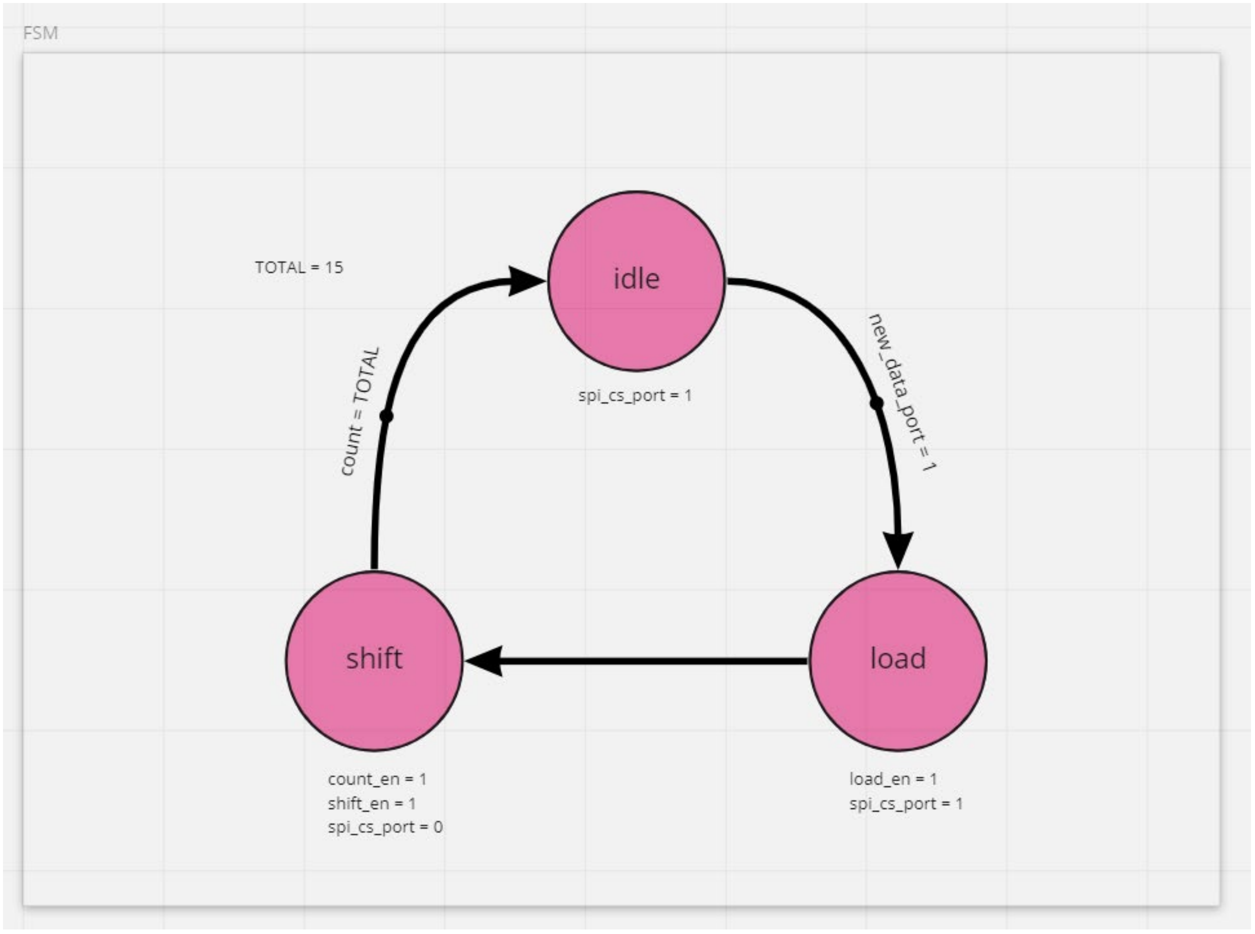
After all shift registers are loaded, logic is performed that checks to ensure that the message is well formed. This is done by ensuring that the starting note each the note value, note status, and note velocity messages begin with a ‘1’ and end with a ‘0’. If the message is not well formed, then neither *store_port* nor *delete_port* will go high and the FSM will return to the *reset* state. Otherwise, the FSM either moves to the *add* state where the velocity register in the Datapath is updated with the new velocity in the shift register, or the FSM moves to the *clear* state where the velocity register is cleared at the current address.

*Contr*oller Sub State Machine



The Controller Sub State Machine is used to regulate the cycling of the main Controller State Machine. This Sub FSM is necessary to delay the moving between the *getstatus*, *getval*, and *getvel* states as 10 baud cycles and a half baud delay must pass before a complete byte is shifted in. This prevents the Main FSM from loading in the next shift register too early before a complete byte has come through. It is a 10 baud cycle delay because each data byte is wrapped with a '1' at the beginning and '0' at the end. The half baud delay is necessary in order to read the values of the incoming stream properly. If the half cycle delay was not included, the bits would be shifted on the exact moment the bits of the MIDI input are changing.

SPI Interface



The SPI Interface was developed with the PmodDA2 datasheet in mind. The PmodDA2 expects the Chip Select input (*spi_cs_port*) to be high when data bits are not being shifted in and for CS to be low when data bits are shifted in. Once the SPI Interface receives new data from the Math Component then the FSM shifts from an *idle* state to the *load* state. Here the 16 bit result from the Math Component is loaded into a Shift Register. After a clock cycle the FSM shifts to a *shift* state where the 16 bits are shifted out.

II. Memory Map—main registers and LUTs

ROM/RAM

Num memory locations	Content
4096 sine wave values (blk_mem_gen_y)	12-bit signed
25 note phases (phase_reg)	12-bit std_logic_vector
25 note y values (y_values_reg)	12-bit std_logic_vector
25 note skip rates (skiprates_reg)	12-bit std_logic_vector
25 note velocities (velocities_reg)	8-bit std_logic_vector
Result (register)	12-bit std_logic_vector
Note_status (shift register)	10-bit std_logic_vector
Note_velocity (shift register)	10-bit std_logic_vector
Note_value (shift register)	10-bit std_logic_vector

III. Math—Performed in Excel

Skip Rates Calculation

In this design, a note's pitch is determined by how quickly the sine wave is sampled through a full cycle. The output waveform value is determined by a simple addition of component notes' waveform values at that point in time (superposition of waves). Since a sum across 25 notes cannot be computed instantaneously in real life, the sum must cycle through all 25 notes, only adding their waveform values if their velocity is >0 (key is depressed). With every cycle, the output wave value is updated. Since the system clock runs at 1 MHz (1 microsecond period), and there is a clock cycle delay for obtaining sine wave values from the LUT, this amounts to a full 25-note summing cycle being completed every (1*2*25 = 50 microseconds, or 20 kHz frequency).

In the sine wave LUT, there are 2^12 = 4096 memory locations, so 4096 sampling points for every cycle. Thus, at a skip rate of 1 (no skipping memory locations), the math algorithm would run through the entire LUT at a rate of 20kHz / 4096 = 4.88Hz. This is a base frequency from which the keyboard note pitches can be calculated.

4.88Hz is an infrasonic (inaudible) tone. To play the lowest note on our keyboard, a C3, whose frequency is 130.81, we assign a skip rate of 27 (130.81/4.88 to the nearest integer). Skipping 27 sampling locations in the LUT has the same effect as sampling 27 times faster, and thus produces an audible tone. The same procedure is repeated for every note to be played on the keyboard, in the following table. Finally, skip rates are converted to binary.

Note	Frequency (Hz)	Skip Rate = frequency/4.88Hz	Rounded Skip Rate	In binary
C4	261.63	53.58182	54	00110110
C#4/Db4	277.18	56.76646	57	00111001
D4	293.66	60.14157	60	00111100
D#4/Eb4	311.13	63.71942	64	01000000
E4	329.63	67.50822	68	01000100
F4	349.23	71.5223	72	01001000
F#4/Gb4	369.99	75.77395	76	01001100
G4	392	80.2816	80	01010000
G#4/Ab4	415.3	85.05344	85	01010101
A4	440	90.112	90	01011010
A#4/Bb4	466.16	95.46957	95	01011111
B4	493.88	101.1466	101	01100101
C5	523.25	107.1616	107	01101011
C#5/Db5	554.37	113.535	114	01110010

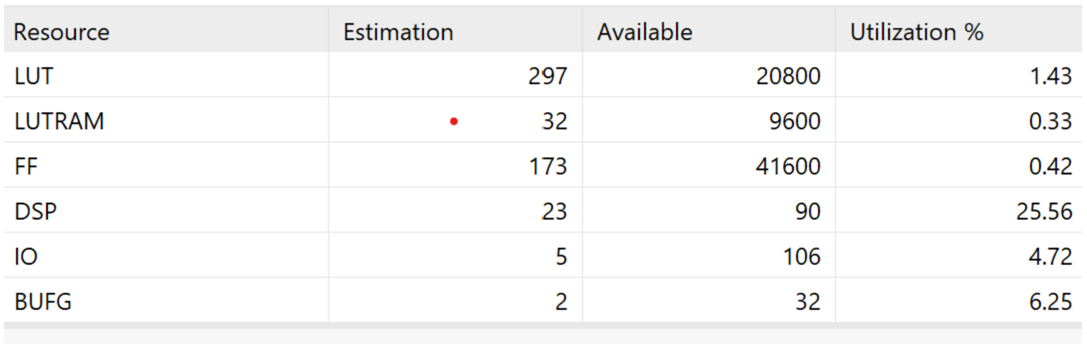
D5	587.33	120.2852	120	01111000
D#5/Eb5	622.25	127.4368	127	01111111
E5	659.25	135.0144	135	10000111
F5	698.46	143.0446	143	10001111
F#5/Gb5	739.99	151.55	152	10011000
G5	783.99	160.5612	161	10100001
G#5/Ab5	830.61	170.1089	170	10101010
A5	880	180.224	180	10110100
A#5/Bb5	932.33	190.9412	191	10111111
B5	987.77	202.2953	202	11001010
C6	1046.5	214.3232	214	11010110

Strength Reduction Calculation

The closest approximation to various fractions is found, with a limit of 6 bitshifts. Instead of dividing a number, strength reduction multiplies it by the constant and shifts bits right (divides by 2^bitshifts).

Fraction	Approximation	Mult. Constant	Bitshifts
1	1	1	0
½	1/2	1	1
1/3	5/16	5	4
¼	¼	1	2
1/5	3/16	3	4
1/6	11/64	11	6
1/7	9/64	9	6
1/8	1/8	1	3
1/9	7/64	7	6
1/10	3/32	3	5
1/11	5/64	5	6
1/12	5/64	5	6
1/13	5/64	5	6
1/14	1/16	1	4
1/15	1/16	1	4
1/16	1/16	1	4

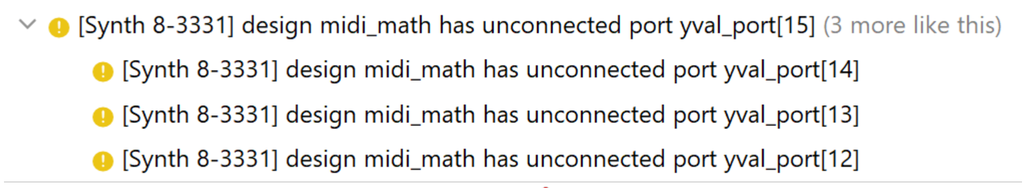
IV. Resource Utilization



Resource	Estimation	Available	Utilization %
LUT	297	20800	1.43
LUTRAM	32	9600	0.33
FF	173	41600	0.42
DSP	23	90	25.56
IO	5	106	4.72
BUFG	2	32	6.25

Resource utilization is shown above. Although this was a large scale project, the FPGA has plenty of computing resources left over.

V. Warnings from synthesis



```
[Synth 8-3331] design midi_math has unconnected port yval_port[15] (3 more like this)
    [Synth 8-3331] design midi_math has unconnected port yval_port[14]
    [Synth 8-3331] design midi_math has unconnected port yval_port[13]
    [Synth 8-3331] design midi_math has unconnected port yval_port[12]
```

Shown are the four warnings of interest, which resulted from truncating signals from the 16-bit y_value lookup table down to 12 bits for the DAC. The table is configured to have the 4 MSBS be 0.

Other warnings are out-of-context module runs, which are benign.