Engs 31 / CoSc 56

Final Project Report

MIDI Controller

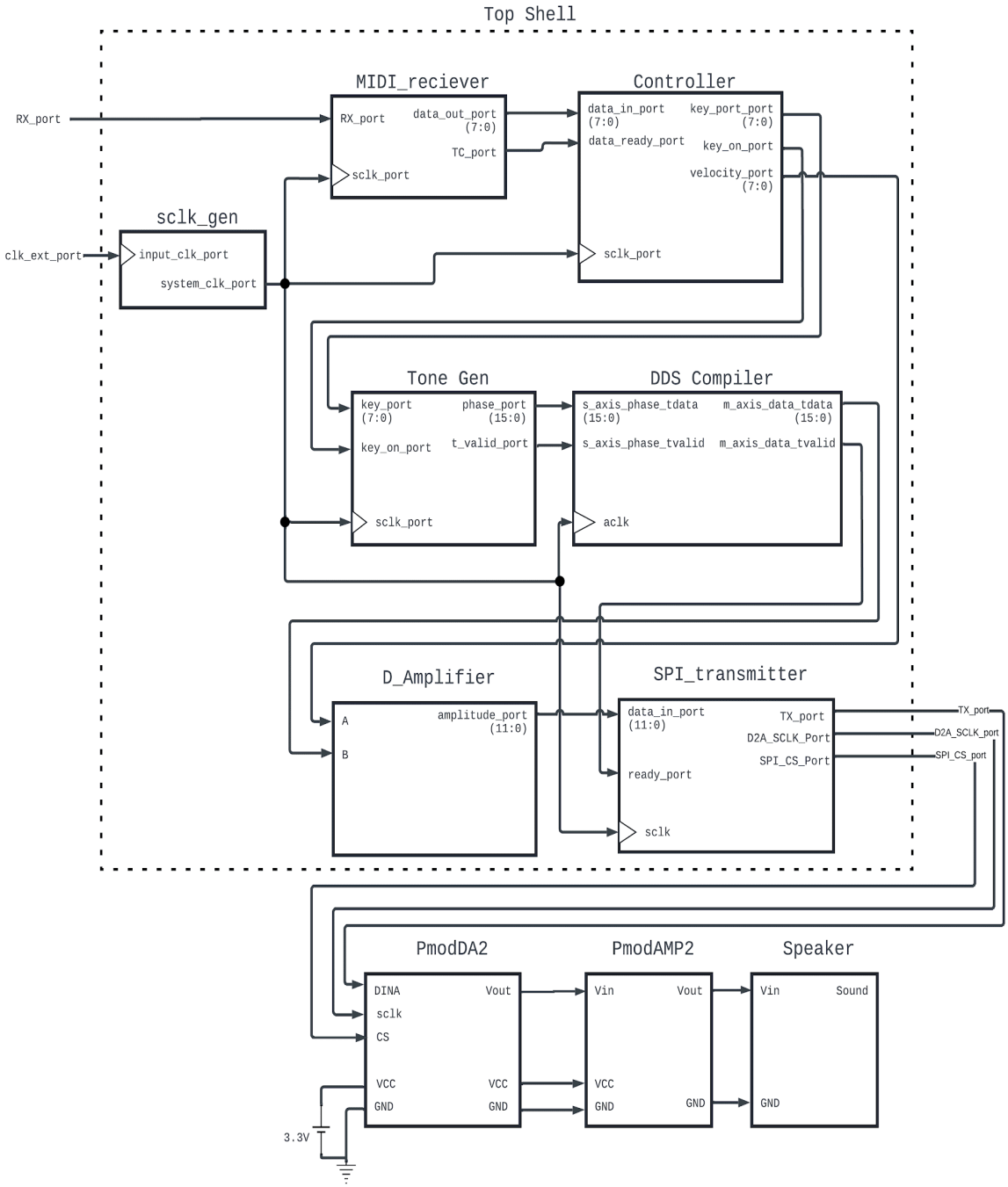
Abstract: The goal of this project is to design a controller that interprets digital input from a MIDI keyboard to produce an audible output. The controller uses the MIDI protocol to communicate with the keyboard and the SPI protocol to send digital signals to a digital-to-analog (D2A) converter. Additionally, a Direct Digital Synthesis (DDS) method is employed to generate a sine wave corresponding to each key pressed. The design was successfully tested and capable of producing sound at a 50 kHz sampling rate, which is within a desirable range for audio quality. Besides sound synthesis, the controller can detect the velocity of key presses, adjusting the volume output accordingly, similar to the dynamic response of a mechanical piano. This project demonstrates a comprehensive digital system, highlighting the integration of MIDI and SPI protocols, the application of DDS for sound generation, and effective testing to ensure high-quality audio output.

Oskar Magnussen, and Vuthy Vey

1. System Overview

The overall goal of this MIDI controller design is to produce sound waves corresponding to the keys pressed on the keyboard, replicating the behavior of a traditional mechanical piano. The intended behavior is to ensure the correct sound frequency is generated for each key. The circuit incorporates a MIDI serial input to process digital signals, which are then sent to a digital-to-analog converter that produces the sound through a speaker. This design ensures accurate frequency generation and seamless conversion from digital input to audible output.

1.1. Top-level Block Diagram



1.2. Description of Ports

Port	In/Out	Description
clk_ext_port	in	100 MHz clock signal from FPGA
RX_port	in	MIDI input signal from the keyboard at 31,250 baud rate
TX_port	out	SPI output signal to D2A
D2A_SCLK_Port	out	An synchronous system clock at 1MHz
SPI_CS_PORT	out	Chip select for the D2A

1.3. Description of Components

Component	Description
sclk_gen	Generates a 1 MHz clock signal from a 100 MHz clock.
MIDI_receiver	Synchronizes the MIDI input and interprets MIDI signals at a 31,250 baud rate, storing data in a 8-bit register
Controller	Handles the key note, its velocity, and the on/off status of each key.
Tone_gen	Produces phase signals over time based on the key note at a sampling rate of 50kHz.
DDS_compiler	Direct digital synthesis, generates digital waveforms by using a phase accumulator and a sine lookup table (LUT) to produce frequency signals
D_amplifier	Amplifies the digital wave by the velocity of the key
SPI_transmitter	Sends 12-bit data using SPI protocol, generates a chip select signal, and provides a new clock signal for synchronous communication to D2A.
Pmod_D2A	Takes in 12-bit data and convert into analog signal from 0-3.3V
Pmod_AMP2	Amplify analog signals with a 3dB or 6dB.

2. Technical Description

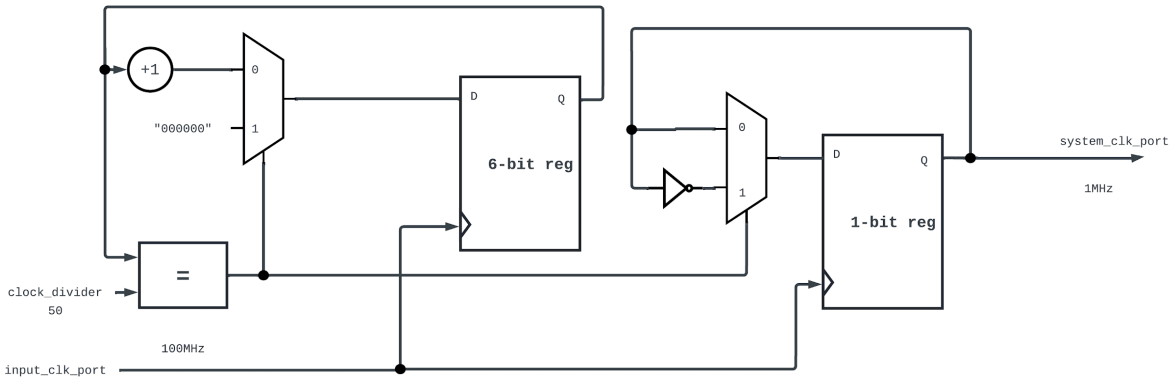
2.1. sclk_gen

This component converts a 100 MHz clock signal to a 1 MHz clock signal while maintaining a 50% duty cycle.

2.1.1. Description of Ports

Port	In/Out	Description
input_clk_port	in	100MHz clock signal
system_clk_port	out	1MHz System Clock

2.1.2. Register Transfer Level (RTL) Diagram



2.2. MIDI Receiver

The MIDI receiver acts similar to a SCI receiver. That is, the component takes a 10-bit input, the start bit (0), the 8 data bits, and the stop bit (1), from the MIDI keyboard, one bit at a time, and stores the data bits in an 8-bit register. The component first synchronizes the input, and then shifts each of the 10 bits through.

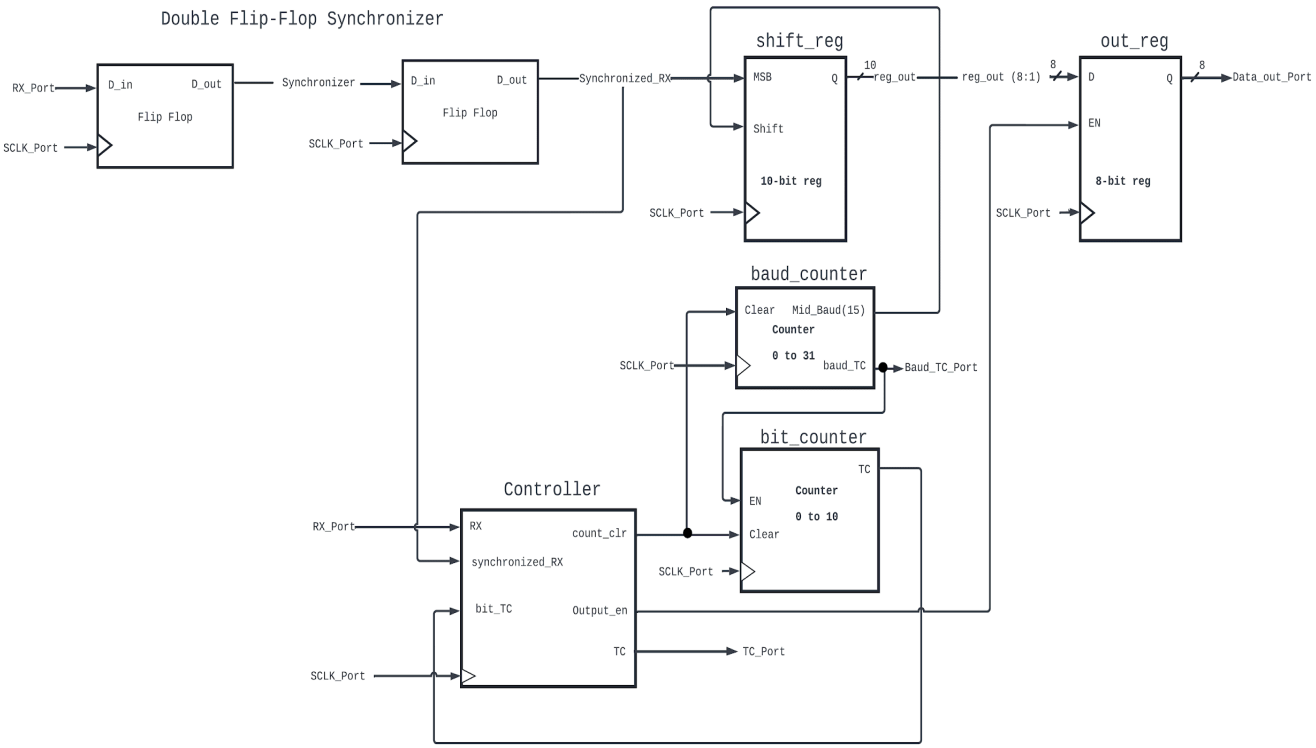
The baud rate for this component is 31,250, meaning that every 32 clock cycles a new bit is shifted into the register. We decided to shift bits in at the middle of the baud count (after 15 clock cycles) to prevent any synchronization errors and make the component run more smoothly.

After all bits have been shifted in, the component outputs a TC signal which can be used to determine when the data is ready to be used.

2.2.1. Description of Ports

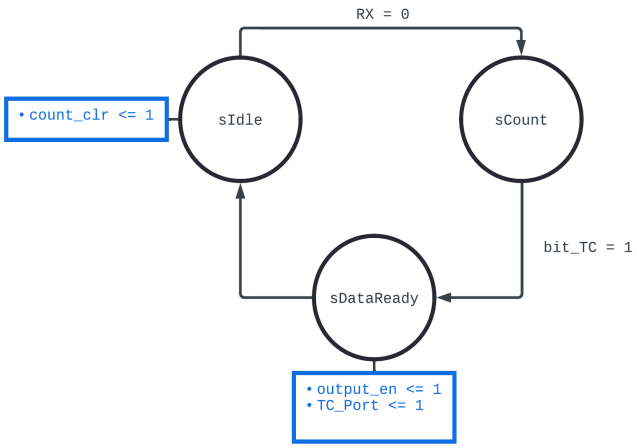
Port	In/Out	Description
sclk_port	in	1MHz system clock
RX_port	in	The current bit that is being shifted in. At idle state, this RX_port is always high. Once this bit becomes low, the component starts processing the input.
Data_out_port	out	The 8-bit data to be used by the MIDI controller. That is, the input after each bit has been synchronized and the start and stop bits have been removed.
Baud_TC_port	out	A bit that goes high after each bit has been shifted out. This output was only used during testing and is not used further in the MIDI controller.
TC_port	out	One bit output that goes high after each 10-bit input has been processed. This can be used to determine whether new data has been received.

2.2.2. Register Transfer Level (RTL) Diagram



2.2.3. Finite State Machine (FSM) Diagram

When the start bit, 0, has been received, we go into the counting state, where each of the 10 bit inputs is shifted into a 10-bit shift register. After all bits have been processed, the start and stop bits are removed and the 8-bit data is sent to the output register. The component is then sent back to the idle state, and all counters and registers are reset.



2.3. Controller

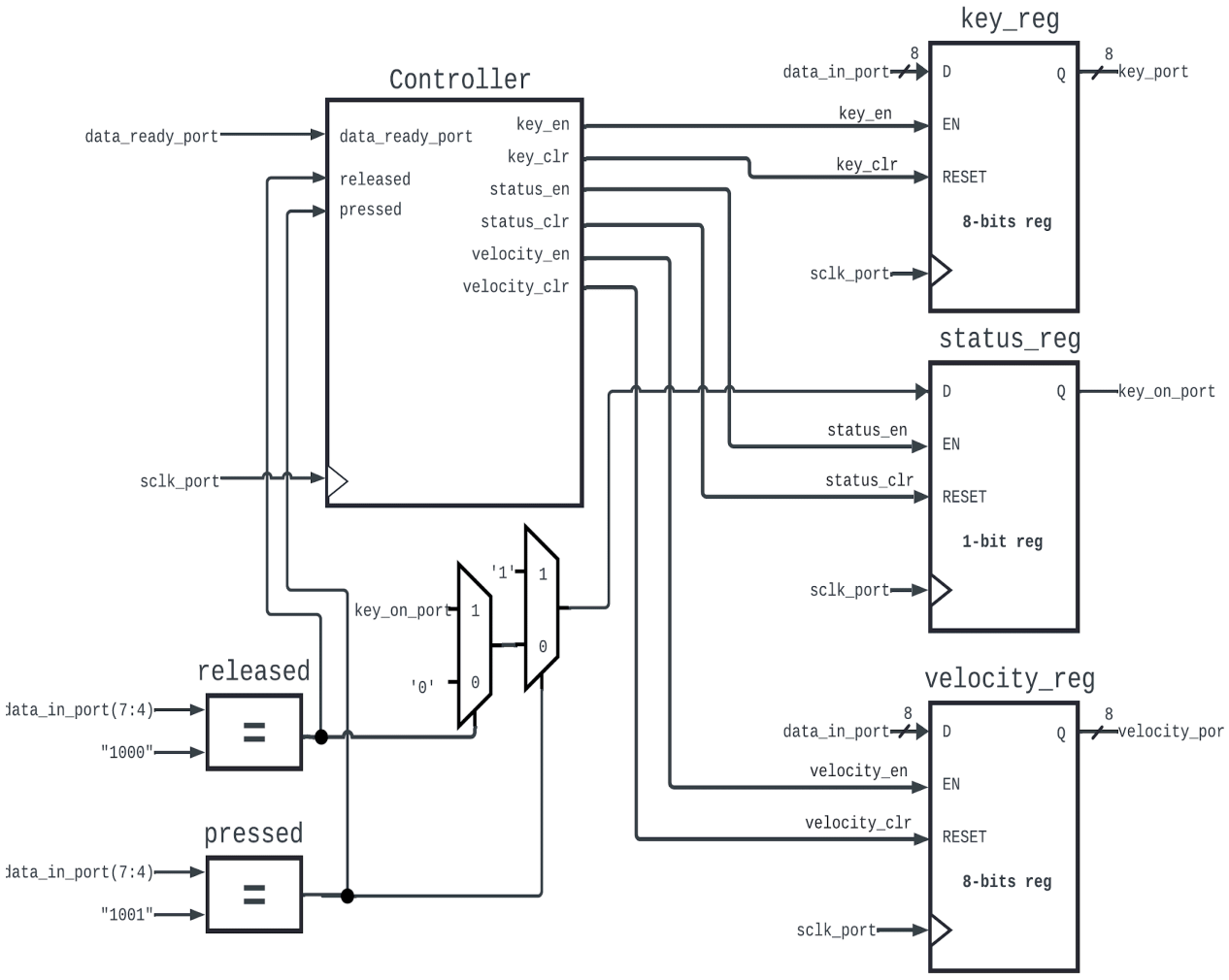
The controller interprets MIDI byte data to determine whether a note is being pressed or released. Additionally, it captures the specific note/key (e.g., C3, D3) being played and the speed at which it was played. Each key press is represented by three bytes of 8-bit MIDI data, with the first byte indicating the key's status (pressed or released), the second byte indicating the specific key, and the third byte indicating the velocity (speed) of the key press.

2.3.1. Description of Ports

Port	In/Out	Description
sclk_port	in	1MHz system clock
data_in_port (7 downto 0)	in	A byte of data from MIDI receiver, containing information about the status of a key, key/note, and the velocity.
data_ready_port	in	A control signal from MIDI Receiver indicating whether there is a new byte of data coming in.
key_port (7 downto 0)	out	A byte representing a key on the MIDI keyboard. For example when C3 is pressed, key_port = "00100100".

`key_on_port`	`out`	Whether the key in key_port is being pressed or released. When `key_on_port = '1'`, then the key is being pressed.
`velocity_port (7 downto 0)`	`out`	An unsigned 8-bit velocity, the larger the velocity port, the harder the key was pressed. In this project, velocity was used to control the output volume.

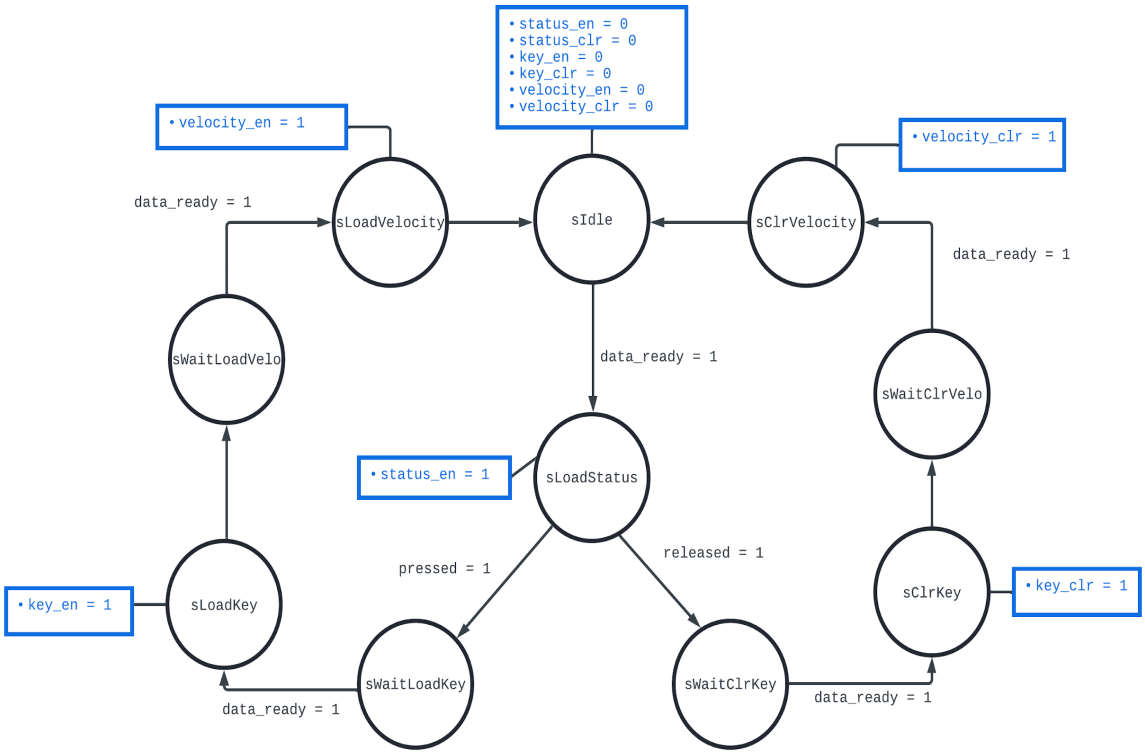
2.3.2. Register Transfer Level (RTL) Diagram



2.3.3. Finite State Machine (FSM) Diagram

The controller operates in two sequences: key press and key release. When a key is pressed, the controller receives a status byte of "10010000," followed by key and velocity bytes. Similarly, when a key is released, a release byte is received ("10000000"), followed by key and velocity bytes.

This controller is designed to play only one key at a time. For example, if C4 is pressed and held, C4 will be played. However, if D4 is pressed while C4 is held, the keyboard will switch to playing D4 instead. Additionally, when C4 is released, D4 will also stop playing.



2.4. Tone Generator

The tone generator produces a phase accumulator for the DDS Compiler. The rate at which the phase accumulates depends on the key pressed on the MIDI keyboard. We let m be the frequency tuning word by which the phase accumulates. To produce a higher-pitched sound, the tone generator requires a larger m than for a lower pitch. Like many DDS components, the equation for m is as follows:

$$F_m = \frac{mFs}{N}$$

Where Fm is a note frequency, N is the resolution of the DDS accumulator, Fs the sampling rate, and m is the frequency tuning word.

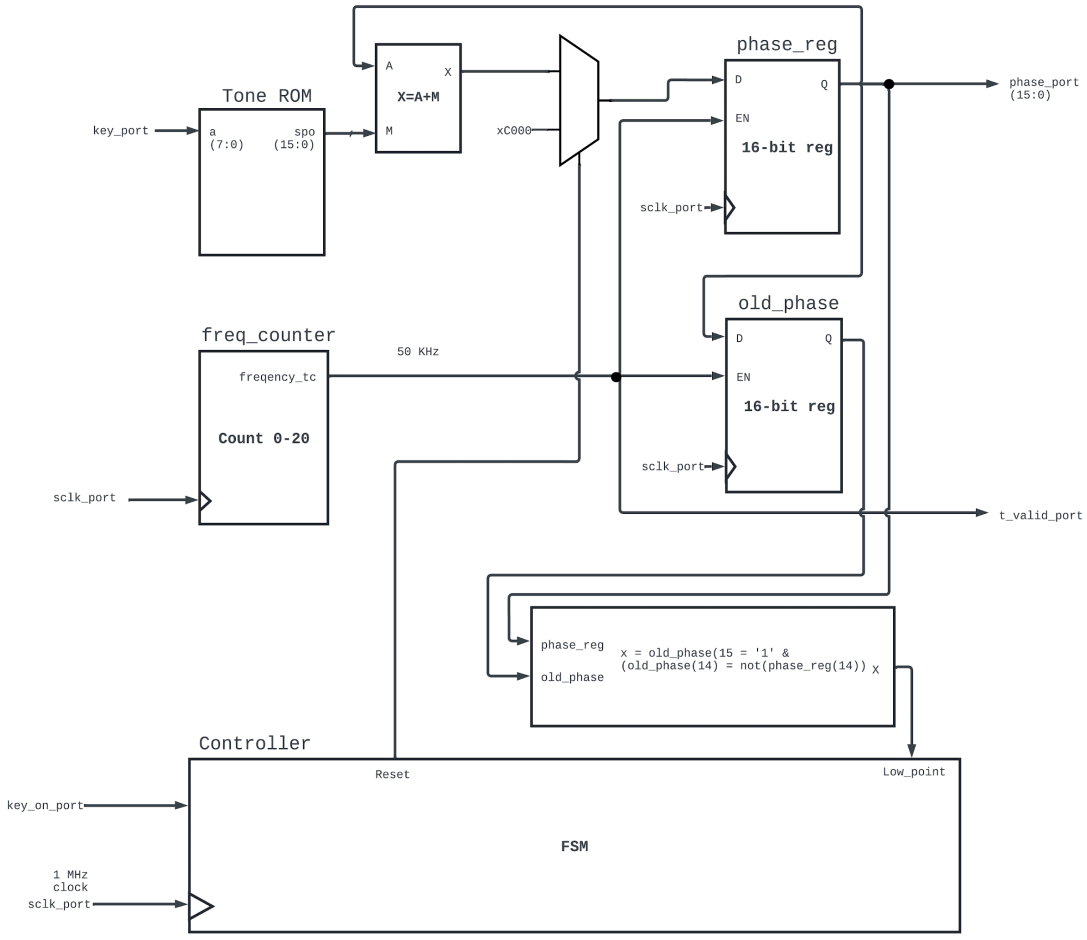
In our case, $N = 2^{16}$ (65536), F_s = 50kHz. For example, C4 has a frequency of 261.626 Hz. We then have F_m = 261.626. With some algebra, we find m for C4 to be approximately 343. This means to generate the tone for C4, we need to accumulate the phase by 343 for the DDS.

This component works in conjunction with the DDS Compiler and its settings, it could have been combined together to a bigger component.

2.4.1. Description of Ports

Port	In/Out	Description
sclk_port	in	1MHz system clock
Key_port (7 downto 0)	in	An 8-bit input representing a key on the MIDI keyboard. For example, when C3 is pressed, key_port = "00100100".
key_on_port	in	A control signal from the controller indicating the status of a key. If key_on_port = '1' then the key in key_port is being pressed.
phase_port (15 downto 0)	out	A 16-bit address of the sine look up table, or phase location of the sine wave. There are 65,536 different possible phase locations.
t_valid_port	out	A control signal indicating that a new phase_port was updated.

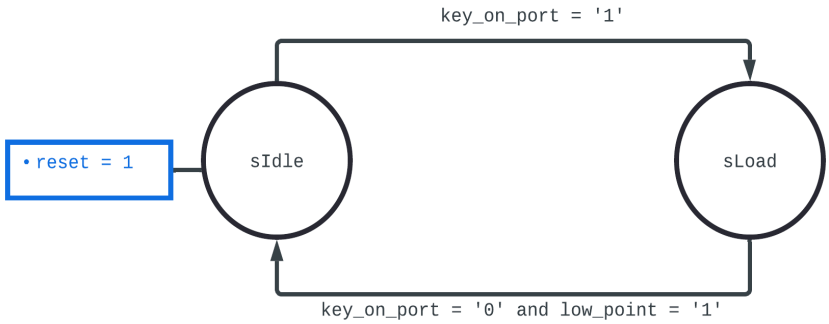
2.4.2. *Register Transfer Level (RTL) Diagram*



2.4.3. *Finite State Machine (FSM) Diagram*

In a simple tone generator and DDS design, we do not need a FSM. However, during our hardware validation, we found that there was a buzzing or clicking noise when a key was released. This is because the tone generator stops when the phase reduces a high magnitude signal, and immediately drops to zero. To solve this problem, we have an FSM machine to smooth out the sound wave when the key is released.

This FSM is staying at sIdle when nothing is playing. However, when a key is pressed, the tone generator switches to the sLoad state and generates the phase. When the key is released, the tone generator still keeps generating the phase until it reaches the lowest point of a sine wave before going back to sIdle. To find the lowest point of the sine wave, since the phase is considered a signed number, all we had to do was find when the most significant bit was high and the second most significant bit changed its value. Since the phase does not always land on exactly the lowest number, there is still some buzzing present in the MIDI controller. However, this method significantly decreases the amount of buzzing in the controller, improving the quality of the sound output.



2.4.4. Description of Memory

This component uses a ROM, look up table (LUT), with a depth of 2^8 or 256, and a width of 2^{16} to store different m constants for different key notes. The constants for different keys were formatted into a .coe file and uploaded to ROM before synthesis. The address is the key representation in 8-bit, and the m is the value in the address in 16-bits.

ROM Block Diagram

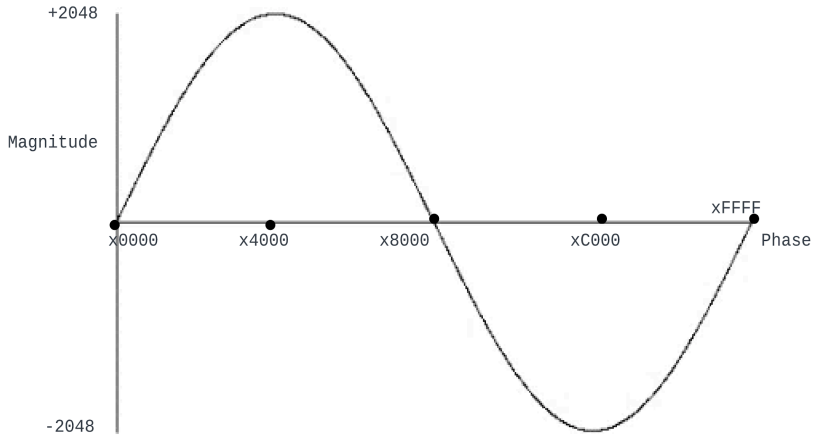
Formatting of the ROM

a	spo (decimal)
0 1 . . 12 13 . . 255	0 0 . . 43 45 . . 0

2.5. DDS Compiler - An IP Component

DDS stands for Direct digital synthesis, and luckily, the FPGA has an IP component for this. This component is coupled with the tone generator component to produce a sound wave from a single sine LUT. The tone generator would give a phase, and the DDS compiler would give the magnitude of the sine wave at that given phase.

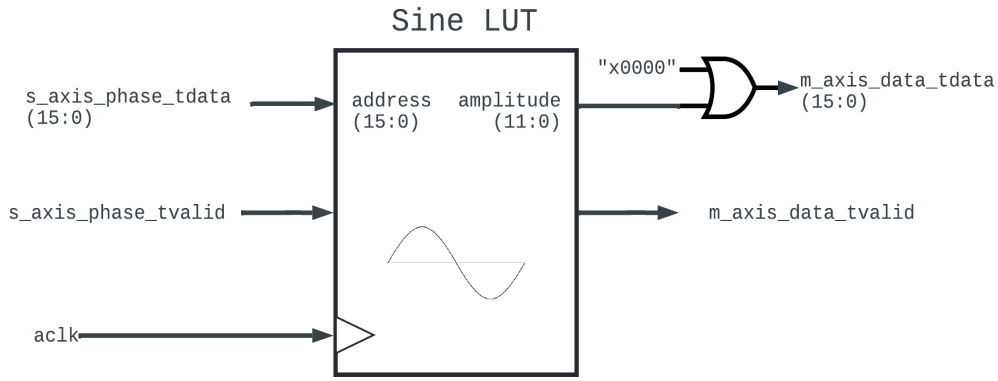
Sine Wave, with phase and magnitude



2.5.1. Description of Ports

Port	In/Out	Description
aclk	in	1MHz system clock
s_axis_phase_tdata (15:0)	in	Address of the sine look up table. The addresses range from 0 to 65536.
s_axis_phase_tvalid	in	A control signal for when to read s_axis_phase_tdata signals.
m_axis_data_tdata (15:0)	out	A 16-bit output, with four leading zeros and 12-bits of data. The 12-bit data represents the magnitude of the sine wave at a given phase. The output is a signed 12-bit number, ranging from -2048 to +2048.
m_axis_data_tvalid	out	A control signal for when m_axis_data_tdata is updated to a new value.

2.5.2. *Register Transfer Level (RTL) Diagram*



2.5.3. *Description of Memory*

This component is a giant look up table, with with a depth of 2^{16} or 65536, and a width of 2^{12}

2.6. *Digital Amplifier*

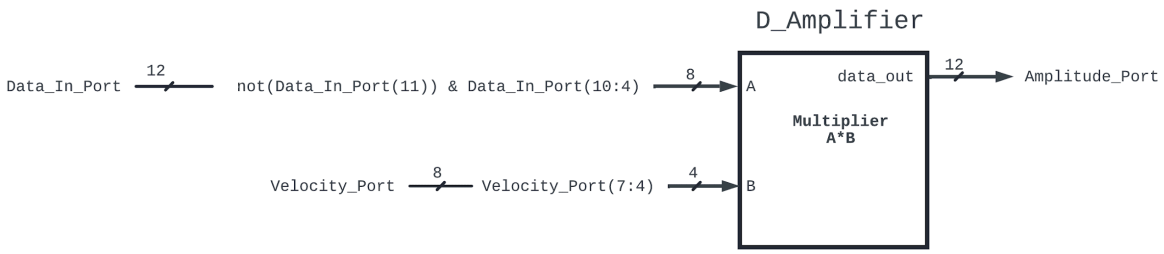
The digital amplifier receives a digital sine input and a velocity input, and amplifies the sine input depending on what the velocity is. The amplifier ignores the 4 least significant bits of each input to prevent the output from having too high of a voltage.

Since the input received is a sine wave, 0 will be the midpoint of the wave. Therefore, to find the minima of the wave and set it as the value 0, we flip the most significant bit. This can be done because the input will be a signed binary number, although the component interprets it as unsigned.

2.6.1. *Description of Ports*

Port	In/Out	Description
data_in_port (11 downto 0)	in	A 12-bit signed sine wave input. That is, the note we want to amplify.
velocity_port (7 downto 0)	in	An 8-bit input for the velocity. That is, the amount we want to amplify the sine wave by.
Amplitude_port (11 downto 0)	out	The sine wave after it has been amplified. That is, the amplified note.

2.6.2. Register Transfer Level (RTL) Diagram



2.7. SPI_transmitter

This component is a simple SPI transmitter. That is, it receives a 12-bit input and shifts out the most significant bit, one bit at a time, until all bits have been shifted out. We added a finite state machine to enable the SPI shift register only when the component receives an input telling it to begin shifting. The state machine will also stop the transmitter when all bits have been shifted out. Since this transmitter will be connected to a D2A converter, and the input it receives is only 12 bits, we added four leading zeros as the most significant bits. This prevents the voltage from reaching its limit as well as converting the data into the correct format for the D2A converter.

While the SPI transmitter is shifting out bits, we set the CS (chip select) signal to low to let the D2A converter know that it can read the input. We also output a one bit signal that goes high when all the input has been shifted out.

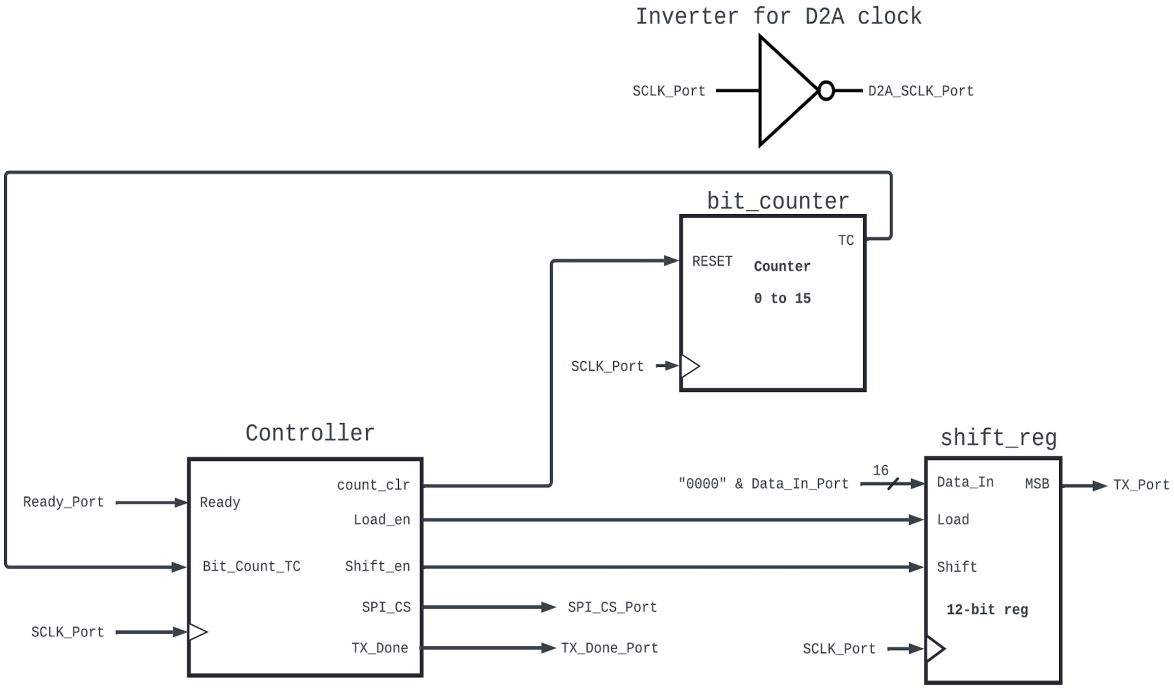
Additionally, because the D2A converter wants to read input at the falling edge of the clock, we generate a clock signal for it by inverting the system clock.

2.7.1. Description of Ports

Port	In/Out	Description
sclk_port	in	1MHz system clock
Data_in_port (11 downto 0)	in	A byte representing a key on the MIDI keyboard. For example when C3 is pressed, key_port = "00100100".
Ready_port	in	A one bit signal telling the component whether a new input has been received or not. When this signal goes high, the component starts shifting out numbers.

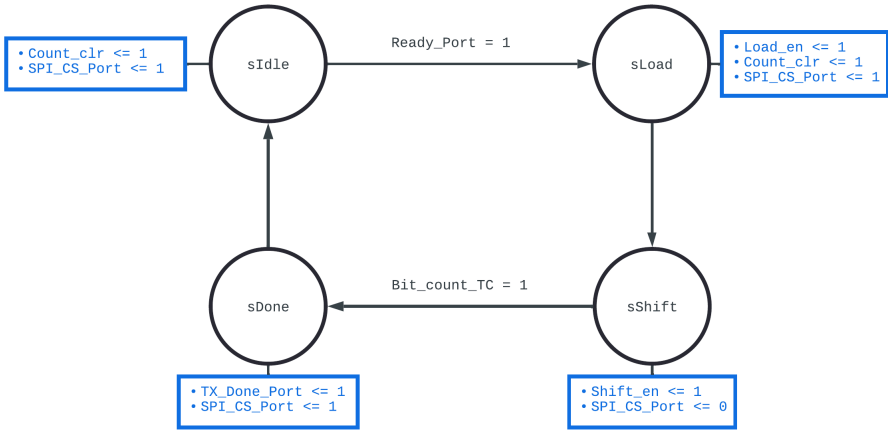
`TX_port`	`out`	The bit currently shifted out, the data. That is, the most significant bit that is left from the input.
`SPI_CS_port`	`out`	A one bit signal that goes low while bits are being shifted out.
`TX_Done_port`	`out`	A one bit signal that goes high when all bits from the input have been shifted out.
`D2A_SCLK_port`	`out`	The clock signal for the D2A converter. That is, the inverted system clock.

2.7.2. Register Transfer Level (RTL) Diagram



2.7.3. Finite State Machine (FSM) Diagram

The state machine waits for the ready signal to go high. Once the ready signal goes high, the state machine goes into the loading state for one clock cycle, which tells the transmitter to load in the new input. The state machine then moves into the shifting state, where each bit is shifted out. In this state, the chip select signal is set to low. When all bits have been shifted out, the state machine goes to the done state for one clock cycle, where it outputs the TX_done signal. Finally, it returns back to the idle state and waits there until the ready signal goes high again.



3. Design Validation

For our design validation, we first made testbenches and behavioral simulations for each component to demonstrate its working functionality.

For the hardware validation, to get the correct input for each component, since each component depends on the previous one, we connected one component at a time to the previous one. We began by connecting the MIDI receiver to the MIDI keyboard, making sure that all outputs from the MIDI receiver behaved appropriately. We then connected the datapath to the MIDI receiver. When the datapath behaved correctly, we connected the tone generator. We implemented each component using this procedure.

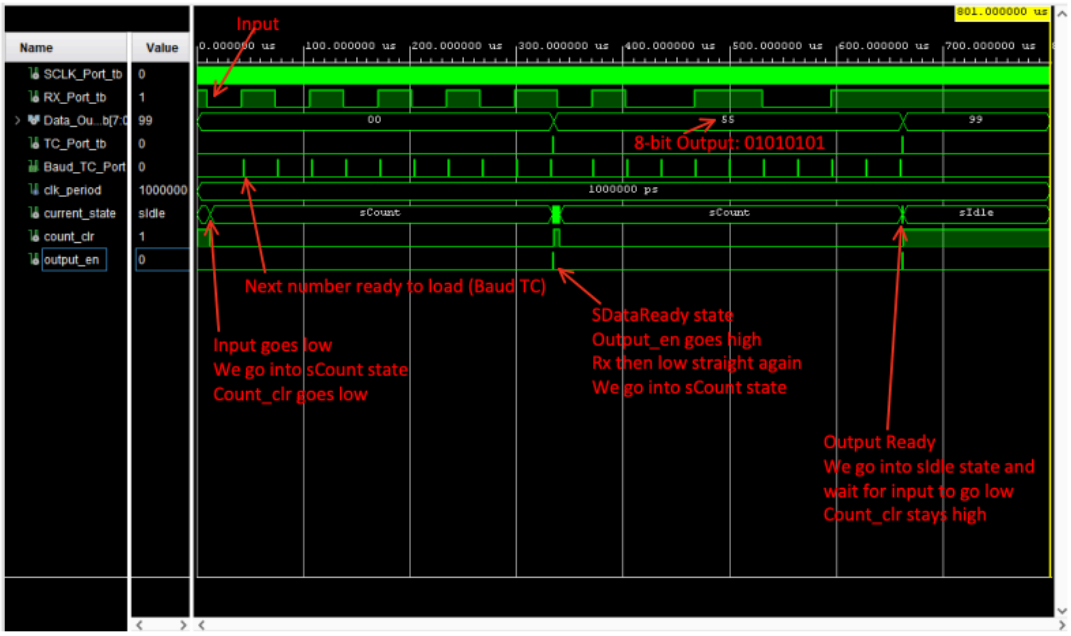
Our only exception to this procedure was the digital amplifier, which we implemented last out of everything. We decided to leave this component until the end since we first wanted to focus on generating the correct keynote with a fixed volume without worrying about the velocity aspect of the controller.

3.1. MIDI_reciever

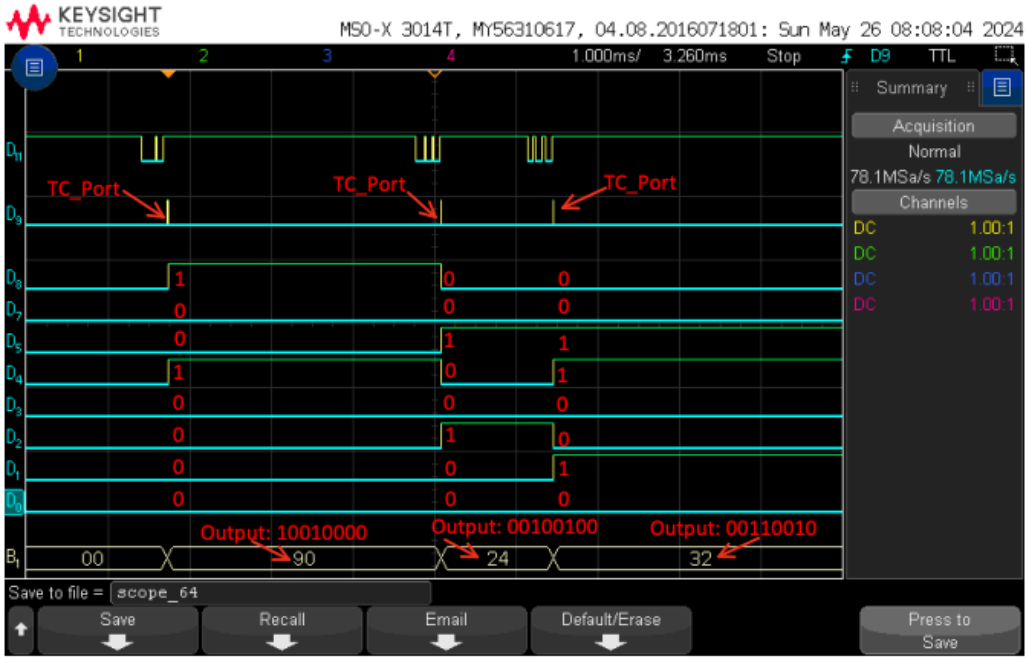
For this component's behavioral simulation, we generated random 10-bit inputs and made sure that the state machine navigated to correct states as well as changing the output enable signal to high when all bits had been shifted in. There were no edge cases that needed to be tested.

For the hardware validation, we connected the receiver to the MIDI keyboard that was used to generate input for the MIDI controller. We then connected the FPGA to an oscilloscope to view the input as well as the 8-bit output and the TC signal. The TC signal should go high when the last bit from the input has been shifted in, which can be seen in the image below. Additionally, we can see that after the TC signal goes high, the output updates to the correct value.

3.1.1. Behavioral Simulations



3.1.2. Hardware Validation



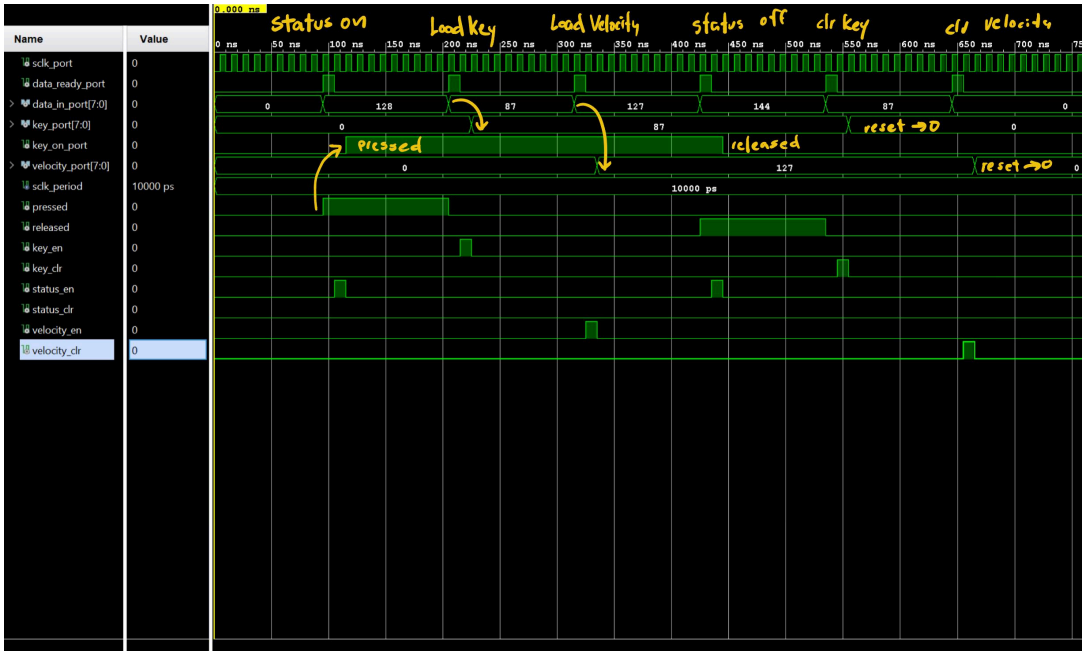
3.2. Controller

For the controller's behavioral simulation, we imitate the MIDI keyboard output signals to see if any of the data is being loaded to the three registers we have that store the key status, key/note, and the velocity.

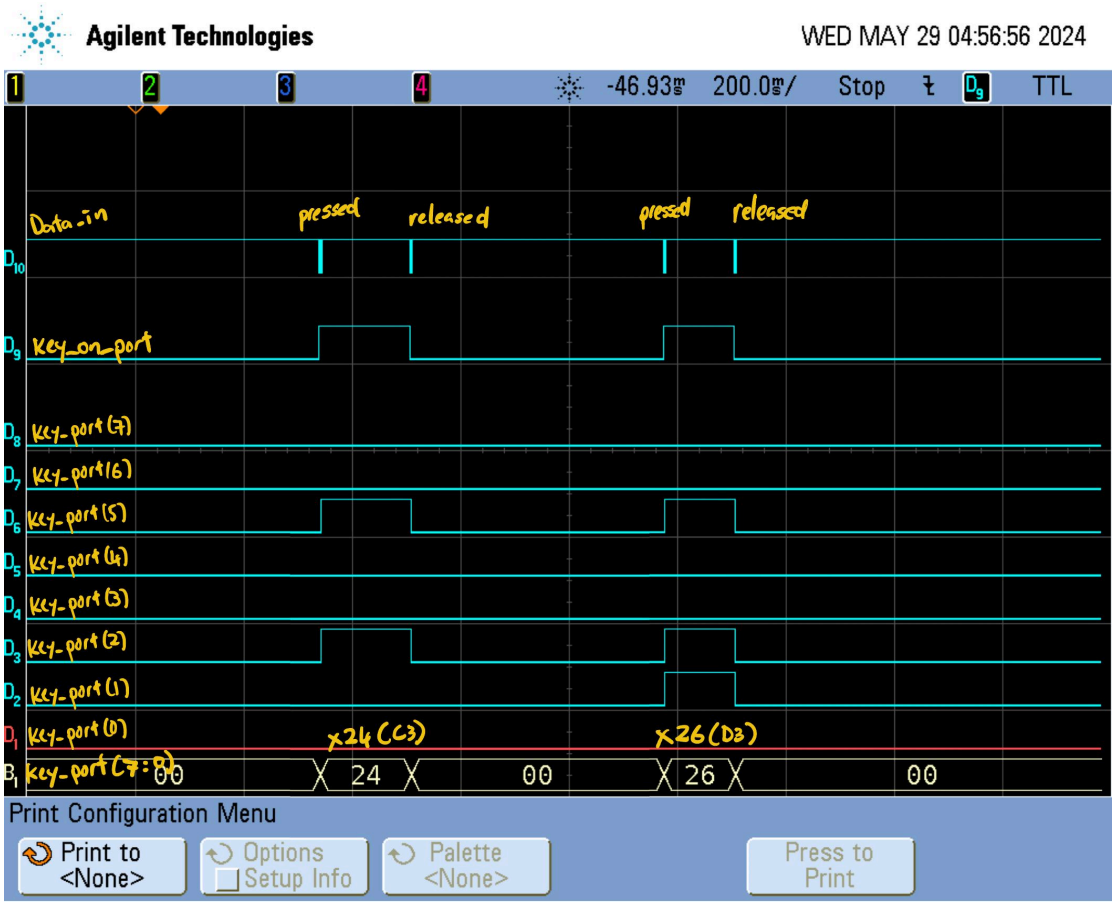
Testing the two possible sequences, when key is pressed, and when key is released. Testing steps are as follows: status on, load key, load velocity, then status off, clear Key, and finally clear velocity.

Similarly, the hardware validation from the logic analyzer on the oscilloscope, we tested a C3 key/note. It was pressed and released, then a D3 key was pressed and released. Note how the key_on_port register changes as it goes through different states.

3.2.1. Behavioral Simulations



3.2.2. Hardware Validation



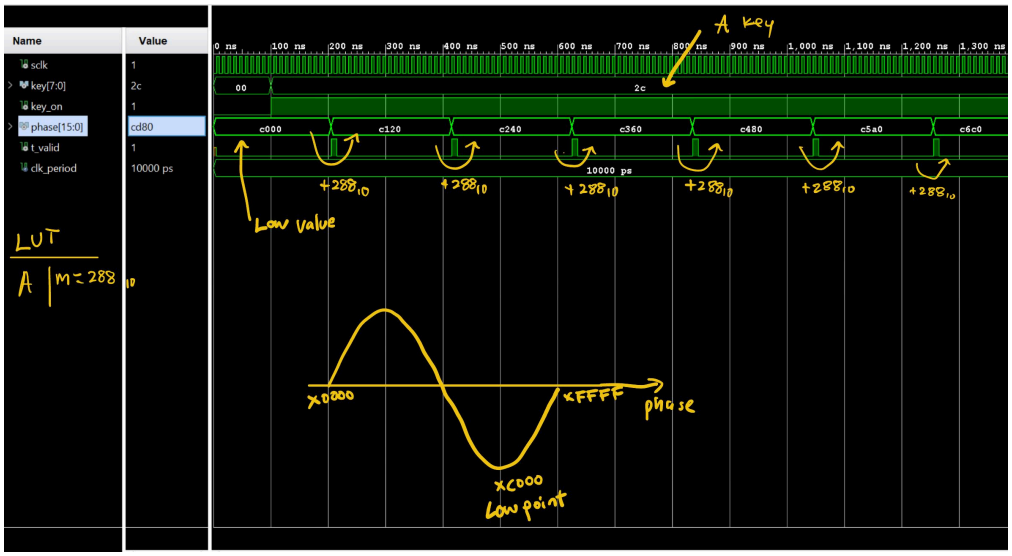
3.3. Tone Generator

To see if the tone generator is running correctly, we simulate a keypress with a note. That note is mapped to a frequency tuner, m, and the phase should be added by m at every sampling rate. When the key is released, the phase should keep accumulating until it reaches a low point, a minimum, of a sine wave.

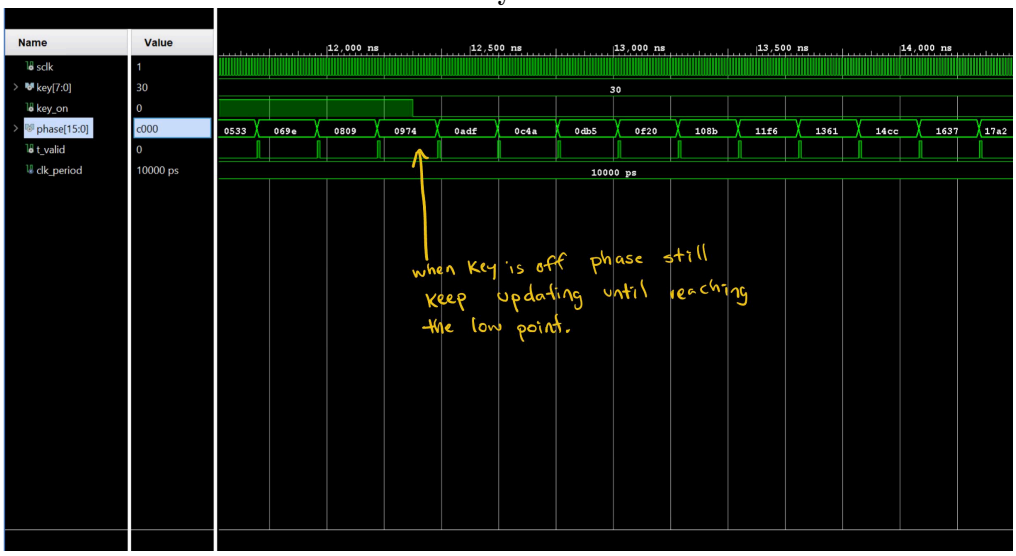
As for the hardware validation, we could only capture a portion of the phase behavior, we expected a consistent change in phase by a constant m for a given key.

3.3.1. Behavioral Simulations

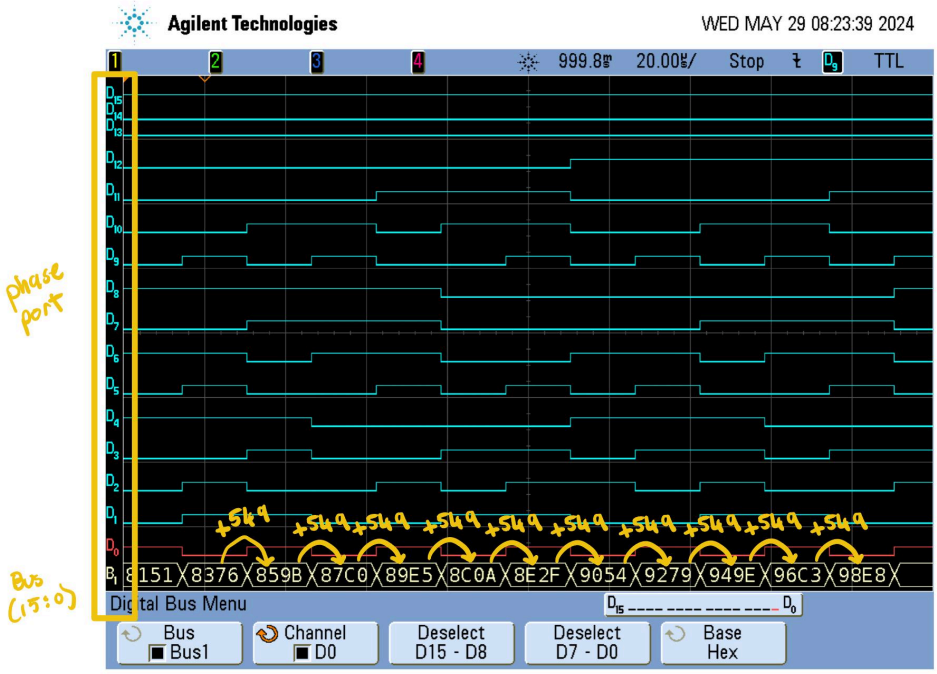
Key On


Name
Value
sclk
1
key[7:0]
2c
key_on
1
phase[15:0]
cd80
t_valid
1
clk_period
10000 ps
A key
00
2c
c000
c120
c240
c360
c480
c5a0
c6c0
10000 ps
+288₁₀
Low value
LUT
A | M = 288₁₀
x0800
xFFFF
phase
xC000
Low point

Key off


Name
Value
sclk
1
key[7:0]
30
key_on
0
phase[15:0]
c000
t_valid
0
clk_period
10000 ps
12,000 ns
12,500 ns
13,000 ns
13,500 ns
14,000 ns
30
0533
069e
0809
0974
0adf
0c4a
0db5
0f20
108b
11f6
1361
14cc
1637
17a2
10000 ps
when key is off phase still keep updating until reaching the low point.

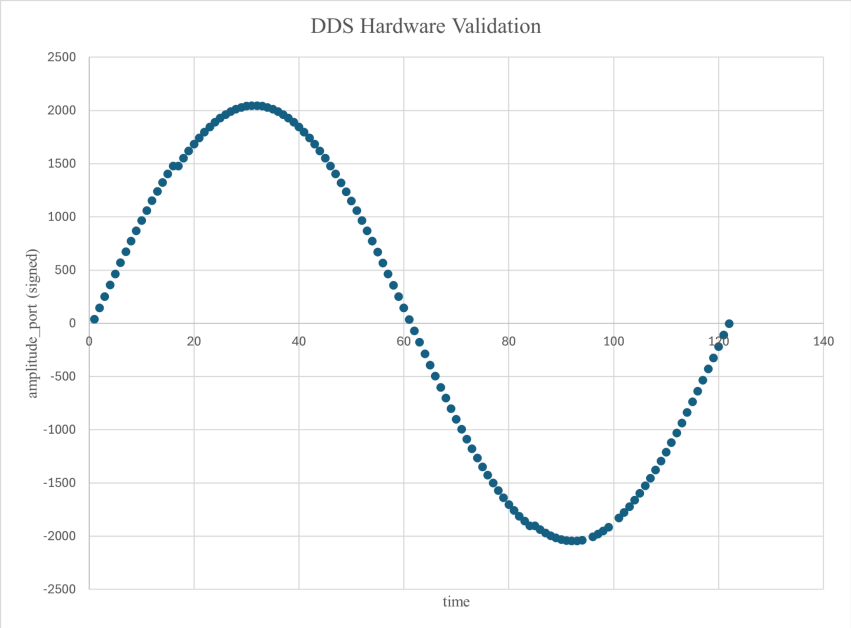
3.3.2. Hardware Validation



3.4. DDS Compiler

Since this is an IP, we did not create behavioral simulation. However, to ensure that this component produces expected values, we hooked the DDS output to the logic analyzer, and plot the magnitude values at different sampling locations. As seen in the graph below, we have produced a single sine wave.

3.4.1. Hardware Validation

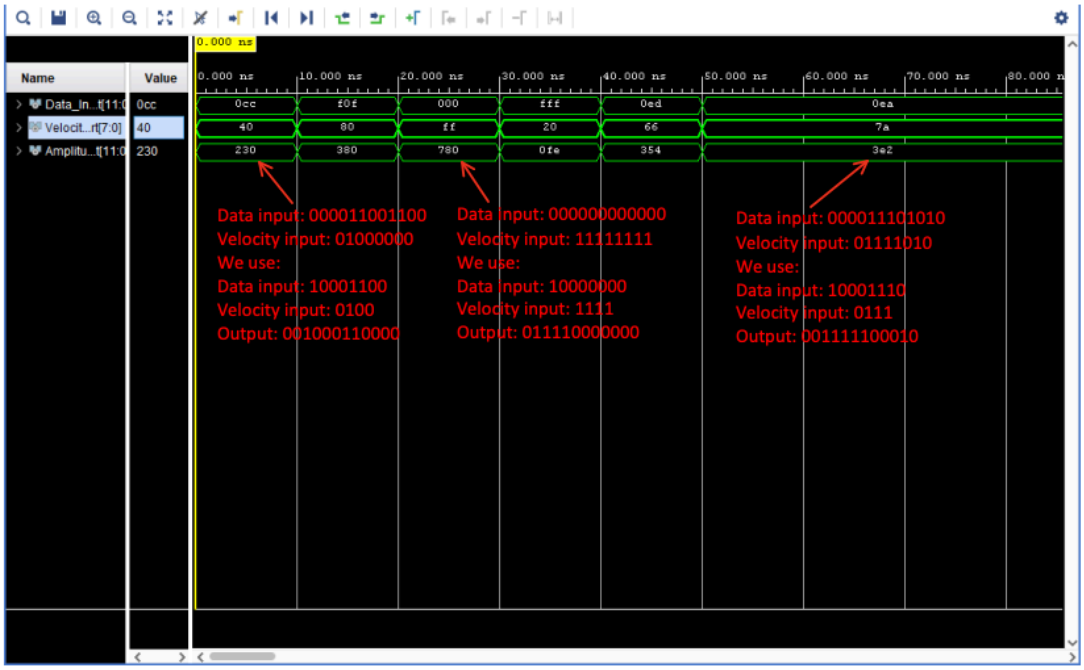


3.5. Digital Amplifier

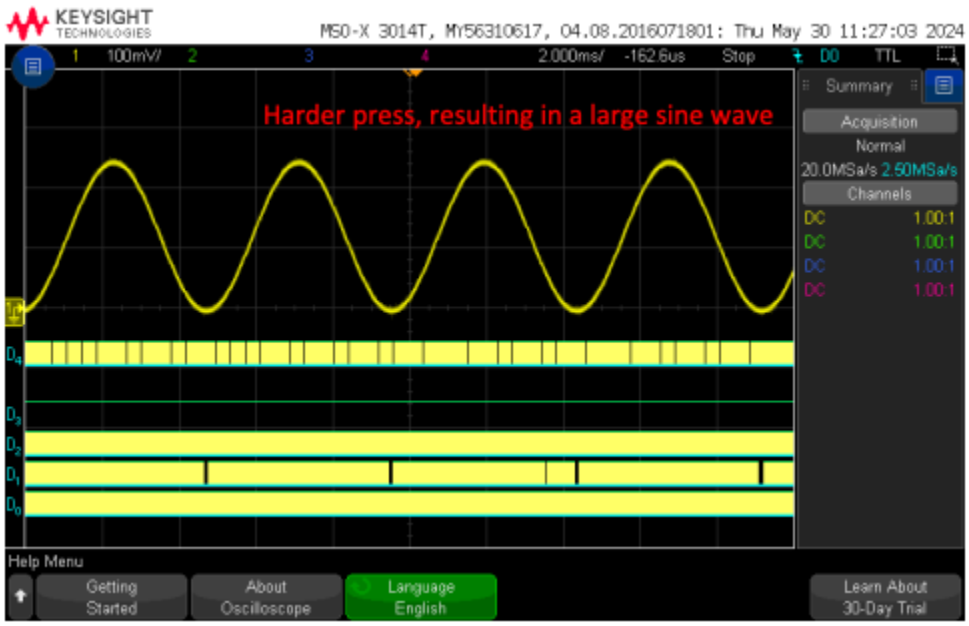
The digital amplifier was the last component implemented into the design. For the behavioral simulation we generated input for both the data and the velocity, where we knew what that output was supposed to be. We then viewed the results of the waveform to determine if they were equivalent to our intended results.

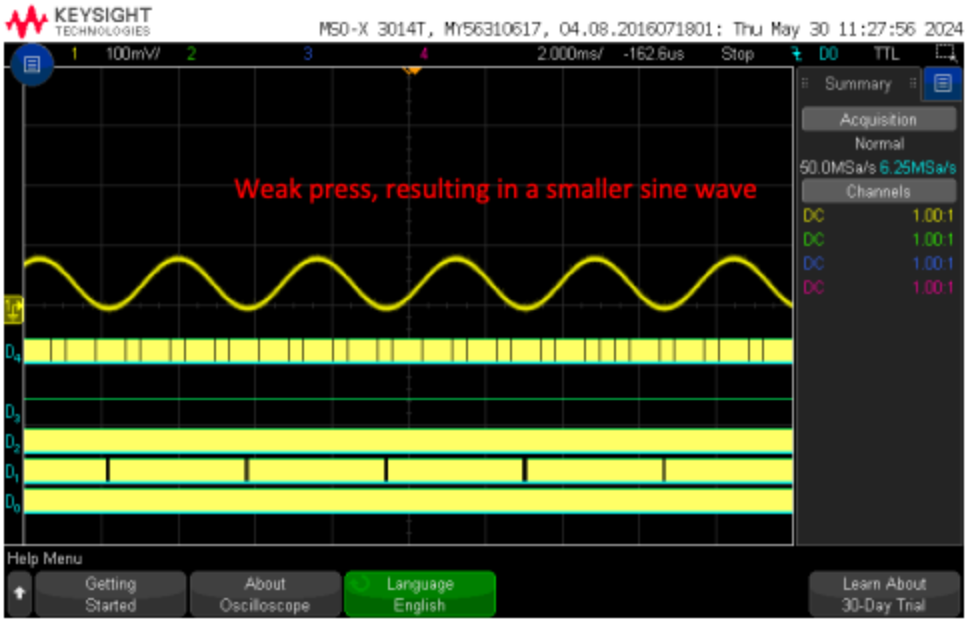
For the hardware validation, we pressed the same not twice, first with considerable force, then with little force. The difference can be seen in the sine waves for each case. The sine wave for the second case (less force) is much smaller, showing that the amplifier works as intended.

3.5.1. Behavioral Simulations



3.5.2. Hardware Validation



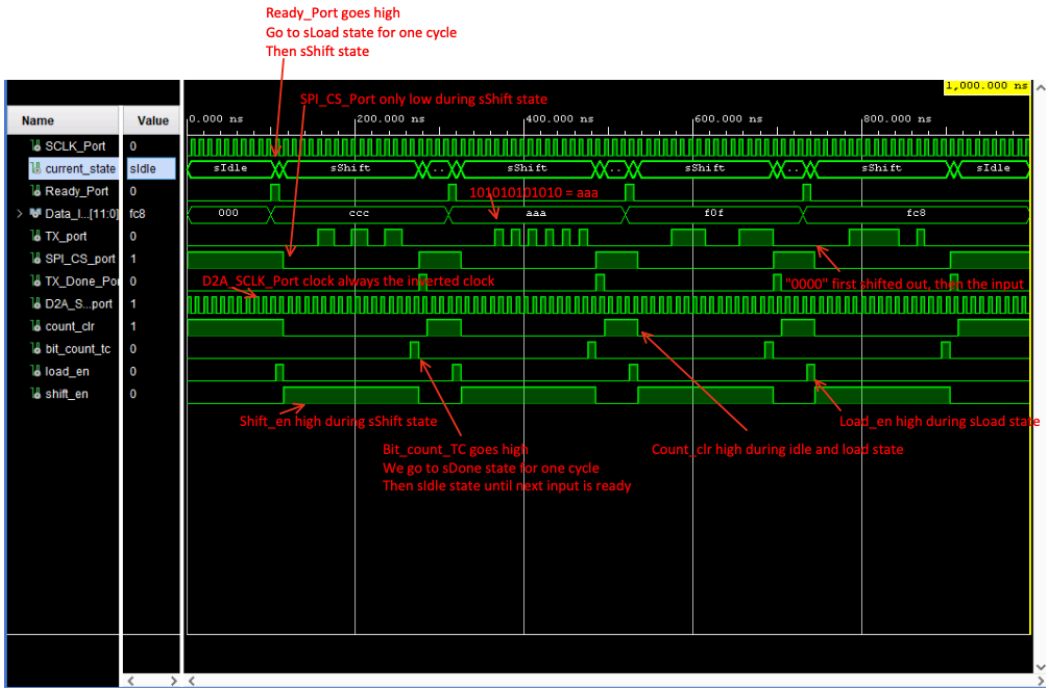


3.6. SPI Transmitter

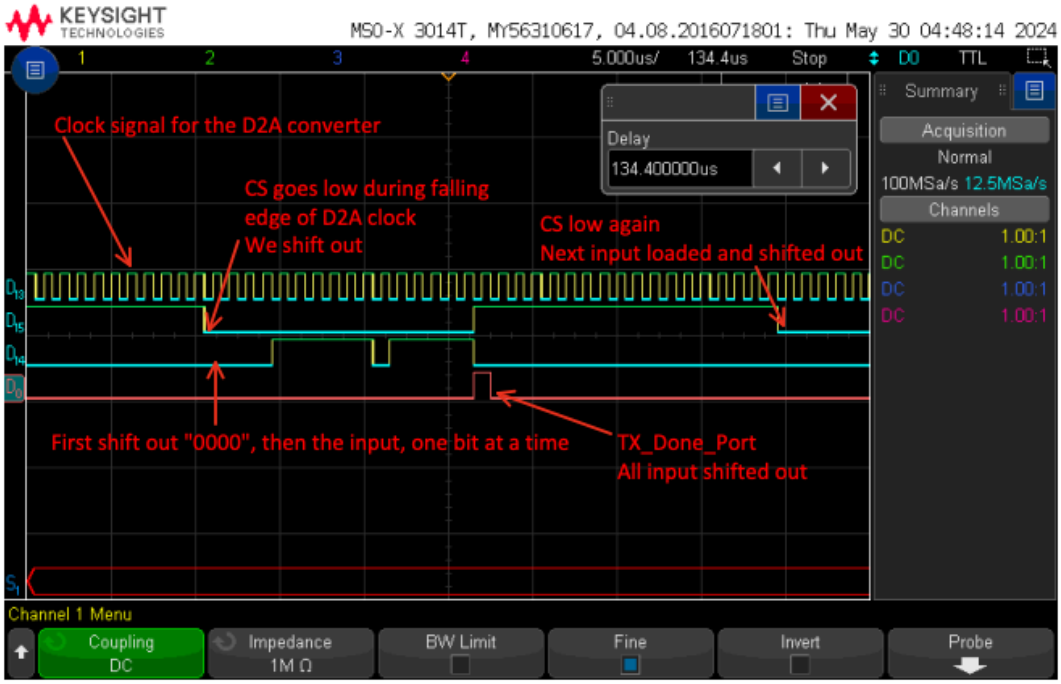
For the behavioral simulation, we focused on making sure that the states were navigated in the correct order. We generated random input and set the ready signal as high. Additionally, we wanted to see that "0000" would be shifted out first, before the input, as well as seeing that the chip select signal would go low during the shifting state. We also wanted to see that the TX port would go high after shifting had been completed. We could see that this was successful if the output was the same as the input, except with four leading zeros in front of it, as well as being shown one bit at a time. Furthermore, we wanted to see that the D2A clock would be equal to the inverted system clock.

For the hardware simulation, we connected this component to the DDS compiler to get correct sine wave input. We wanted to see that when the chip select signal would go low, the output would first show the four leading zeros and then the input. Additionally we wanted to see that chip select would go low during the falling edge of the D2A clock, and that TX port would go high when the all bits had been shifted out.

3.6.1. Behavioral Simulations



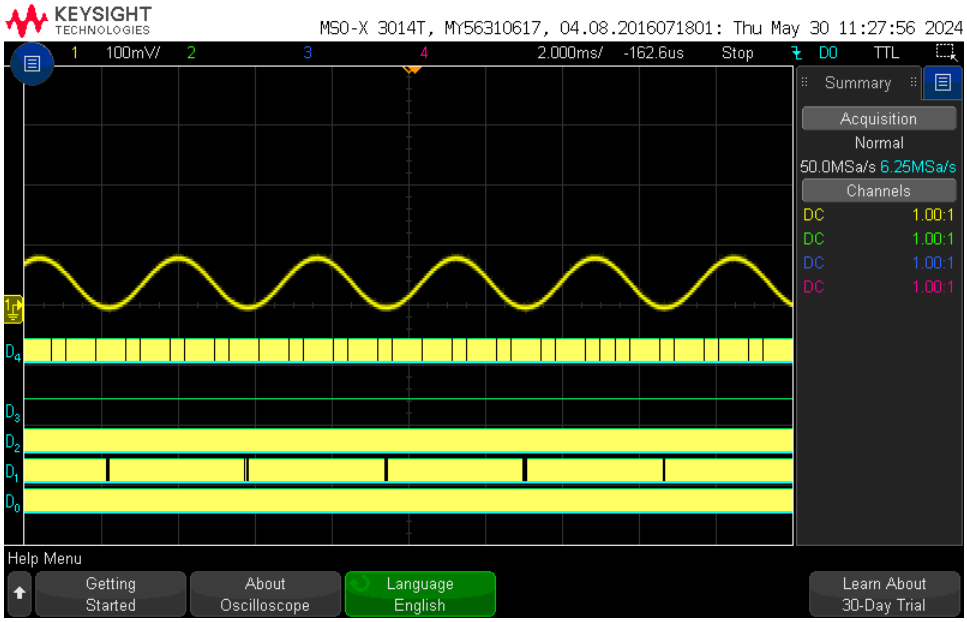
3.6.2. Hardware Validation



3.7. Overall Design

To ensure we have produced a proper sound wave for different keys, we used a frequency toner app to record and tell if our design produced a correct note.

3.7.1. *Hardware Validation*



4. Analysis of the Design

Despite not being able to play with multiple keys at the same time, this project overall was a big success. We were able to include more keys in the tone generator lookup table which produce notes from the first octave to the seventh octave. While this is great, it is terribly quiet to hear any note in the Octave 2 or below, or rather really loud in the highest notes.

As we were designing this controller, we were not expecting to encounter the two's complement output from the DDS, nor expecting the D2A to not be able to handle such integers. On the oscilloscope, we could see the D2A produced upper-half sine waves, instead of full sine waves. That did not sound as good. It turned out this could be solved with the method explained in the tone generator RTL. This proved to be rather challenging, but after discussing the problem and through our testing, we succeeded in solving this problem.

Furthermore, in the effort to produce better quality audio, we tried our best to smoothen out the sound wave. When a key on the keyboard was released, we could hear a buzzing sound. This happens when the sine signal suddenly stops, and drops from high magnitude to zeros. While we were able to reduce the buzzing noise by stopping the sine wave when it reached its lowest point, it was still evident that we needed to take a further look at the tone generator.

Lastly, we were incredibly satisfied with the volume setting by using the key velocity to produce four different volumes. When tapping a key very slowly, no sound was produced, and when pressing the key really fast, the volume went up at maximum.

4.1. Resource Utilization

For this MIDI design we used existing FPGA components in IP Catalog such as the DDS Compiler, and ROM. While it is possible to design these components from scratch, it would not be useful for us to spend time on it since these IP catalogs exist.

4.2. Residual Warnings

All components RTL Elaborate Designs were checked and none have any existing latches that could potentially cause any errors.

4.3. Division of Labor

To begin with, we met and worked on creating the high level diagram. This helped us see which components were needed for this project and roughly how much work would be needed to be put into each component for it to work appropriately.

Next, we divided the components up between us. Oskar worked on the MIDI_receiver, the digital amplifier, and the SPI transmitter, while Vuthy worked on the controller and datapath, and the tone generator. We worked separately on creating the RTL designs, code and behavioral simulations for these components, while keeping each other updated on all changes that we felt were necessary to improve each component.

When RTL designs, code and behavioral simulations were completed for all components, we worked together on the hardware validation for each component by connecting it to its previous component. During this process, we discussed together what modifications were needed to improve our design and how we could improve the controller. Finally, together, we connected all the components to each other and tested the full functionality of the completed design.

This strategy was very effective, especially since both of us put in a lot of effort to complete this project. We made sure to stay in good communication if we were working separately which made it easy for us to modify the designs if needed. Because of how we divided the components between each other, the workload was similar for the both of us. However, if either one of us was struggling with a particular component, we made sure to meet and discuss how to solve the problem.

Overall, this strategy made this project very manageable and enjoyable, creating a great foundation for a successful project.

4.4. Future Work

For future improvements, our next step would be to integrate being able to press multiple keys at the same time, and finding the correct sound from the keys pressed. Additionally, we could add the usage of the volume knob on the MIDI keyboard we were using to control the volume of the note, along with the velocity. Furthermore, we could experiment with implementing the modulation and pitch knobs on the keyboard to change the output sound of the controller.

5. Acknowledgements

Thank you to Dr. Luke, our professor for giving us engaging lectures on complex topics on Digital Electronics, setting the foundation for a successful and enjoyable final project.

Thank you so much to Tad Truex, our lab instructor for such a wonderful facilitation, and guidance on different design challenges.

Thank you to our TA, Wyatt Ellison, for guiding the overall design process especially through the overall RTL design process, as well as helping us researching ways to change the volume using the velocity register.

6. Conclusions

In conclusion, this project successfully demonstrated the design and implementation of a MIDI controller capable of interpreting digital input from a MIDI keyboard and producing high-quality audible output. By integrating MIDI and SPI protocols with a Direct Digital Synthesis (DDS) method, we developed a system that accurately replicates the dynamic response and sound generation of a traditional mechanical piano. Our design used a 50 kHz sampling rate, ensuring desirable audio quality, and effectively handled key velocity to adjust volume output.

Through our design, simulation, and hardware validation process, we addressed and overcame challenges such as the buzzing noise during the release of a key and the handling of two's complement outputs from the DDS. The project not only met the initial goals but also provided a valuable learning experience in digital electronics design, FPGA programming, and system integration.

If we were given the opportunity to go back one month and choose our final project again with all the knowledge we have now, we would definitely choose the MIDI controller again as our final project, since this proved to be a challenging but an enjoyable project. For future students interested in designing a MIDI controller for their final project, we advise them to keep Tad and their TA up to date throughout the project, since their advice will significantly improve the design process. Additionally, we would advise future students to fix the two's complement by shifting the sine wave up, making the minima of the wave be zero instead of the middle of the wave.

Future enhancements could include enabling the pressing of multiple keys simultaneously, integrating volume and modulation controls from the MIDI keyboard, and further refining the audio output quality. Overall, the project was a success, demonstrating comprehensive understanding and application of digital electronic concepts.

Appendix A: VHDL Source Code

Topshell

```
----------------------------------------------------------------------------------
-- Engineer: Oskar Magnusson & Vuthy Vey
--
-- Create Date: 05/24/2024
-- Design Name: MIDI Controller Top Shell
-- Module Name: MIDI_Controller_Shell - Behavioral
-- Project Name: MIDI Controller
-- Description: The top shell for the MIDI controller.
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library UNISIM;
use UNISIM.VComponents.all;

entity MIDI_Controller_Shell is
    Port ( Clk_Ext_Port     : in STD_LOGIC;
           RX_Port          : in STD_LOGIC;
           spi_cs_port      : out std_logic;
           TX_port   :   out STD_LOGIC;
           TX_done_port : out std_logic;
           d2a_sclk_port : out std_logic);
end MIDI_Controller_Shell;
```

```
architecture Behavioral of MIDI_Controller_Shell is

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--System Clock Generation:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
component MIDI_Controller_system_clock_generation is
    Port (
        --External Clock:
            input_clk_port          : in std_logic;
        --System Clock:
            system_clk_port         : out std_logic);
end component;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--MIDI SCI Receiver:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
signal Data_Out_Port :  STD_LOGIC_VECTOR (7 downto 0);
signal TC_Port :  STD_LOGIC;
signal Baud_TC_Port:  STD_LOGIC;

component MIDI_Receiver is
    Port (
          SCLK_Port        : in STD_LOGIC;
       RX_Port          : in STD_LOGIC;
       Data_Out_Port    : out STD_LOGIC_VECTOR (7 downto 0);
       TC_Port          : out STD_LOGIC;
       Baud_TC_Port     : out std_logic);
end component;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--datapath
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
component datapath is
PORT ( sclk_port                    :      in    STD_LOGIC;
        data_ready_port      :   in STD_LOGIC;
        data_in_port         :   in STD_LOGIC_VECTOR(7 downto 0);
        key_port             :   out STD_LOGIC_VECTOR(7 downto 0);
        key_on_port          :   out STD_LOGIC;
        velocity_port      :  out STD_LOGIC_VECTOR(7 downto 0));
end component;

signal key_port         : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
signal key_on_port      : STD_LOGIC;
signal velocity_port  : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Tone generator
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
component tone_gen is
PORT ( sclk_port                    :      in    STD_LOGIC;
        key_port             :   in STD_LOGIC_VECTOR(7 downto 0);
```

```
        key_on_port          :   in STD_LOGIC;
        phase_port           :   out STD_LOGIC_VECTOR(15 downto 0);
        t_valid_port         :   out STD_LOGIC);
end component;

signal phase_port           :  STD_LOGIC_Vector(15 downto 0);
signal t_valid_port   : std_logic;
--////////////////////////////////////////////////////////////////////////////////
--Direct Digital Synthesis
--////////////////////////////////////////////////////////////////////////////////
COMPONENT dds_compiler_0
  PORT (
    aclk : IN STD_LOGIC;
    s_axis_phase_tvalid : IN STD_LOGIC;
    s_axis_phase_tdata : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    m_axis_data_tvalid : OUT STD_LOGIC;
    m_axis_data_tdata : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
  );
END COMPONENT;
signal m_axis_data_tvalid : STD_LOGIC;
signal m_axis_data_tdata :  STD_LOGIC_VECTOR(15 DOWNTO 0);

--////////////////////////////////////////////////////////////////////////////////
--Digital Amplifier
--////////////////////////////////////////////////////////////////////////////////
component D_Amplifier is
    Port ( Data_In_Port : in STD_LOGIC_VECTOR(11 downto 0);
           Velocity_Port : in STD_LOGIC_VECTOR (7 downto 0);
           Amplitude_Port : out STD_LOGIC_VECTOR(11 downto 0));
end component;

signal amplitude : STD_LOGIC_VECTOR(11 downto 0) := (others => '0');

--////////////////////////////////////////////////////////////////////////////////
--SPI Transmitter
--////////////////////////////////////////////////////////////////////////////////
component SPI_Transmitter is
    Port ( SCLK_Port : in STD_LOGIC;
           Ready_Port : in STD_LOGIC;
           Data_In_Port : in STD_LOGIC_VECTOR (11 downto 0);
           SPI_CS_port : out std_logic;
           TX_port : out STD_LOGIC;
           TX_Done_Port : out STD_LOGIC;
           D2A_Sclk_port: out std_logic);
end component;

--////////////////////////////////////////////////////////////////////////////////
--Timing:
--////////////////////////////////////////////////////////////////////////////////
signal system_clk: std_logic := '0';

begin
```

```
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Wire the system clock generator into the shell with a port map:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
clocking: MIDI_Controller_system_clock_generation port map(
    input_clk_port  => Clk_Ext_Port,     -- External clock
    system_clk_port => system_clk );     -- System clock

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Wire the MIDI Receiver into the shell with a port map:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
FSM: MIDI_Receiver port map(
   SCLK_Port        => system_clk,
   RX_Port          => RX_Port,
   Data_Out_Port    => Data_Out_Port,
   TC_Port          => TC_Port,
   Baud_TC_Port     => Baud_TC_Port);

controller: datapath port map(
    sclk_port         => system_clk,
    data_ready_port    => TC_Port,
    data_in_port       => Data_Out_Port,
    key_port           => key_port,
    key_on_port        => key_on_port,
    velocity_port      => velocity_port);

tone_generator: tone_gen port map(
    sclk_port       => system_clk,
    key_port        => key_port,
    key_on_port     => key_on_port,
    phase_port      => phase_port,
    t_valid_port    => t_valid_port);

sine_lut : dds_compiler_0
  PORT MAP (
    aclk => system_clk,
    s_axis_phase_tvalid => t_valid_port,
    s_axis_phase_tdata => phase_port,
    m_axis_data_tvalid => m_axis_data_tvalid,
    m_axis_data_tdata => m_axis_data_tdata
  );

d_amp: D_Amplifier
    Port map (
        Data_In_Port => m_axis_data_tdata(11 downto 0),
        Velocity_Port => velocity_port,
        Amplitude_Port => amplitude
    );

spi: SPI_Transmitter
    Port map (
        SCLK_Port => system_clk,
```

```
        Ready_Port => m_axis_data_tvalid,
        Data_In_Port => amplitude,
        SPI_CS_port => SPI_CS_port,
        TX_port => TX_port,
        TX_done_port => TX_done_port,
        D2A_SCLK_port => D2A_SCLK_port
    );
end Behavioral;
```

Clock Generator

```
----------------------------------------------------------------------------------
-- Engineer: Oskar Magnusson & Vuthy Vey
--
-- Create Date: 05/24/2024
-- Design Name: MIDI Controller Clocking System
-- Module Name: MIDI_Controller_system_clock_generation - Behavioral
-- Project Name: MIDI Controller
-- Description: Clocking system for the MIDI controller. Generating a 1 MHz clock from
100MHz
--
----------------------------------------------------------------------------------



--================================================================================
--Library Declarations:
--================================================================================
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use ieee.math_real.all;
library UNISIM;
use UNISIM.VComponents.all;

--================================================================================
--Entity Declaration:
--================================================================================
entity MIDI_Controller_system_clock_generation is
    Port (
        --External Clock:
            input_clk_port          : in std_logic;
        --System Clock:
            system_clk_port         : out std_logic);
end MIDI_Controller_system_clock_generation;

--================================================================================
--Architecture Type:
--================================================================================
architecture behavioral_architecture of MIDI_Controller_system_clock_generation is
--================================================================================
--Signal Declarations:
```

```
--================================================================================
-- Signals for the 1 MHz system clock divider

--CONSTANT FOR SYNTHESIS:
constant CLOCK_DIVIDER_TC: integer := 50;
--CONSTANT FOR SIMULATION:
--constant CLOCK_DIVIDER_TC: integer := 5;

--Automatic register sizing:
constant COUNT_LEN                                  : integer := integer(ceil( log2(
real(CLOCK_DIVIDER_TC) ) ));
signal system_clk_divider_counter    : unsigned(COUNT_LEN-1 downto 0) := (others => '0');
signal system_clk_tog                        : std_logic := '0';


--================================================================================
--Processes:
--================================================================================
begin
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Clock (frequency) Divider):
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- Divide the master clock down to 16 Hz, then toggling the
-- clkdiv_tog signal at 2 MHz gives a 1 MHz clock with 50% duty cycle.
Clock_divider: process(input_clk_port)
begin
       if rising_edge(input_clk_port) then
              if system_clk_divider_counter = CLOCK_DIVIDER_TC-1 then                 --
Counts to 1/2 clk period
                     system_clk_tog <= NOT(system_clk_tog);
              -- T flip flop
                     system_clk_divider_counter <= (others => '0');
              -- Reset
              else
                     system_clk_divider_counter <= system_clk_divider_counter + 1;     --
Count up
              end if;
       end if;
end process Clock_divider;

--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- Clock buffer for the system clock
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- The BUFG component puts the slow clock onto the FPGA clocking network
Slow_clock_buffer: BUFG
      port map (I => system_clk_tog,
                O => system_clk_port );

end behavioral_architecture;
```

MIDI_Receiver

```
----------------------------------------------------------------------------------
-- Engineer: Oskar Magnusson & Vuthy Vey
--
-- Create Date: 05/22/2024 07:10:18 PM
-- Design Name: MIDI Receiver
-- Module Name: MIDI Receiver - Behavioral
-- Project Name: MIDI Controller
-- Description: An SCI receiver that takes in input from a MIDI keyboard one bit at a time
and outputs
-- an 8 bit byte to be used by the MIDI controller to generate the correct sound.
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity MIDI_Receiver is
    Port ( SCLK_Port : in STD_LOGIC;
           RX_Port : in STD_LOGIC;
           Data_Out_Port : out STD_LOGIC_VECTOR (7 downto 0);
           TC_Port : out STD_LOGIC;
           Baud_TC_Port: out STD_LOGIC);
end MIDI_Receiver;

architecture Behavioral of MIDI_Receiver is

-- FSM States
type state_type is (sIdle, sDataReady, sCount);
signal current_state, next_state : state_type := sIdle;

-- Signal declarations
signal count_clr    : std_logic := '0';
signal bit_tc       : std_logic := '0';
signal output_en    : std_logic := '0';
signal baud_tc      : std_logic := '0';

-- Registers
signal shift_reg    : std_logic_vector(9 downto 0) := (others => '0'); -- 10 bit input shift
register
signal out_reg      : std_logic_vector(7 downto 0) := (others => '0'); -- 8 bit output
regiter

-- Counters
signal baud_counter : unsigned(8 downto 0) := (others => '0'); -- Baud counter from 0 to
31249
signal bit_counter  : unsigned(3 downto 0) := (others => '0'); -- Bit counter to count all
10 bits in each input signal

-- Constants
constant MAX_BAUD   : unsigned(8 downto 0) := "000100000"; -- Maximum value for the baud
```

```
counter
constant MAX_BIT    : unsigned(3 downto 0) := "1010"; -- Maximum value for the bit counter
constant MID_BAUD   : unsigned(8 downto 0) := "000010000"; -- to sample in the middle of bit
width
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Synchronizer
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
signal synchronizer  : std_logic := '0';
signal synchronized_RX  : std_logic := '0';

begin
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Double-Flop Synchronizer:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Prevents metastability on the input by passing the raw button press through a
--double flop synchronizer.
synchronize: process(SCLK_Port)
begin
    if rising_edge(SCLK_Port) then
        synchronizer <= RX_Port;
        synchronized_RX <= synchronizer;
    end if;
end process;

-- State update
state_update: process(SCLK_Port)
begin
    if rising_edge(SCLK_Port) then
        current_state <= next_state;
    end if;
end process state_update;

-- NS Logic
NS_Logic: process(current_state, synchronized_RX, bit_tc)
begin
    -- Defaults
    next_state <= current_state;

    case current_state is
        when sIdle =>
            if (synchronized_RX = '0') then
                next_state <= sCount;
            end if;

        when sCount =>
            if (bit_tc = '1') then
                next_state <= sDataReady;
            end if;

        when sDataReady =>
            next_state <= sIdle;
```

```
        when others =>
            next_state <= sIdle;
    end case;
end process NS_Logic;

-- Output Logic
Output_Logic: process(current_state)
begin
     -- Defaults
    count_clr <= '0';
    output_en <= '0';
    TC_Port <= '0';

    case current_state is
        when sIdle =>
            count_clr <= '1';

        when sCount =>
            -- Nothing to output

        when sDataReady =>
            output_en <= '1';
            TC_Port <= '1';

        when others =>
            count_clr <= '0';
            output_en <= '0';
    end case;
end process Output_Logic;

-- Counters
Counters: process(SCLK_Port, baud_counter, bit_counter)
begin
    -- Baud counter
    if rising_edge(SCLK_Port) then
        baud_counter <= baud_counter + 1;

        if (count_clr) = '1' then
            baud_counter <= (others => '0');
        elsif (baud_tc = '1') then
                baud_counter <= (others => '0');
        end if;
    end if;

    -- Asynchronous TC
    baud_tc <= '0';
    if (baud_counter = MAX_BAUD-1) then
        baud_tc <= '1';
    end if;

    -- Bit counter
    if rising_edge(SCLK_Port) then
```

```
        if (baud_tc = '1') then
            bit_counter <= bit_counter + 1;
        end if;

        if (count_clr) = '1' then
            bit_counter <= (others => '0');
        elsif (bit_tc = '1') then
            bit_counter <= (others => '0');
        end if;
    end if;

    -- Asynchronous TC
    bit_tc <= '0';
    if (bit_counter = MAX_BIT) then
        bit_tc <= '1';
    end if;

end process Counters;

-- Registers
Registers: process(SCLK_Port, baud_counter)
begin
    -- Input shift Register
    if rising_edge(SCLK_Port) then
        if (baud_counter = MID_BAUD - 1) then
            shift_reg <= synchronized_RX & shift_reg(9 downto 1);
        end if;
    end if;

    -- Output shift register
    if rising_edge(SCLK_Port) then
        if (output_en = '1') then
            out_reg <= shift_reg(8 downto 1);
        end if;
    end if;

end process Registers;

-- Connect the outputs to the correct signals
Data_Out_Port <= std_logic_vector(out_reg);
Baud_TC_Port <= baud_tc;
end Behavioral;
```

Controller

```
----------------------------------------------------------------------------------
-- Engineer: Oskar Magnusson & Vuthy Vey
--
-- Create Date: 05/23/2024 01:40:27 PM
-- Design Name: MIDI Datapath
```

```
-- Module Name: datapath - Behavioral
-- Project Name: MIDI Controller
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity datapath is
PORT ( sclk_port                    :      in    STD_LOGIC;
        data_ready_port      :   in STD_LOGIC;
        data_in_port         :   in STD_LOGIC_VECTOR(7 downto 0);
        key_port             :   out STD_LOGIC_VECTOR(7 downto 0);
        key_on_port          :   out STD_LOGIC;
        velocity_port      :  out STD_LOGIC_VECTOR(7 downto 0));
end datapath;

architecture Behavioral of datapath is

type state_type is (sIdle, sLoadStatus, sWaitLoadKey, sLoadKey, sWaitLoadVelo,
sLoadVelocity, sWaitClrKey, sClrKey, sWaitClrVelo, sClrVelocity);
signal current_state, next_state : state_type := sIdle;

SIGNAL key_reg : STD_LOGIC_VECTOR(7 downto 0):= (others => '0');
SIGNAL status_reg: STD_LOGIC := '0';
SIGNAL velocity_reg: STD_LOGIC_VECTOR(7 downto 0) := (others => '0');

SIGNAL pressed, released : STD_LOGIC := '0';

SIGNAL key_en, key_clr, status_en, status_clr, velocity_en, velocity_clr : STD_LOGIC := '0';

begin

--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--State Update
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
state_update: process(sclk_port)
begin
    if rising_edge(sclk_port) then
        current_state <= next_state;
    end if;
end process state_update;
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Next State Update
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
next_state_logic: process(current_state, data_ready_port, pressed, released)
begin
    next_state <= current_state;

    case (current_state) is
        when sIdle => if data_ready_port = '1' then
                        next_state <= sLoadStatus;
```

```
                    end if;

        when sLoadStatus =>
            if pressed = '1' then
                next_state <= sWaitLoadKey;
            elsif released = '1' then
                next_state <= sWaitClrKey;
            end if;

        when sWaitLoadKey =>
            if data_ready_port = '1' then
                next_state <= sLoadKey;
            end if;

        when sLoadKey =>
            next_state <= sWaitLoadVelo;

        when sWaitLoadVelo =>
            if data_ready_port = '1' then
                next_state <= sLoadVelocity;
            end if;

        when sLoadVelocity =>
            next_state <= sIdle;

        when sWaitClrKey =>
            if data_ready_port = '1' then
                next_state <= sClrKey;
            end if;

        when sClrKey => next_state <= sWaitClrVelo;

        when sWaitClrVelo =>
            if data_ready_port = '1' then
                next_state <= sClrVelocity;
            end if;

        when sClrVelocity =>
            next_state <= sIdle;

        when others =>
            next_state <= sIdle;

    end case;
end process next_state_logic;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Output Logic
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
output_logic : process(current_state)
begin
     -- default values
```

```
    key_en <= '0';
    key_clr <= '0';
    status_en<= '0';
    status_clr <= '0';
    velocity_en <= '0';
    velocity_clr <= '0';

    case (current_state) is
        when sIdle =>

        when sLoadStatus =>
            status_en <= '1';

        when sLoadKey =>
            key_en <= '1';

        when sLoadVelocity =>
            velocity_en <= '1';

        when sClrKey =>
            key_clr <= '1';

        when sClrVelocity =>
            velocity_clr <= '1';

        when others =>

    end case;
end process output_logic;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Datapath
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
datapath: process(sclk_port)
begin

    -- Status Register
    if rising_edge(sclk_port) then
        if status_clr = '1' then
            status_reg <= '0';
        else
           if status_en = '1' then
                if pressed =  '1' then
                    status_reg <= '1';
                elsif released = '1' then
                    status_reg <= '0';
                end if;

           end if;
        end if;
    end if;
```

```
    -- Key Register
    if rising_edge(sclk_port) then
        if key_clr = '1' then
            key_reg <= (others => '0');
        else
            if key_en = '1' then
                key_reg <= data_in_port;
            end if;
        end if;
    end if;

    -- Velocity Register
    if rising_edge(sclk_port) then
        if velocity_clr = '1' then
            velocity_reg <= (others => '0');
        else
            if velocity_en = '1' then
                velocity_reg <= data_in_port;
            end if;
        end if;
    end if;

    --////////////////////////////////////////////////////////////////////////////////
    --Asynchronous
    --////////////////////////////////////////////////////////////////////////////////
    -- comparators
    pressed <= '0';
    if data_in_port(7 downto 4) = "1001" then
        pressed <= '1';
    end if;

    released <= '0';
    if data_in_port(7 downto 4) = "1000" then
        released <= '1';
    end if;

end process;

-- output
key_port <= key_reg;
key_on_port <= status_reg;
velocity_port <= velocity_reg;

end Behavioral;
```

Tone Gen

```
----------------------------------------------------------------------------------
-- Engineer: Oskar Magnusson & Vuthy Vey
--
-- Create Date: 05/23/2024 01:40:27 PM
-- Design Name: Tone Generator
```

```
-- Module Name: tone_gen - Behavioral
-- Project Name: MIDI Controller
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity tone_gen is
    PORT (     sclk_port                    :      in    STD_LOGIC;
            key_port             :   in STD_LOGIC_VECTOR(7 downto 0);
            key_on_port          :   in STD_LOGIC;
            phase_port           :   out STD_LOGIC_VECTOR(15 downto 0);
            t_valid_port         :   out STD_LOGIC
        );
end tone_gen;

architecture Behavioral of tone_gen is
    COMPONENT dist_mem_gen_0
      PORT (
        a : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        spo : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
      );
    END COMPONENT;

    SIGNAL m : STD_LOGIC_VECTOR(15 downto 0);
    SIGNAL phase_reg : UNSIGNED(15 downto 0) := "1100000000000000"; -- Lowest possible bit
(Low of the sin wave)

    CONSTANT FS_MAX : UNSIGNED(4 downto 0) := "10100";
    SIGNAL frequency_counter : UNSIGNED(4 downto 0) := "00000";
    SIGNAL frequency_tc : STD_LOGIC := '0';

    signal reset : std_logic := '0';
    signal Low_Point : std_logic := '0';
    signal old_phase : unsigned(15 downto 0) := (others => '0');

    type state_type is (sIdle, sLoad);
    signal current_state, next_state : state_type := sIdle;

begin

-- State update
```

```
state_update: process(SCLK_Port)
begin
    if rising_edge(SCLK_Port) then
        current_state <= next_state;
    end if;
end process state_update;

-- NS Logic - Only change states when sin wave is at 0
NS_Logic: process(current_state, key_on_port, Low_Point)
begin
    -- Defaults
    next_state <= current_state;

    case current_state is
        when sIdle =>
            if ((key_on_port = '1')) then
                next_state <= sLoad;
            end if;

        when sLoad =>
            if ((key_on_port = '0') and (Low_Point = '1')) then
                next_state <= sIdle;
            end if;

        when others =>
            next_state <= current_state;

    end case;
end process NS_Logic;

-- NS Logic
Output_Logic: process(current_state)
begin
    -- Defaults
    reset <= '0';

    case current_state is
        when sIdle =>
        reset <= '1';

        when sLoad =>

        when others =>
            reset <= '0';

    end case;
end process Output_Logic;

frequency_lut : dist_mem_gen_0
  PORT MAP (
    a => key_port,
    spo => m
```

```
  );

phase_counter : process(sclk_port,frequency_tc, key_on_port)
begin
    if rising_edge(sclk_port) then

        t_valid_port <= '0';
        if reset = '1' then
            phase_reg <= "1100000000000000"; -- Reset to low
        else
            if frequency_tc = '1' then
            old_phase <= phase_reg;
                phase_reg <= phase_reg + unsigned(m);
            end if;
        end if;

        t_valid_port <= '0';
        if frequency_tc = '1' then
            t_valid_port <= '1';
        end if;
    end if;

    --
    Low_Point <= '0';
    if ((old_phase(15) = '1') and (old_phase(14) = not(phase_reg(14)))) then
        Low_Point <= '1';
    end if;

end process phase_counter;

frequency_count : process(sclk_port, frequency_counter)
begin
    if rising_edge(sclk_port) then

        if frequency_tc = '1' then
            frequency_counter <= "00000";
        else
            frequency_counter <= frequency_counter + 1;
        end if;
    end if;

    frequency_tc <= '0';
    if frequency_counter = FS_MAX then
        frequency_tc <= '1';
    end if;
end process frequency_count;


phase_port <= STD_LOGIC_VECTOR(phase_reg);
end Behavioral;
```

Digital Amplifier

```
----------------------------------------------------------------------------------
-- Engineer: Oskar Magnusson & Vuthy Vey
--
-- Create Date: 05/24/2024
-- Design Name: Digital Amplifier
-- Module Name: Digital Amplifier - Behavioral
-- Project Name: MIDI Controller
-- Description: An asynchronous amplifier for a MIDI controller that takes in a 12-bit input
from Direct Digital Synthesizer and a 3-bit velocity and amplifies the input using the
velocity byte.
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity D_Amplifier is
    Port ( Data_In_Port : in STD_LOGIC_VECTOR(11 downto 0);
           Velocity_Port : in STD_LOGIC_VECTOR (7 downto 0);
           Amplitude_Port : out STD_LOGIC_VECTOR(11 downto 0));
end D_Amplifier;

architecture Behavioral of D_Amplifier is

-- Signal declaration
signal Amplified_Data : unsigned(11 downto 0) := (others => '0');

begin
-- Amplification process
Amplify : process(Data_In_Port, Velocity_Port)
begin
       -- Multiply data with velocity to amplify
    -- Ignore the three least significant bits of the data (make sure we will never reach
the voltage limit)
    -- Only use three most significant bits of the velocity
    Amplified_Data <= (unsigned((not(Data_In_Port(11))) & (Data_In_Port(10 downto 4))) *
unsigned(Velocity_Port(7 downto 4)));  -- Flip MSB to fix two's complement bug (move the
number up)
end process Amplify;

-- Assign to output
Amplitude_Port <= std_logic_vector(Amplified_Data);

end Behavioral;
```

SPI Transmitter

```
----------------------------------------------------------------------------------
-- Engineer: Oskar Magnusson & Vuthy Vey
--
-- Create Date: 05/24/2024
-- Design Name: SPI Transmitter
-- Module Name: SPI Transmitter - Behavioral
-- Project Name: MIDI Controller
-- Description: An SPI transmitter that takes in 12-input from the digital amplifier and
transmits it, one bit at a time to the D2A converter.
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity SPI_Transmitter is
    Port ( SCLK_Port : in STD_LOGIC;
           Ready_Port : in STD_LOGIC;
           Data_In_Port : in STD_LOGIC_VECTOR (11 downto 0);
           SPI_CS_port : out std_logic;
           TX_port : out STD_LOGIC;
           TX_Done_Port : out STD_LOGIC;
           D2A_Sclk_port: out std_logic);
end SPI_Transmitter;

architecture Behavioral of SPI_Transmitter is

-- FSM States
type state_type is (sIdle, sLoad, sShift, sDone);
signal current_state, next_state : state_type := sIdle;

-- Signal declarations
signal count_clr    : std_logic := '0';
signal bit_count_tc   : std_logic := '0';
signal load_en        : std_logic := '0';
signal shift_en     : std_logic := '0';

-- Register
signal shift_reg    : std_logic_vector(15 downto 0) := (others => '0'); -- 12 bit input
shift register with 4 leading 0s

signal bit_counter  : unsigned(3 downto 0) := (others => '0'); -- Bit counter to count all
12 bits in each input signal

-- Constants
constant MAX_BIT    : unsigned(3 downto 0) := "1111"; -- Maximum value for the bit counter
(11)

begin
```

```
-- State update
state_update: process(SCLK_Port)
begin
    if rising_edge(SCLK_Port) then
        current_state <= next_state;
    end if;
end process state_update;

-- NS Logic
NS_Logic: process(current_state, Ready_Port, bit_count_tc)
begin
    -- Defaults
    next_state <= current_state;

    case current_state is
        when sIdle =>
            if (Ready_Port = '1') then
               next_state <= sLoad;
            end if;

        when sLoad =>
            next_state <= sShift;

        when sShift =>
            if (bit_count_tc = '1') then
               next_state <= sDone;
            end if;

        when sDone =>
            next_state <= sIdle;

        when others =>
            next_state <= sIdle;
    end case;
end process NS_Logic;

-- Output Logic
Output_Logic: process(current_state)
begin
     -- Defaults
    count_clr <= '0';
    load_en <= '0';
    shift_en <= '0';
    SPI_CS_port <= '1';
    TX_Done_Port <= '0';

    case current_state is
        when sIdle =>
            count_clr <= '1';

        when sLoad =>
            load_en <= '1';
```

```
            count_clr <= '1';

        when sShift =>
            shift_en <= '1';
            SPI_CS_port <= '0';

        when sDone =>
            TX_Done_Port <= '1';

        when others =>
            count_clr <= '0';
            load_en <= '0';
               shift_en <= '0';
               SPI_CS_port <= '1';
               TX_Done_Port <= '0';
    end case;
end process Output_Logic;

-- Datapath
Datapath: process(SCLK_Port, bit_counter)
begin
    D2A_Sclk_port <= not(SCLK_Port);

    -- Bit counter
    if rising_edge(SCLK_Port) then
        if (count_clr) = '1' then
            bit_counter <= (others => '0');
        else
            if (bit_counter = MAX_BIT) then
                bit_counter <= (others => '0');
            else
                bit_counter <= bit_counter + 1;
            end if;
        end if;
    end if;

    -- Asynchronous TC
    bit_count_tc <= '0';
    if (bit_counter = MAX_BIT) then
        bit_count_tc <= '1';
    end if;

    -- Shift Register
    if rising_edge(SCLK_Port) then
      if (load_en = '1') then
          -- Send four leading 0s to D2A
          shift_reg <= "0000" & Data_In_Port;
      else
          if (shift_en = '1') then
              shift_reg <= shift_reg(14 downto 0) & '0';
          end if;
      end if;
```

```
    end if;

end process Datapath;

-- Connect the outputs to the correct signals
TX_Port <= shift_reg(15);

end Behavioral;
```

Appendix B: VHDL Testbenches

MIDI Receiver

```
----------------------------------------------------------------------------------
-- Engineer: Oskar Magnusson & Vuthy Vey
--
-- Create Date: 05/23/2024
-- Design Name: MIDI Receiver testbench
-- Project Name: MIDI Controller
-- Description: A testbench for the SCI MIDI Receiver
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity MIDI_Receiver_tb is
end MIDI_Receiver_tb;

architecture Behavioral of MIDI_Receiver_tb is
    -- Component declaration for the MIDI_Receiver
    component MIDI_Receiver
        Port (
            SCLK_Port : in STD_LOGIC;
            RX_Port : in STD_LOGIC;
            Data_Out_Port : out STD_LOGIC_VECTOR (7 downto 0);
            TC_Port : out STD_LOGIC
        );
    end component;

    -- Signals to connect to the MIDI_Receiver
    signal SCLK_Port_tb : STD_LOGIC := '0';
    signal RX_Port_tb : STD_LOGIC := '1';
    signal Data_Out_Port_tb : STD_LOGIC_VECTOR(7 downto 0);
    signal TC_Port_tb : STD_LOGIC;

    -- Clock period definition
    constant clk_period : time := 1000 ns;
```

```
    begin

-- Instantiate the MIDI_Receiver
uut: MIDI_Receiver
    Port map (
        SCLK_Port => SCLK_Port_tb,
        RX_Port => RX_Port_tb,
        Data_Out_Port => Data_Out_Port_tb,
        TC_Port => TC_Port_tb);

-- Clock generation process
process
begin
    SCLK_Port_tb <= '0';
    wait for clk_period/2;
    SCLK_Port_tb <= '1';
    wait for clk_period/2;
end process;

-- Stimulus process
process
begin
    -- Initialize inputs
    RX_Port_tb <= '1';
    wait for 10 * clk_period*325;

    -- Start bit
    RX_Port_tb <= '0';
    wait for clk_period*32;

    -- Send 8-bit data (01010101)
    RX_Port_tb <= '1'; -- Bit 0
    wait for clk_period*32;
    RX_Port_tb <= '0'; -- Bit 1
    wait for clk_period*32;
    RX_Port_tb <= '1'; -- Bit 2
    wait for clk_period*32;
    RX_Port_tb <= '0'; -- Bit 3
    wait for clk_period*32;
    RX_Port_tb <= '1'; -- Bit 4
    wait for clk_period*32;
    RX_Port_tb <= '0'; -- Bit 5
    wait for clk_period*32;
    RX_Port_tb <= '1'; -- Bit 6
    wait for clk_period*32;
    RX_Port_tb <= '0'; -- Bit 7
    wait for clk_period*32;

    -- Stop bit
    RX_Port_tb <= '1';
    wait for clk_period*325;
```

```
        -- Wait for the receiver to process the data
        wait for 20 * clk_period*325;

        -- Start bit
        RX_Port_tb <= '0';
        wait for clk_period*32;

        -- Send 8-bit data (10011001)
        RX_Port_tb <= '1'; -- Bit 0
        wait for clk_period*32;
        RX_Port_tb <= '0'; -- Bit 1
        wait for clk_period*32;
        RX_Port_tb <= '0'; -- Bit 2
        wait for clk_period*32;
        RX_Port_tb <= '1'; -- Bit 3
        wait for clk_period*32;
        RX_Port_tb <= '1'; -- Bit 4
        wait for clk_period*32;
        RX_Port_tb <= '0'; -- Bit 5
        wait for clk_period*32;
        RX_Port_tb <= '0'; -- Bit 6
        wait for clk_period*32;
        RX_Port_tb <= '1'; -- Bit 7
        wait for clk_period*32;

        -- Stop bit
        RX_Port_tb <= '1';
        wait for clk_period*325;

        -- Finish simulation
        wait;
    end process;
end Behavioral;
```

Datapath

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY datapath_controller_tb IS
END datapath_controller_tb;

ARCHITECTURE behavior OF datapath_controller_tb IS

    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT datapath
```

```
    PORT(
         sclk_port : IN  std_logic;
         data_ready_port : IN  std_logic;
         data_in_port : IN  std_logic_vector(7 downto 0);
         key_port : OUT  std_logic_vector(7 downto 0);
         key_on_port : OUT  std_logic;
         velocity_port : OUT  std_logic_vector(7 downto 0)
        );
    END COMPONENT;


    --Inputs
    signal sclk_port : std_logic := '0';
    signal data_ready_port : std_logic := '0';
    signal data_in_port : std_logic_vector(7 downto 0) := (others => '0');

    --Outputs
    signal key_port : std_logic_vector(7 downto 0);
    signal key_on_port : std_logic;
    signal velocity_port : std_logic_vector(7 downto 0);

    -- Clock period definitions
    constant sclk_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: datapath PORT MAP (
          sclk_port => sclk_port,
          data_ready_port => data_ready_port,
          data_in_port => data_in_port,
          key_port => key_port,
          key_on_port => key_on_port,
          velocity_port => velocity_port
        );

    -- Clock process definitions
    sclk_process :process
    begin
        sclk_port <= '0';
        wait for sclk_period/2;
        sclk_port <= '1';
        wait for sclk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        wait for 95 ns;

        -- Test sequence
```

```
        -- Test Case 1: Load Key
        data_ready_port <= '1';
        data_in_port <= "10000000"; -- "1000" indicates a key press
        wait for sclk_period;
        data_ready_port <= '0';
        wait for sclk_period*10;

        -- Test Case 2: Load Key
        data_ready_port <= '1';
        data_in_port <= "01010111";
        wait for sclk_period;
        data_ready_port <= '0';
        wait for sclk_period*10;

        -- Test Case 2: Load Velocity
        data_ready_port <= '1';
        data_in_port <= "01111111";
        wait for sclk_period;
        data_ready_port <= '0';
        wait for sclk_period*10;

        -- Test Case 3: Clear Key
        data_ready_port <= '1';
        data_in_port <= "10010000"; -- "1001" indicates a key release
        wait for sclk_period;
        data_ready_port <= '0';
        wait for sclk_period*10;

        -- Test Case 2: Clear Key
        data_ready_port <= '1';
        data_in_port <= "01010111";
        wait for sclk_period;
        data_ready_port <= '0';
        wait for sclk_period*10;

        -- Test Case 4: Clear Velocity
        data_ready_port <= '1';
        data_in_port <= "00000000";
        wait for sclk_period;
        data_ready_port <= '0';
        wait for sclk_period*10;

        -- Add more test cases here if needed

        -- End of test
        wait;
    end process;

END;
```

Tone Generator

```
----------------------------------------------------------------------------------
-- Engineer: Oskar Magnusson & Vuthy Vey
-- Create Date: 05/23/2024 09:32:20 PM
-- Design Name: Tone Generator testbench
-- Module Name: tone_gen_tb - Behavioral
-- Project Name: MIDI Controller
-- Description: Testbench for the tone_gen module
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity tone_gen_tb is
end tone_gen_tb;

architecture Behavioral of tone_gen_tb is
    -- Component declaration for the Unit Under Test (UUT)
    component tone_gen is
        port (
            sclk_port       : in  std_logic;
            key_port        : in  std_logic_vector(7 downto 0);
            key_on_port     : in  std_logic;
            phase_port      : out std_logic_vector(15 downto 0);
            t_valid_port    : out std_logic
        );
    end component;

    -- Signal declarations to connect to UUT
    signal sclk         : std_logic := '0';
    signal key          : std_logic_vector(7 downto 0) := (others => '0');
    signal key_on       : std_logic := '0';
    signal phase        : std_logic_vector(15 downto 0);
    signal t_valid      : std_logic;

    -- Clock period
    constant clk_period : time := 10 ns;

begin
    -- Instantiate the Unit Under Test (UUT)
    uut: tone_gen
        port map (
            sclk_port     => sclk,
            key_port      => key,
            key_on_port   => key_on,
            phase_port    => phase,
            t_valid_port  => t_valid
        );

    -- Clock process definitions
```

```
    clk_process : process
    begin
        begin
        sclk <= '0';
        wait for clk_period/2;
        sclk <= '1';
        wait for clk_period/2;
    end process;    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;

        -- Simulate C key press with key_on signal
        key <= "00101100"; -- example key
        key_on <= '1';
        wait for 600 * clk_period ;

        key_on <= '0';
        wait for 20 * clk_period ;

        -- Test different keys
        key <= "00110000"; -- another example key
        key_on <= '1';
        wait for 600 * clk_period ;

        key_on <= '0';
        wait for 100 ns;

        -- Add more tests as necessary
        wait;
    end process;
end Behavioral;
```

Digital Amplifier

```
----------------------------------------------------------------------------------
-- Engineer: Oskar Magnusson & Vuthy Vey
--
-- Create Date: 05/24/2024
-- Design Name: Digital Amplifier testbench
-- Project Name: MIDI Controller
-- Description: A testbench for the digital amplifier
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.NUMERIC_STD.ALL;

entity D_Amplifier_tb is
end D_Amplifier_tb;

architecture Behavioral of D_Amplifier_tb is

    -- Component declaration for the unit under test (UUT)
    component D_Amplifier
        Port ( Data_In_Port : in STD_LOGIC_VECTOR(11 downto 0);
               Velocity_Port : in STD_LOGIC_VECTOR (7 downto 0);
               Amplitude_Port : out STD_LOGIC_VECTOR(11 downto 0));
    end component;

    -- Signals to connect to UUT
    signal Data_In_Port : STD_LOGIC_VECTOR(11 downto 0);
    signal Velocity_Port : STD_LOGIC_VECTOR (7 downto 0);
    signal Amplitude_Port : STD_LOGIC_VECTOR(11 downto 0);

begin

    -- Instantiate the Unit Under Test (UUT)
    uut: D_Amplifier
        Port map (
            Data_In_Port => Data_In_Port,
            Velocity_Port => Velocity_Port,
            Amplitude_Port => Amplitude_Port
        );

    -- Stimulus process
    stimulus: process
    begin
        -- Test case 1
        Data_In_Port <= "000011001100"; -- Example 12-bit data input
        Velocity_Port <= "01000000"; -- Example 8-bit velocity input
        wait for 10 ns;

        -- Test case 2
        Data_In_Port <= "111100001111"; -- Another 12-bit data input
        Velocity_Port <= "10000000"; -- Another 8-bit velocity input
        wait for 10 ns;

        -- Test case 3
        Data_In_Port <= "000000000000"; -- Zero data input
        Velocity_Port <= "11111111"; -- Maximum velocity input
        wait for 10 ns;

        -- Test case 4
        Data_In_Port <= "111111111111"; -- Maximum data input
        Velocity_Port <= "00100000"; -- Minimum velocity input
        wait for 10 ns;
```

```
        -- Test Case 5
        Data_In_Port <= "000011101101"; -- Example 12-bit data input
        Velocity_Port <= "01100110"; -- Example 8-bit velocity input
        wait for 10 ns;

        -- Test Case 6 - Should produce the same output as Test Case 5
        Data_In_Port <= "000011101010"; -- Example 12-bit data input
        Velocity_Port <= "01111010"; -- Example 8-bit velocity input
        wait for 10 ns;

        wait;
    end process;

end Behavioral;
```

SPI Transmitter

```
----------------------------------------------------------------------------------
-- Engineer: Oskar Magnusson & Vuthy Vey
--
-- Create Date: 05/24/2024
-- Design Name: SPI Transmitter tb
-- Module Name: SPI Transmitter tb - Behavioral
-- Project Name: MIDI Controller
-- Description: A testbench for the SPI transmitter.
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity SPI_Transmitter_tb is
end SPI_Transmitter_tb;

architecture Behavioral of SPI_Transmitter_tb is

    -- Component Declaration for the Unit Under Test (UUT)
    component SPI_Transmitter is
    Port ( SCLK_Port : in STD_LOGIC;
           Ready_Port : in STD_LOGIC;
           Data_In_Port : in STD_LOGIC_VECTOR (11 downto 0);
           SPI_CS_port : out std_logic;
           TX_port : out STD_LOGIC;
           TX_Done_Port : out STD_LOGIC;
           D2A_SCLK_port: out std_logic);
    end component;

    -- Signals to connect to the UUT
```

```
    signal SCLK_Port : STD_LOGIC := '0';
    signal Ready_Port : STD_LOGIC := '0';
    signal Data_In_Port : STD_LOGIC_VECTOR (11 downto 0) := (others => '0');
    signal TX_port : STD_LOGIC;
    signal SPI_CS_port : std_logic;
    signal TX_Done_Port: std_logic;
    signal D2A_SCLK_port: std_logic;

    -- Clock period definitions
    constant clk_period : time := 10 ns;

begin
    -- Instantiate the Unit Under Test (UUT)
    uut: SPI_Transmitter
    Port map (
        SCLK_Port => SCLK_Port,
        Ready_Port => Ready_Port,
        Data_In_Port => Data_In_Port,
        SPI_CS_port => SPI_CS_port,
        TX_port => TX_port,
        TX_done_port => TX_done_port,
        D2A_SCLK_port => D2A_SCLK_port
    );

    -- Clock process definitions
    clk_process :process
    begin
        SCLK_Port <= '0';
        wait for clk_period/2;
        SCLK_Port <= '1';
        wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;

        -- Load first data into the SPI transmitter
        Ready_Port <= '1';
        Data_In_Port <= "110011001100"; -- Example data
        wait for clk_period;
        Ready_Port <= '0';

        -- Wait for the data to be shifted out
        wait for 20 * clk_period;

        -- Check another data sequence
        Ready_Port <= '1';
        Data_In_Port <= "101010101010"; -- Example data
        wait for clk_period;
```

```
        Ready_Port <= '0';

        -- Wait for the data to be shifted out
        wait for 20 * clk_period;

        -- Check another data sequence
        Ready_Port <= '1';
        Data_In_Port <= "111100001111"; -- Example data
        wait for clk_period;
        Ready_Port <= '0';

        -- Wait for the data to be shifted out
        wait for 20 * clk_period;

              -- Check another data sequence
        Ready_Port <= '1';
        Data_In_Port <= "111111001000"; -- Example data
        wait for clk_period;
        Ready_Port <= '0';

        wait;
    end process;

end Behavioral;
```