Engs 31 / CoSc 56

Final Project Report

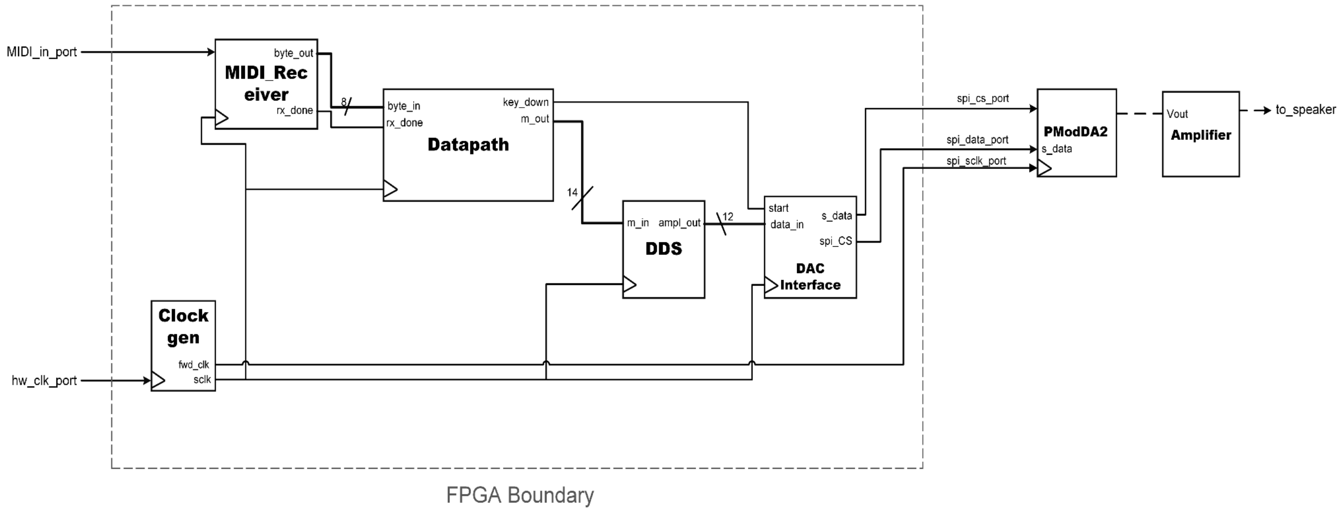
Abstract: This document presents the design and implementation of a MIDI keyboard controller system on a Basys3 FPGA board. The system is designed to accurately receive MIDI input from a keyboard, generate corresponding frequencies, and output a sine wave on an analog speaker. The key components include a MIDI receiver, clock generator, datapath, Direct Digital Synthesis (DDS), and DAC interface. Each component is thoroughly described, including its functionality, ports, RTL diagrams, FSM diagrams, and validation strategies. The overall structure of this report is as follows: First we present a system overview where we approach the design at a high level, with brief subcomponent descriptions. Next, in the technical description section, we describe each component in greater detail. Thirs, we document the testing and validation process, starting with each component in isolation, and then the overall system. We validate the design both in simulation and on hardware. Finally, we analyze our overall design and design process, and conclude with remarks on our experience with the project.

Samuel R. Barton & Grant M. Foley

1. System Overview

The system should handle key inputs from a MIDI keyboard and play frequencies consistent with the correct musical tone on an analog speaker. The overall goal of the design is to be able to handle a single key press at a time, and have nine different volumes (including note off). The design should be simple, with low-latency so that there is no delay from key press to output and no output without a key press. The design should also be robust so that it will not move into a bad state—if it receives MIDI messages other than key_press or key_release, it instead safely ignores them.

1.1. *Top-level Block Diagram*



1.2. Description of Ports

- **Input** *MIDI_in_port* is the serial stream of MIDI packets coming from the keyboard.
- **Input** *hw_clk_port* is the 100 MHz clock on the Basys3 board.
- **Output** *spi_cs_port* is the chip select port for SPI communication with the PModDA2 external component.
- **Output** *spi_data_port* is the data connection to the DA2.
- **Output** *spi_sclk_port* is the shared clock between the MIDI controller and the external DA2 component.

1.3. Description of Components

- *MIDI_Receiver*: The midi receiver processes the serial data stream coming in over the MIDI connector at the specified baud rate, and passes each byte to the datapath.
- *Clockgen*: The clockgen component generates a 1 MHz system clock used to drive all clocked processes in the design. It also forwards a clock signal to the external DA2 component for SPI communication.
- *Datapath*: The datapath interprets each MIDI byte, and passes along correct *m* and *velocity* values to the corresponding components on the output end of the controller. The datapath also keeps track of whether a key is pressed or not.
- *DDS*: The direct digital synthesis component outputs the appropriate amplitudes of a sine wave at a specified sample rate to play a specific frequency.
- *DAC Interface*: The digital to analog converter communicates over a SPI protocol to output an analog voltage from 12-bit digital values.

2. Technical Description

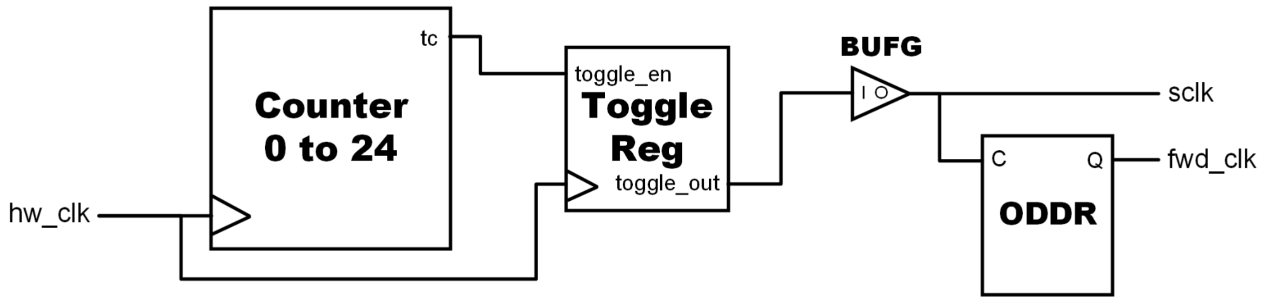
2.1. Clock_Gen

The Clock_Gen component generates a 1 MHz clock signal using the hardware clock which runs at 100 MHz. This system clock frequency was chosen according to the recommended specifications of the PMOD DA2 component. The maximum clock speed for the DA2 is 30 MHz, and the 1 MHz clock fits well into these specifications.

2.1.1. Description of Ports

- **Input** *hw_clk* is the hardware clock signal on the Basys3 which has a frequency of 100 MHz
- **Output** *sclk* is the 1 MHz clock signal used to drive all clocked components in the system.
- **Output** *fwd_clk* is the 1 MHz clock signal which is forwarded to the DA2 external component.

2.1.2. Register Transfer Level (RTL) Diagram



The *counter* counts from 0 to 24, and counts for half of the intended period, i.e. 500 ns. Once the counter has reached its terminal count, it enables the *toggle_en* port of the *Toggle_Reg* and the *sclk* output toggles from high to low, or vise versa. The *fwd_clk* port uses the ODDR IP core provided by Xilinx to forward the clock to an external component.

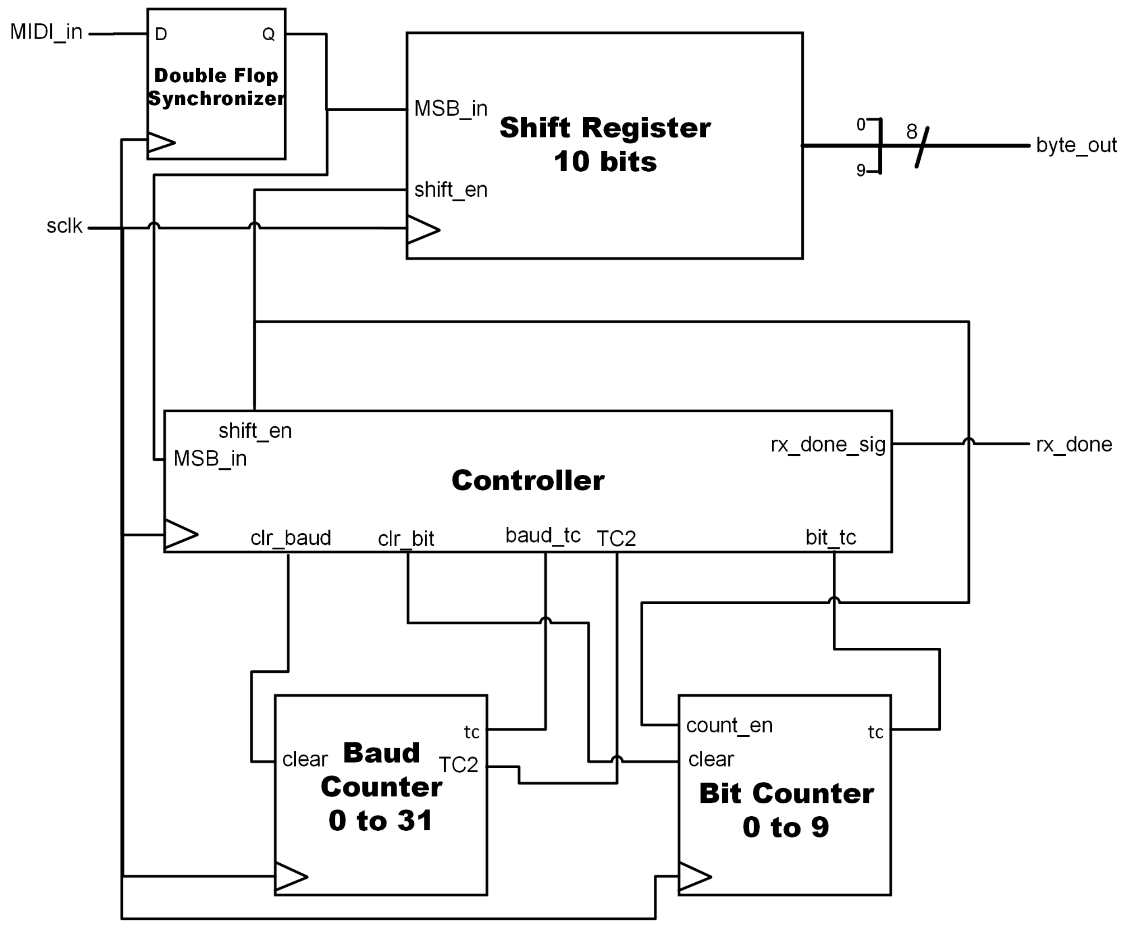
2.2. MIDI Receiver

The MIDI receiver takes the 10-bit serial data from the port MIDI_in, synchronizes it to the specified baud rate, and provides parallel out an 8-bit data_out signal to pipe into the datapath. Internally, it uses a shift register to store each bit serially, and asserts a signal once all the entire MIDI message byte has been received.

2.2.1. Description of Ports

- **External input** *MIDI_in* is a 1-bit serial data line from the midi controller into the system. The MIDI controller will send 3 bytes in succession per key event, each byte packaged between a low start bit and a high stop bit.
- **Input** *sclk* is the 1 MHz system clock generated by the clock generator block.
- **Output** *byte_out* is an 8-bit parallel output from the shift register, transferring all data bits (ignoring status bits) to the datapath of the system.
- **Output** *Rx_done* is a 1-bit control signal to the controller of the system, going high for one clock cycle when a byte has been transferred.

2.2.2. Register Transfer Level (RTL) Diagram



The *Double Flop Synchronizer* synchronizes the serial data stream from the MIDI_input to the system clock, creating a clocked serial stream *MSB_in* to the shift register and *MIDI_in* to the controller.

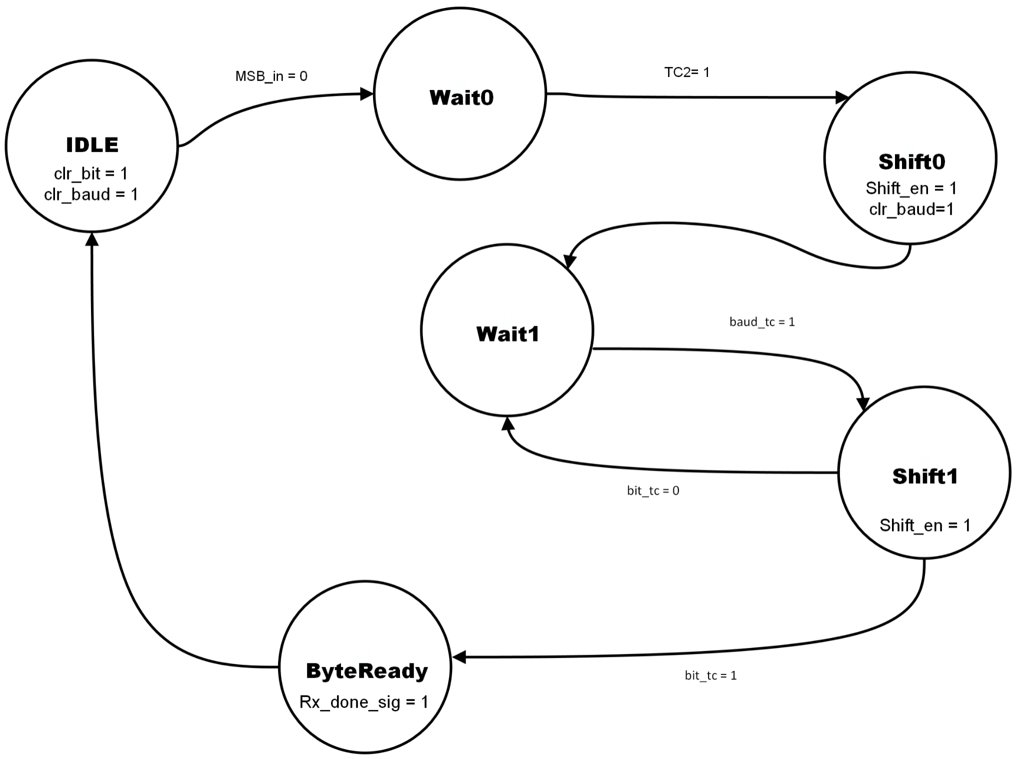
The *10-bit shift register* shifts in data when enabled by the controller, until all ten bits of a packet are shifted in. Then, it outputs the data bits (all but the MSB and LSB) to the datapath.

The *Baud Counter* ensures that the receiver looks for data on the 31.25kHz baud rate. It updates on every rising edge of the system clock, counting to 31 and enabling *baud_tc* to the controller at the terminal count. It also has an additional terminal count, *TC2*, for counting half of the baud rate. The baud counter is cleared by control signal *clr_baud* from the controller.

The *Bit Counter* is enabled by shift_en from the controller, counting 0 to 9 to alert the controller when 10 bits have been shifted in. It is cleared by *bit_clr* from the controller.

The Controller idles until Rx leaves the idle high state. It has control signals *tc_baud*, *baud_clr*, and *tc_bit*, *bit_clr* between itself and the two counters, respectively. *Shift_en*, going high for the first time when the controller reads a start ‘0’ bit from Rx, will tick 10 times, enabling the register to shift and the bit counter to increment. It outputs Rx_done to the system controller when 10 bits of Rx have been loaded into the register. See diagram below.

2.2.3. Finite State Machine (FSM) Diagram



This FSM diagram shows the expected behavior of the *Controller* component in the RTL diagram, including output signals and state transition logic. The controller waits in the IDLE state until it receives a '0' from the MIDI_in signal. It then waits half the baud period (*TC2*) so that the shifting occurs when the data is most stable. It then shifts all 10 bits into the shift register, and asserts that the MIDI receiver has successfully received a full byte of data.

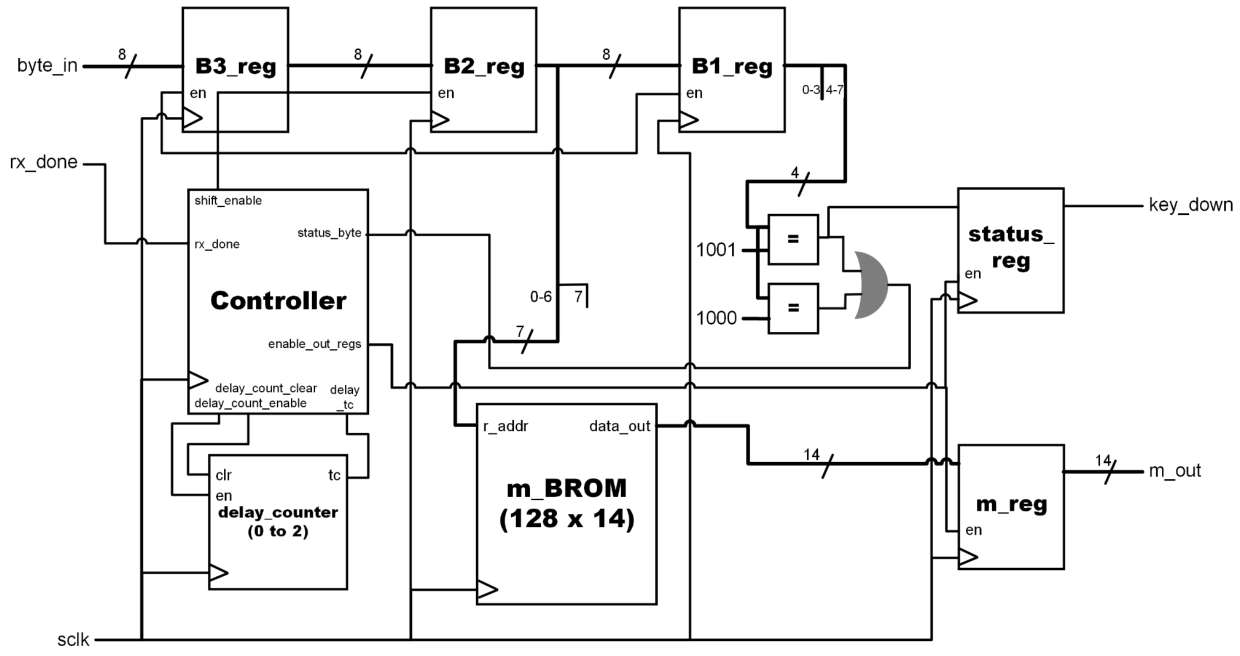
2.3. Datapath

The datapath receives bytes of data from the MIDI receiver one byte at a time, and stores these bytes in specified registers according to their type, whether it be a status byte, pitch byte, or a velocity byte. The datapath processes this data, and passes the necessary information to the DDS and DAC components.

2.3.1. Description of Ports

- **Input** *Byte_in* is a 8-bit parallel data line from the MIDI receiver.
- **Input** *s_clk* is the system clock generated by the clock generator block.
- **Input** *Rx_done* enables the registers to shift in new data,
- **Output** *m_out* is a 14-bit vector for controlling the frequency of the DDS.
- **Output** *key_down* is a 1-bit signal when the datapath has received a key_down message, and it controls whether there is a voltage output from the DAC.

2.3.2. Register Transfer Level (RTL) Diagram



The three connected registers, *B1_reg (Status)*, *B2_reg (Pitch)*, and *B3_reg (Velocity)*, act as a 3-byte buffer for the bytes arriving from the *MIDI Receiver*. These registers act together as a shift register for bytes, so the first received byte will be in the right-most register, and the second in the next right-most, and so on. The bytes shift through when the *shift_enable* control signal is high. *B2_reg* is attached to the *m_ROM* to index the pitch's associated *m* value. And, *B1_reg* connects to logic involving the status message. Currently, the receiver does not change its output according to the volume value, so there is nothing connected to *B3_reg*.

The *Controller* (FSM below) asserts control signals for data flow logic throughout the datapath. The input signals are *sclk*, *delay_tc*, *status_byte*, and the outputs signals are *delay_count_clear*, *delay_count_enable*, *status_byte*, *enable_out_regs*, and *shift_enable*.

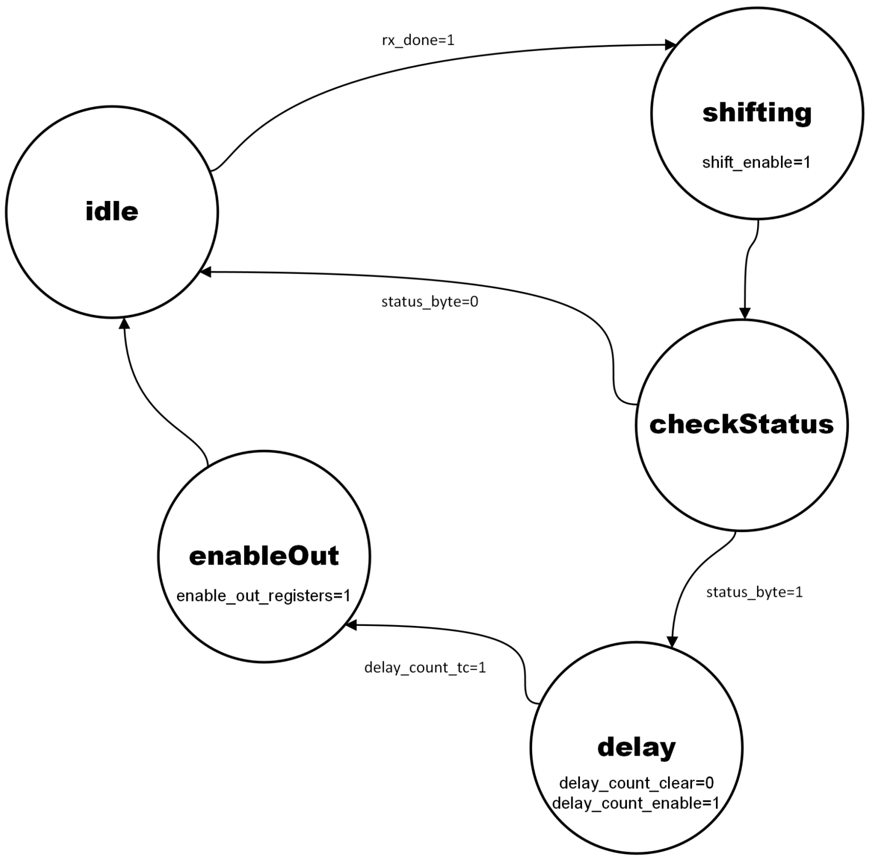
The *delay_counter* counts up to 3 clock signals to delay reading the value from the *m_BROM* until the propagation delay of 2 clock cycles has passed. The terminal count is asserted when it is done counting, telling the controller that the registers can update.

The *status_reg* updates its value whenever a 'key_up' or 'key_down' message is received, and takes the value '0' for 'key_up', and '1' for 'key_down.' All logic involving checking the status message is done using the two comparators to check the four least significant bits of *B3_reg*. The output of *status_reg* is tied to the *key_down* signal which is high when the datapath receives a 'key_down' message, and low when the datapath receives a 'key_up' message.

The *ROM* component serves as a lookup table of *m* values keyed by the *pitch*. The *pitch* input which is 7-bits is used to index into the ROM, and the output gives the 9-bit *m* value. There is a 2 clock-cycle propagation delay for reading a value from the *m_BROM* which had to be accounted for within the *Controller*.

The *m_reg* stores the current *m* value coming from the *m_ROM*, and the register is updated whenever a 'key_up' or 'key_down' status message is received.

2.3.3. Finite State Machine (FSM) Diagram



The controller clearly specifies the behavior of the datapath. The shifting state allows for the input byte to shift through the byte registers. At the checkStatus byte, the datapath checks whether the status byte represents a ‘key_up’ or ‘key_down’ message, continuing to shift bytes if not and loading the output signals if yes. The delay state is important because there is a 2 clock-cycle propagation delay on the *m_BROM*, so the controller must wait until loading the *m value* into the output register.

2.3.4. Memory

The memory in the datapath is a block of read-only-memory, 9-bit width and 7-bit depth. The read address is the 7 bit value stored in the 2nd byte of the MIDI message, representing the MIDI note value. The read data is a corresponding *m*, output to the Direct Digital Synthesis block, which holds the value by which the DDS counter will increment to create the correct frequency.

2.4. *DDS*

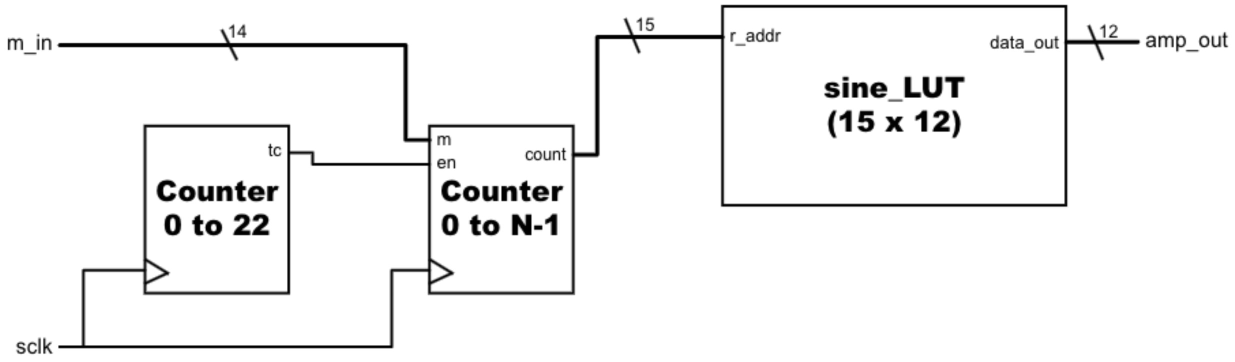
The Direct Digital Synthesis component converts the *m* value (derived from the MIDI pitch number) and outputs a sine wave at the correct frequency. The sample rate is approx. 44.1 kHz (de-facto 43.8kHz), typical for cd-quality audio. The sine wave amplitudes are sampled from a BROM lookup table which was provided by Xilinx. A counter which counts by *m* each time specifies the output frequency (higher *m* means a higher frequency).

2.4.1. Description of Ports

- **Input** *m_in* is a 14 bit signal from the Datapath that holds the value of the increment (max). This m value is higher for higher frequencies, so that the counter runs through the amplitudes more quickly.

- **Output** *amp_out* is a 12-bit signal to the Digital Analog Conversion block. It holds the value of the amplitude read from the Read Only Memory which stores the values of one period of a sine wave.

2.4.2. Register Transfer Level Diagram



The sample rate counter counts from 0 to 22, powered by the system clock. Our system clock is 1MHz, therefore to achieve a value close to the typical audio industry standard 44.1kHz, the best divider would be 23 (for a terminal count frequency 43,478Hz). The terminal count of this counter enables the address counter next to it.

The address counter takes *M_in* from the datapath, holding the 14 bit increment. The address counter moves from 0 to N-1, N being the number of samples in the LUT. We have selected a 15 bit address, therefore N is equal to 32,768. The counter is clocked so that it can update the value of m on every clock cycle, but it is only incrementing on the sample rate.

The sine_LUT is a read-only block memory, depth of 15 bits and width of 12 bits. To accommodate the PModDA2, which cannot produce negative voltages, the leading bit of the signed binary number output by the Sine LUT must be flipped. This is, in essence, adding 2048 to all of the values, translating the wave up so that the highest value is 4095 and the lowest value is zero.

Additionally, the IP core Sine LUT takes a 16 bit input and outputs 16 bits regardless of the settings, so in practice, an additional bit must be added to the input and the most significant 4 bits must be removed from the output of the memory.

2.4.3. Memory

The memory in the DDS block is a Read Only Memory that holds 32,768 (2^{15}) samples of the values of a sine wave. It takes a 15-bit read address from the counter, and spits out a 12-bit amplitude in two's complement. The formula for that value is $sin 2\pi \cdot k/N$, *k* being the address and *N* the total samples.

2.5. DAC interface

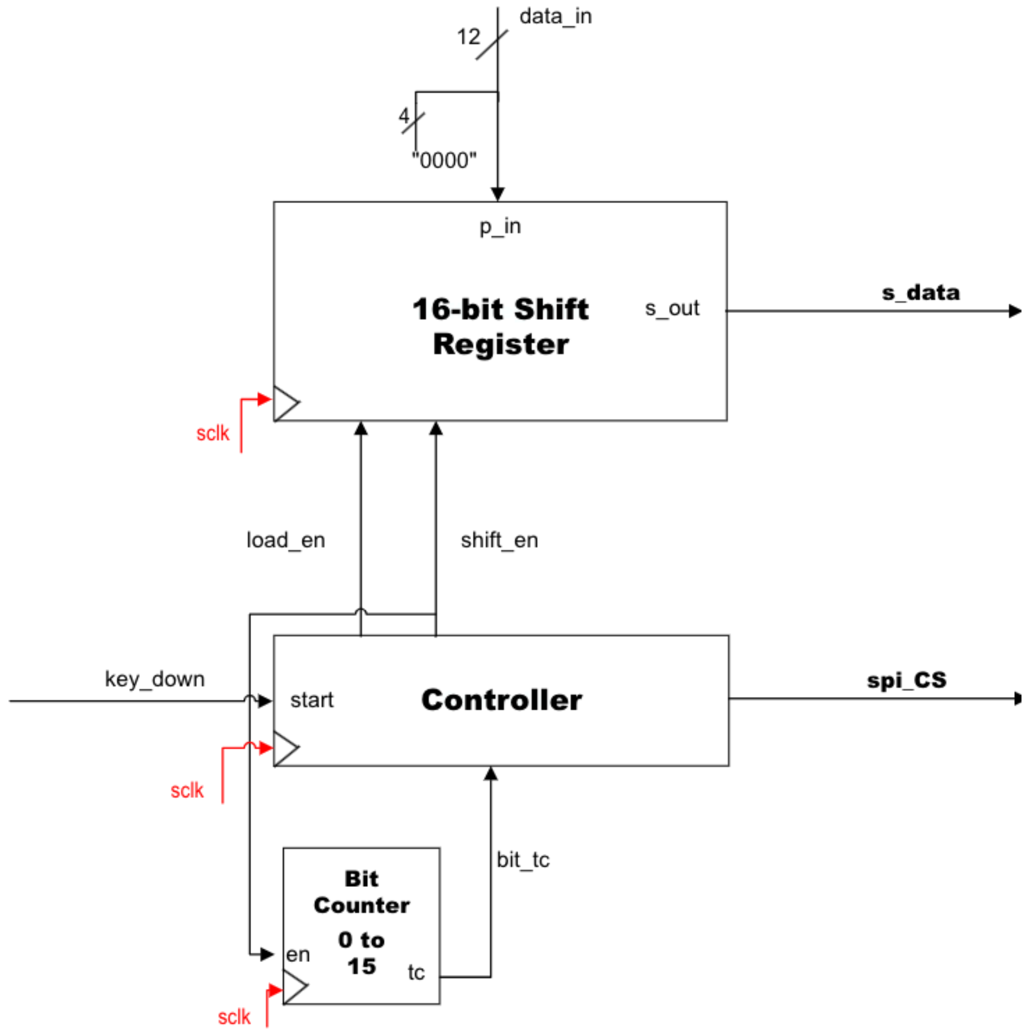
The DAC Interface controls communication from the FPGA to the external PMOD DA2 component. It is a serial peripheral interface (SPI). It consists of a parallel to serial shift register and a controller. The controller asserts CS' to the external DAC. The specific digital to analog protocol used in this design is 16 bits in big-endian with four leading zeroes.

2.5.1. Ports

- **Input** *sclk* is the 1 MHz system clock which comes from the clock generator

- **Input** *data_in* is a 12-bit signal from digital synthesis, holding the amplitude of a sine wave.
- **Input** *key_down* is a control bit from the datapath that asserts when a key is pressed down.
- **Output** *s_data* is serial data converted and shifted out one at a time in big-endian order.
- **Output** *spi_CS* is chip select, wired to DA2 and controlling when DA2 reads bits.

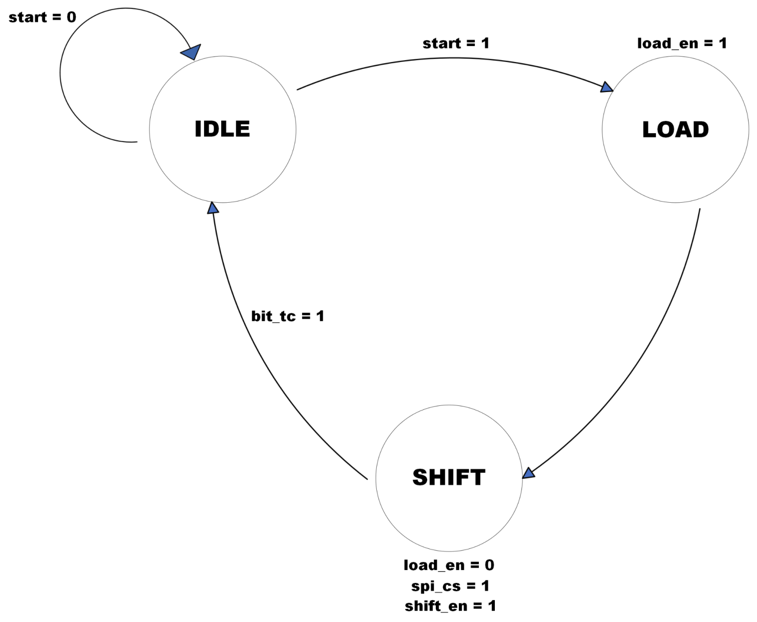
2.5.2. Register Transfer Level Diagram



The *shift register* is a parallel-to-serial converter. It takes 12 data bits with 4 leading zeroes on *load_en* from the controller. *Shift_en* is high for 16 clock cycles, allowing the bits (MSB first) to move to the *s_data* signal one at a time. *s_out* is tied to the MSB of the register.

The *bit counter* is enabled by the same *shift_en* from the controller that enables the register to shift a new bit onto the data line. It counts 16 bits before sending a terminal count control signal to signify that the controller should idle.

2.5.3. Finite State Machine



When in *IDLE* state, the controller is waiting for the *key_down* input into the *start* port to go high, signifying that a key down status byte has been processed in the datapath and a note should be played.

When in *LOAD* state, the controller enables the register to load a parallel 16 bits (4 leading zeroes, 12 data) on one clock cycle, then the controller moves to *SHIFT.*

When in *SHIFT* state, the controller sends the *spi_cs* high (the opposite shows up due to SPI protocol) to signify that it will move data onto the line to the converter. *Shift_en* is held high until the controller receives the *bit_tc* signal, which signifies that all 16 bits have been shifted, and the controller should return *IDLE*.

3. Design Validation

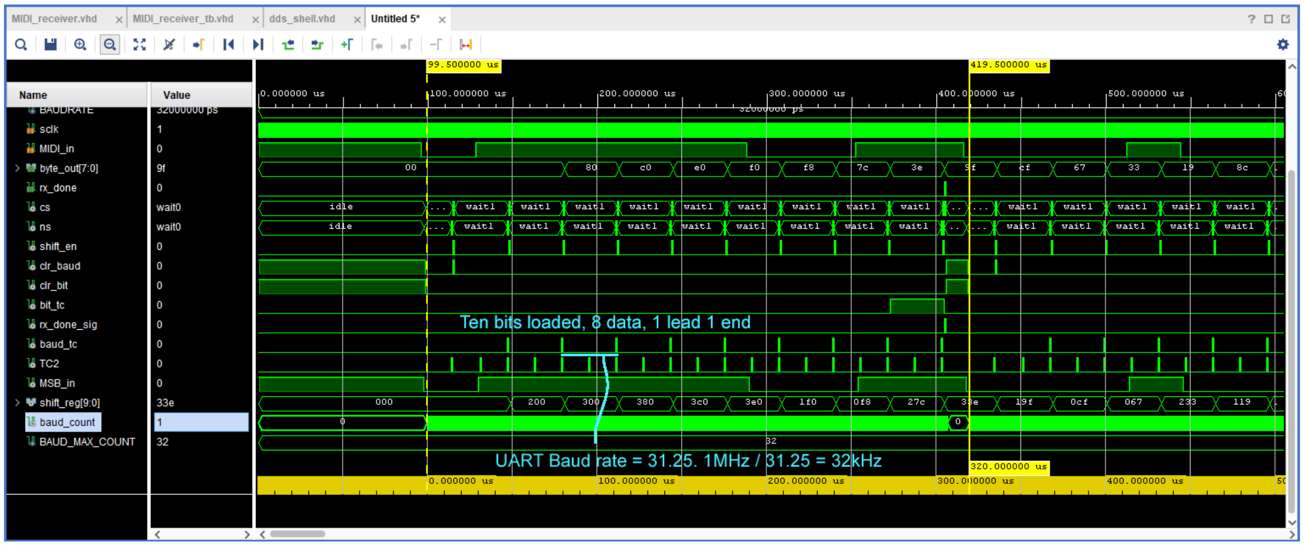
This section will be used to demonstrate successful implementation of the design. You should use a modular implementation and testing strategy. That means, each component should be verified through simulation and hardware testing (as appropriate) before it is combined with other components. This section should have a parallel structure to Section 2.

Note that if you need to go more than two levels deep (top level and components) in your testing strategy, then include a subsystem in your testing plan (See Section 3.3)

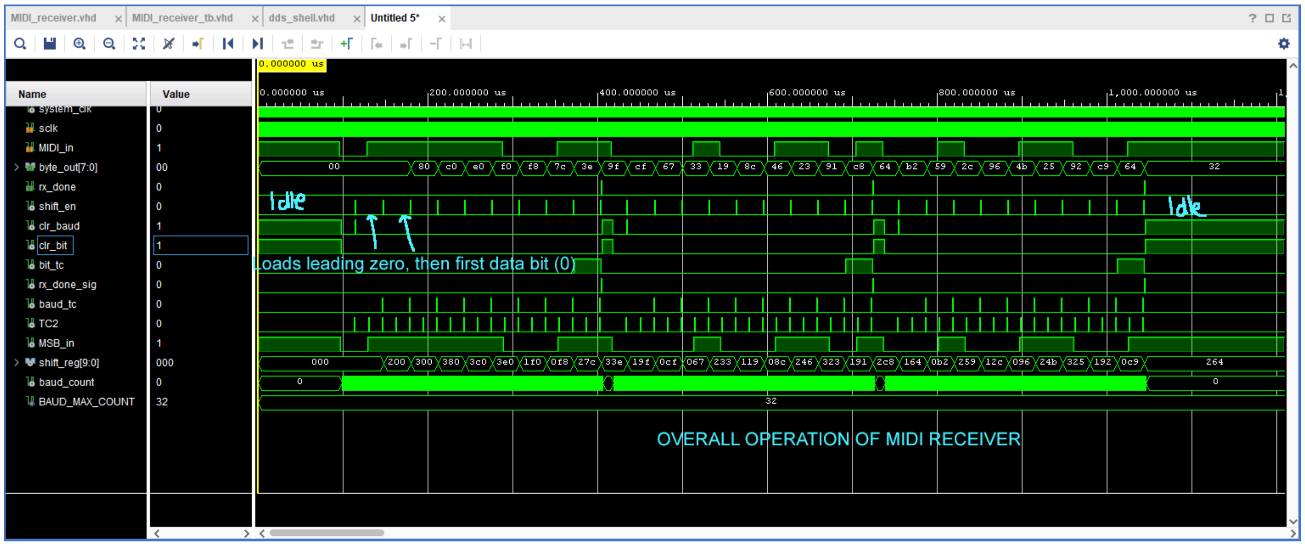
3.1. MIDI Receiver

With the MIDI receiver it is important to test the timing, bit order, and overall speed of the various components since it is so important to match the MIDI protocol exactly, and be aware of how quickly the rest of the system needs to respond to MIDI messages. When testing, we looked for whether data was synchronized to the clock, and whether it was shifted when the signal should be strongest – in the middle of the baud period.

3.1.1. Behavioral Simulations



MIDI Receiver: UART Baud rate, annotated

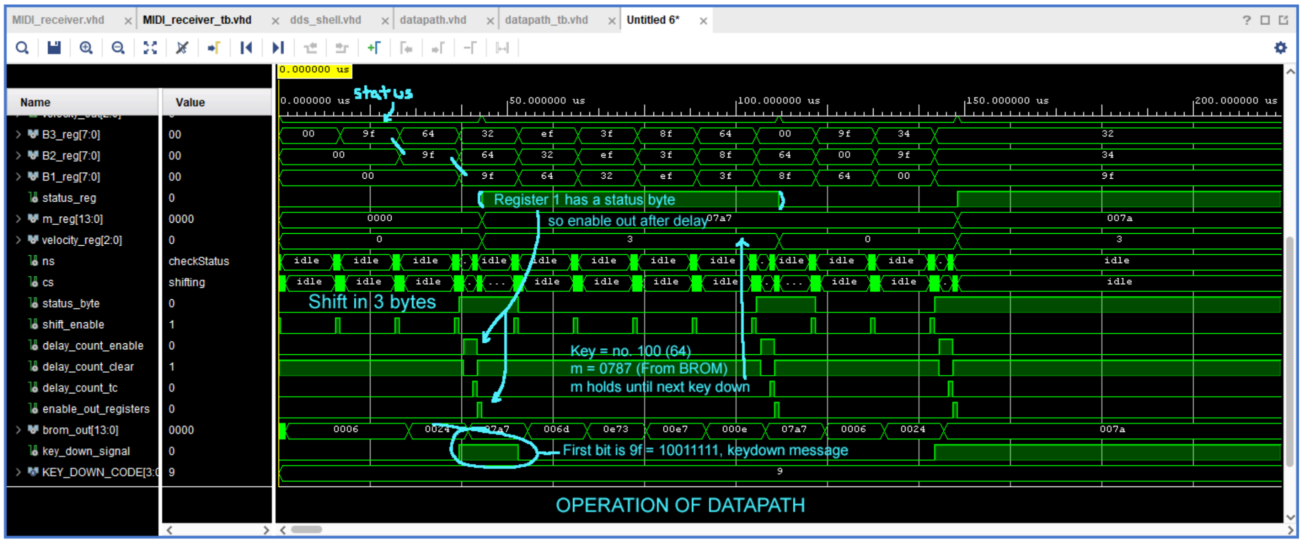

Idle
Idle
Loads leading zero, then first data bit (0)
OVERALL OPERATION OF MIDI RECEIVER

MIDI Receiver: Overall operation, annotated

3.2. Datapath

The datapath component must be tested for not only key_up and key_down messages, but also erroneous bytes such that it is robust against all MIDI messages. Correct operation is indicated by the correct outputs *m_out* and *key_down*. It was also important to test whether the correct *m* value was outputted according to the given 7-bit pitch value. Also, we paid attention to whether the *key_down* signal corresponded to whether a key was pressed or released.

3.2.1. Behavioral Simulations


status
Register 1 has a status byte
so enable out after delay
Shift in 3 bytes
Key = no. 100 (64)
m = 0787 (From BROM)
m holds until next key down
First bit is 9f = 10011111, keydown message
OPERATION OF DATAPATH

3.2.2. Hardware Validation (If Applicable)

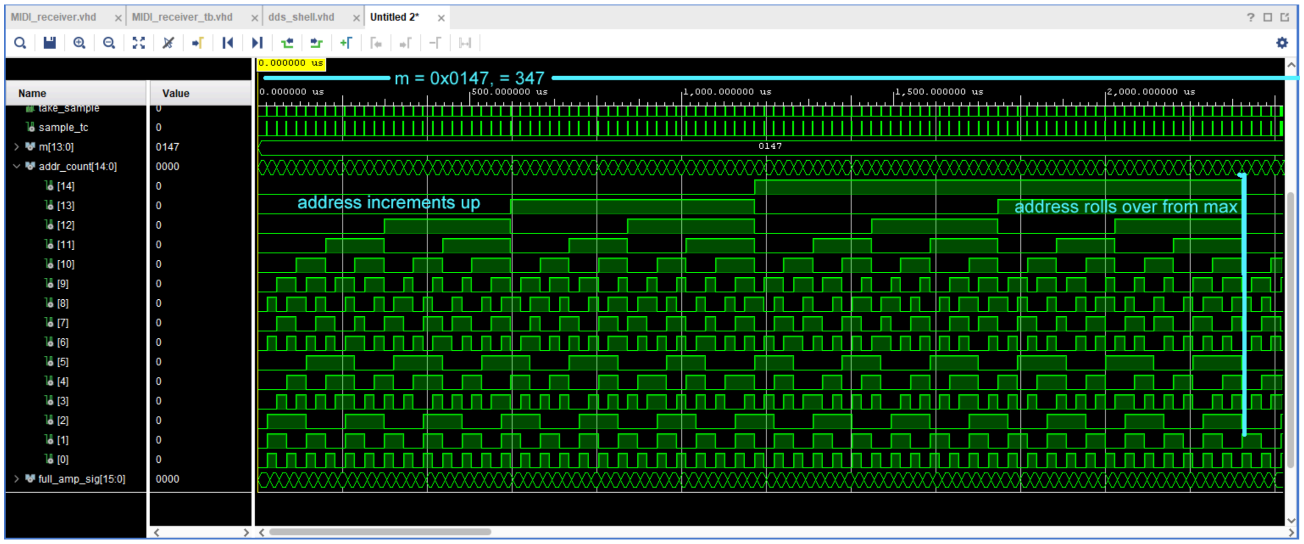
3.3. Direct Digital Synthesis & Digital to Analog Converter

Before testing the entire top level design in simulation and on hardware, we tested the functionality of the DDS & DAC components together to verify that they produced the appropriate output waves. Since there was no way to see an analog voltage in simulation, this subsystem was tested solely on FPGA. This test used a constant *m* value of 327 which should correspond with an output frequency of 440 Hz, or the note A4 on a typical piano scale. The test also kept the signal *key_down* high so that the frequency is always outputted.

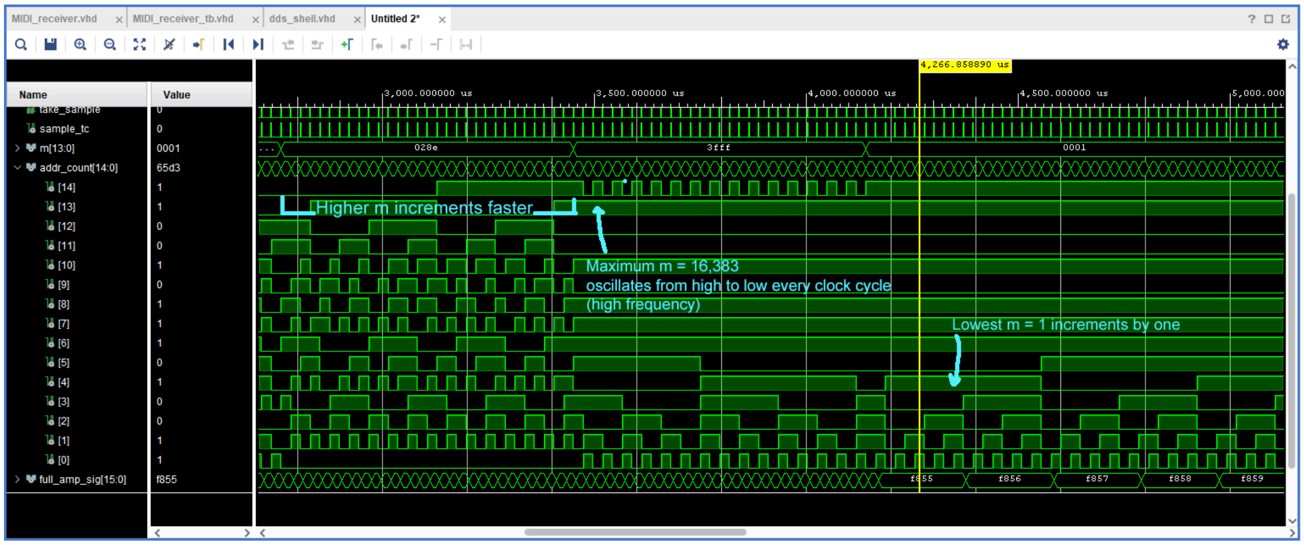
3.3.1. Direct Digital Synthesis

The Direct Digital Synthesis Block must be tested to show that the Sine LUT is generating correct values of a sine wave amplitude on the 12 bit output, and that the frequency of those outputs is accurately determined by the m value and the incrementer with a 15 bit output to the r_addr port on the ROM. We also want to ensure that the counter enabling the LUT address counter has a terminal count at a rate of 44.1kHz, our target sample rate.

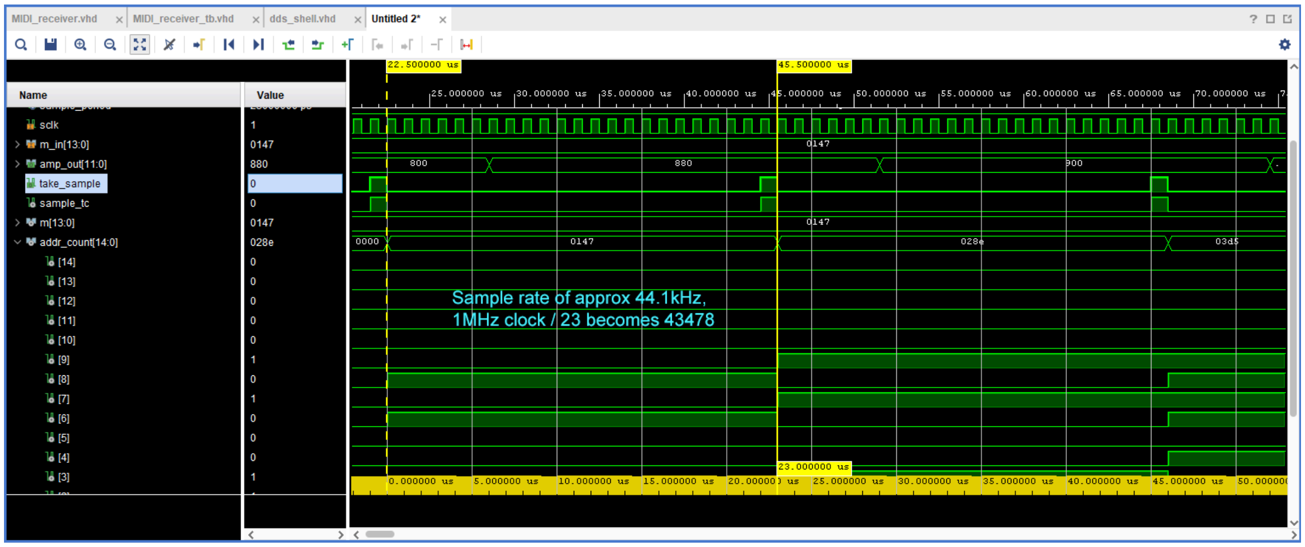
3.3.1.1. Behavioral Simulations



Direct Digital Synthesis: operation at a rate m = 347, annotated



Direct Digital Synthesis: operation at varying rates, max and min, annotated

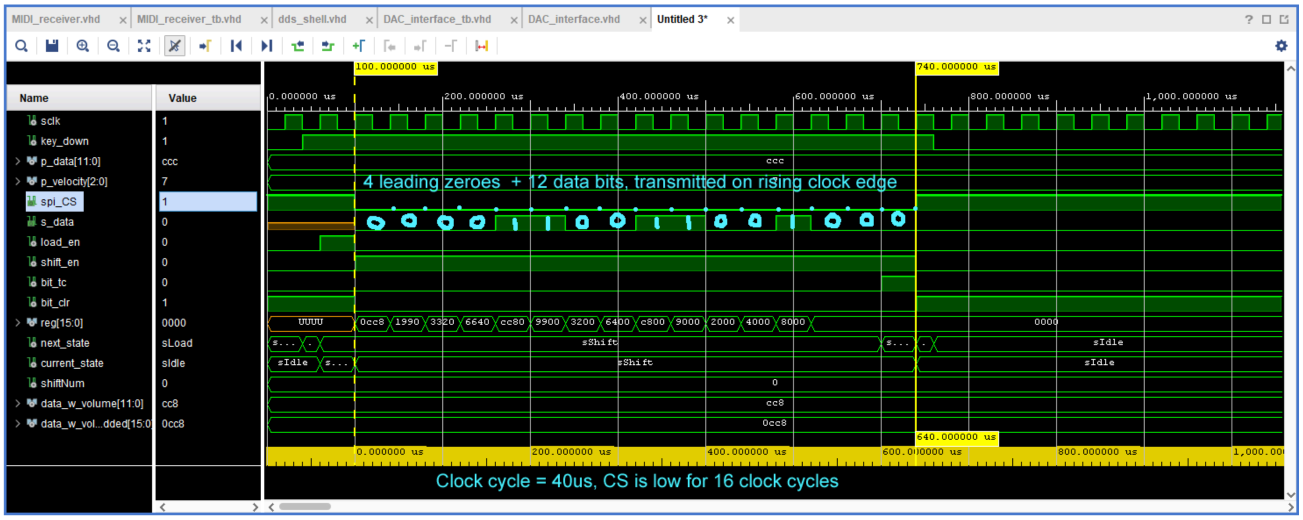


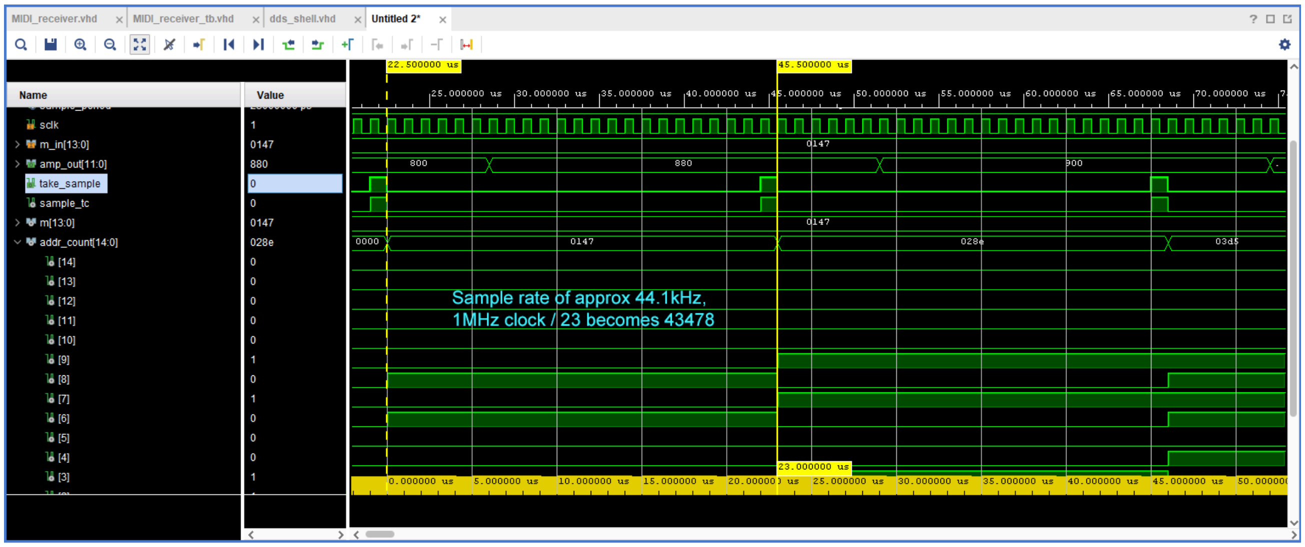
Direct Digital Synthesis: sample rate, annotated

3.3.2. DAC Interface

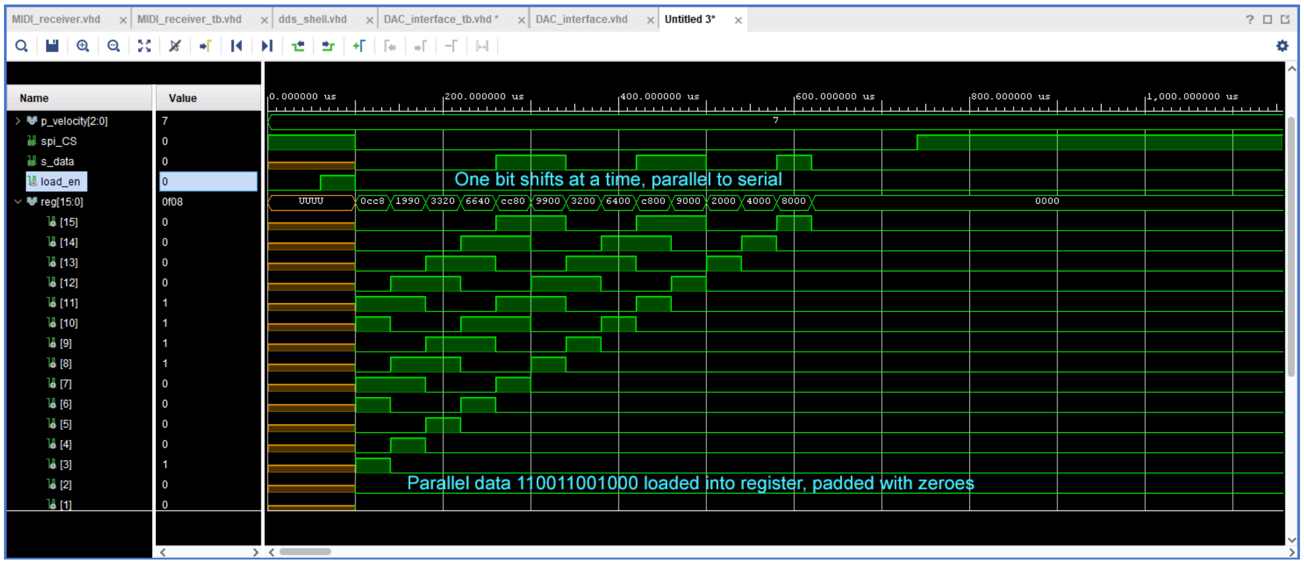
The DAC Interface must be tested to show that the timing of chip select to the PModDA2 is correct, allowing 16 bits to be transferred on the next 16 rising edges of *sclk* after *CS* goes low. Furthermore, the data transferred must be shown to be four leading zeroes (anything else risks entering a special power-down mode) followed by the 12 data bits, MSB first.

3.3.2.1. Behavioral Simulations

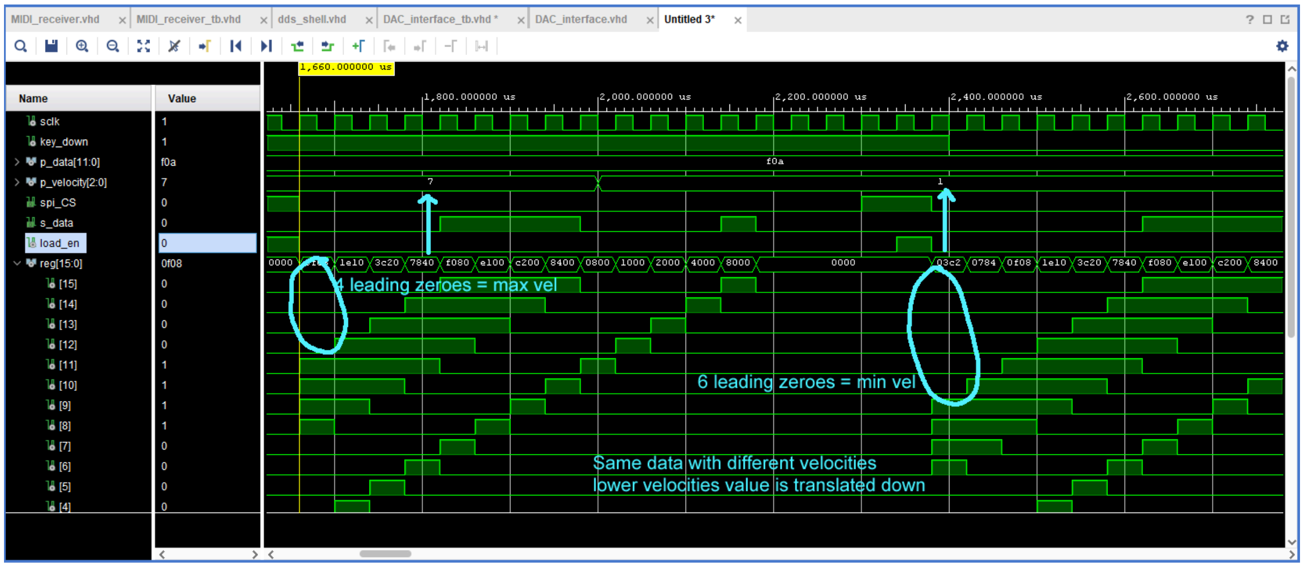




DAC Interface: One full cycle of data transmission

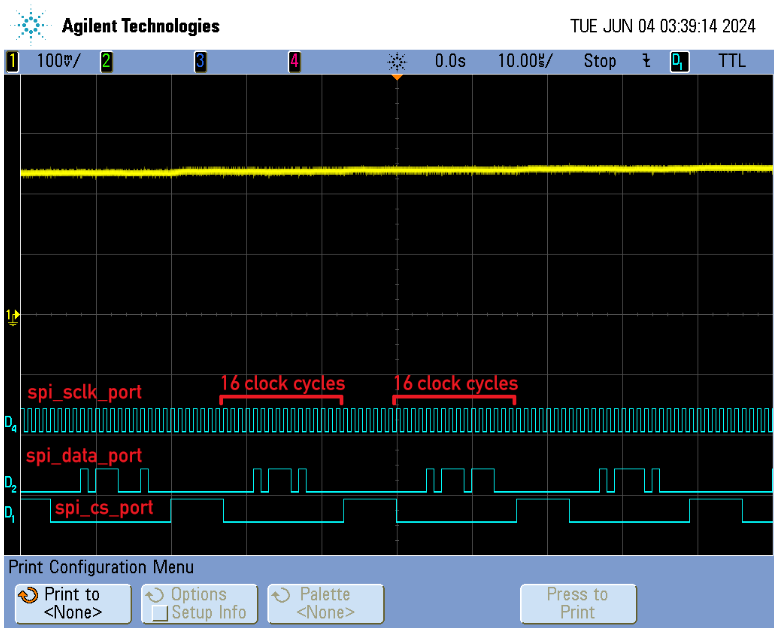

One bit shifts at a time, parallel to serial
Parallel data 110011001000 loaded into register, padded with zeroes
p_velocity[2:0]
spi_CS
s_data
load_en
reg[15:0]

DAC Interface: Parallel to serial conversion, annotated

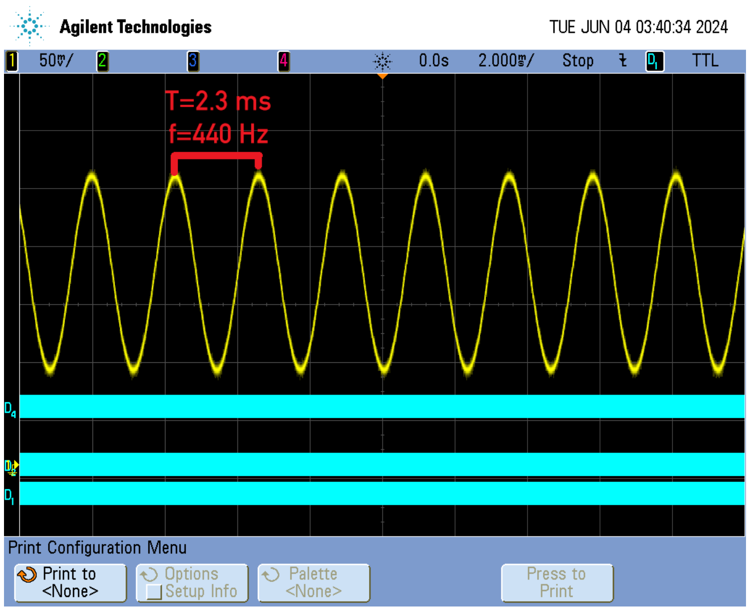

4 leading zeroes = max vel
6 leading zeroes = min vel
Same data with different velocities
lower velocities value is translated down

DAC Interface: Velocity difference with same note, annotated

3.3.3. Hardware Validation of DDS + DAC



Digital Output from DAC Component, Annotated

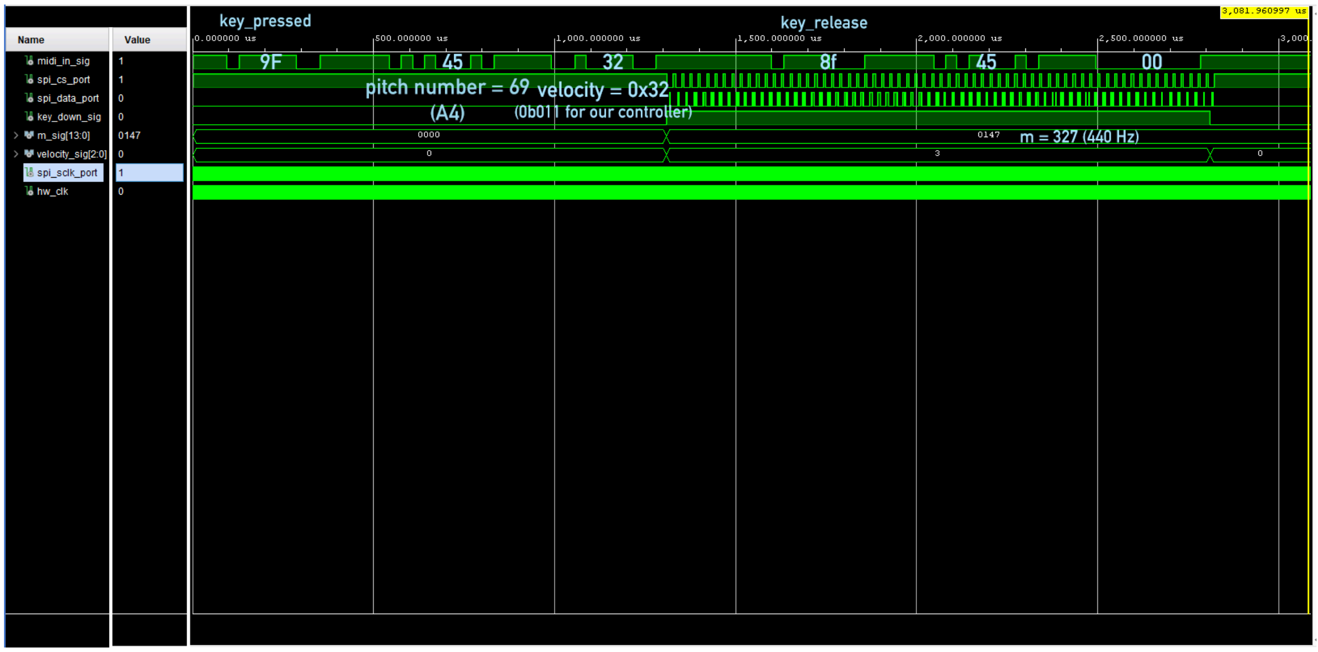


Analog Output of 440 Hz Sine Wave (note A4 on a piano), Annotated

3.4. Overall Design

Once you demonstrate that each individual component has proper operation, demonstrate the correct operation of the overall design.

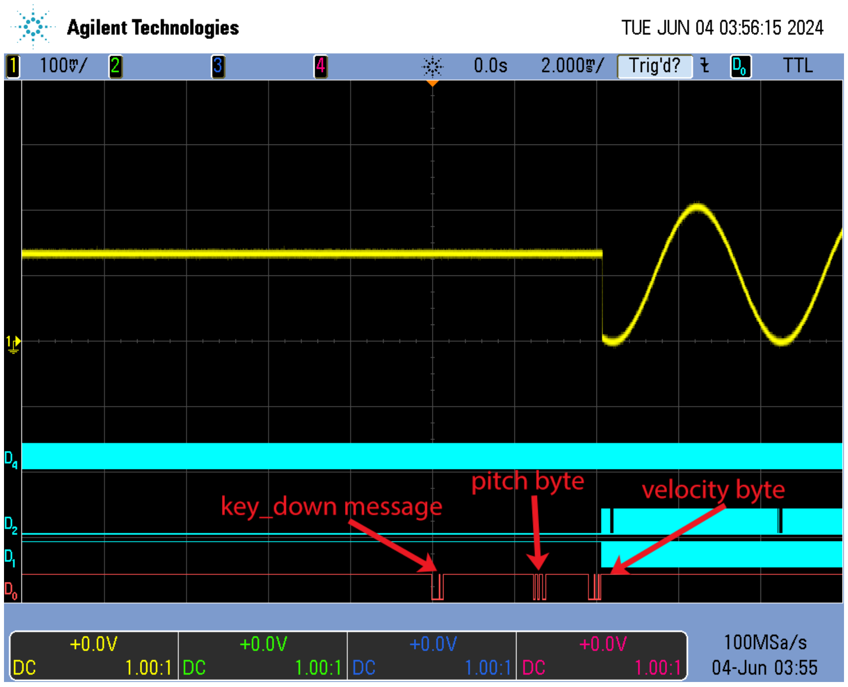
3.4.1. Behavioral Simulation



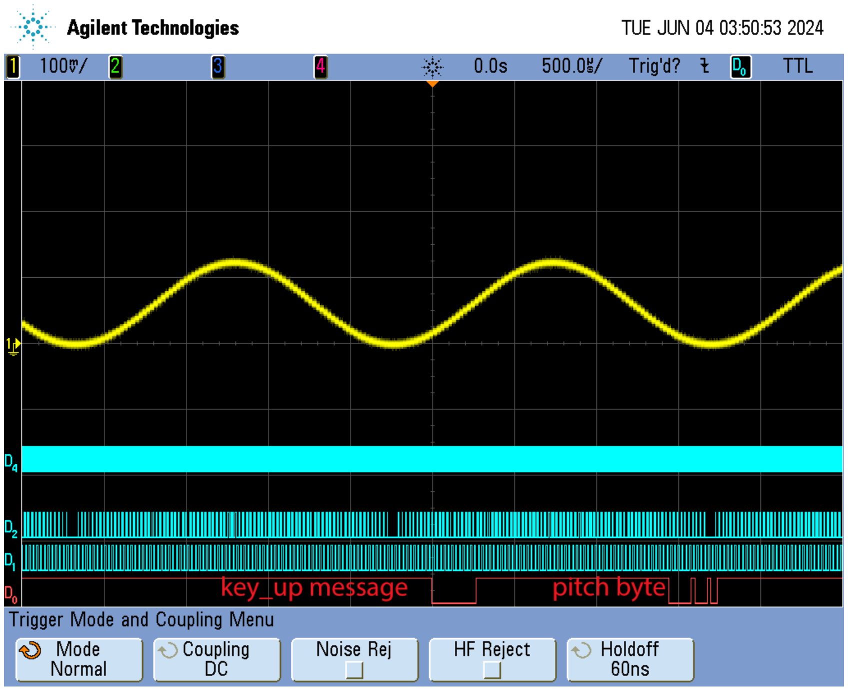
Name	Value
midi_in_sig	1
spi_cs_port	1
spi_data_port	0
key_down_sig	0
m_sig[13:0]	0147
velocity_sig[2:0]	0
spi_sclk_port	1
hw_clk	0

Simulation of Top Level Shell, Annotated

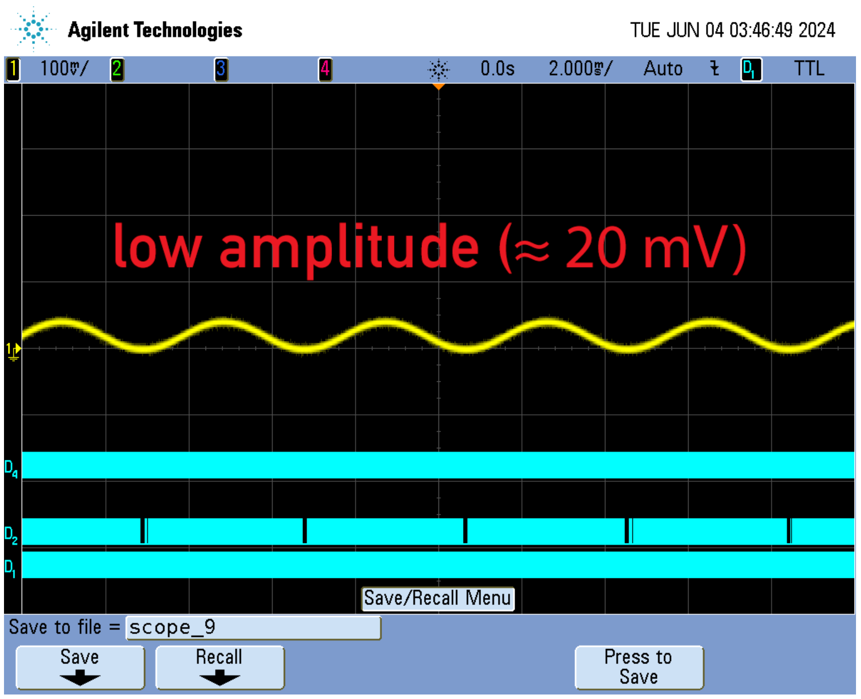
3.4.2. Hardware Validation



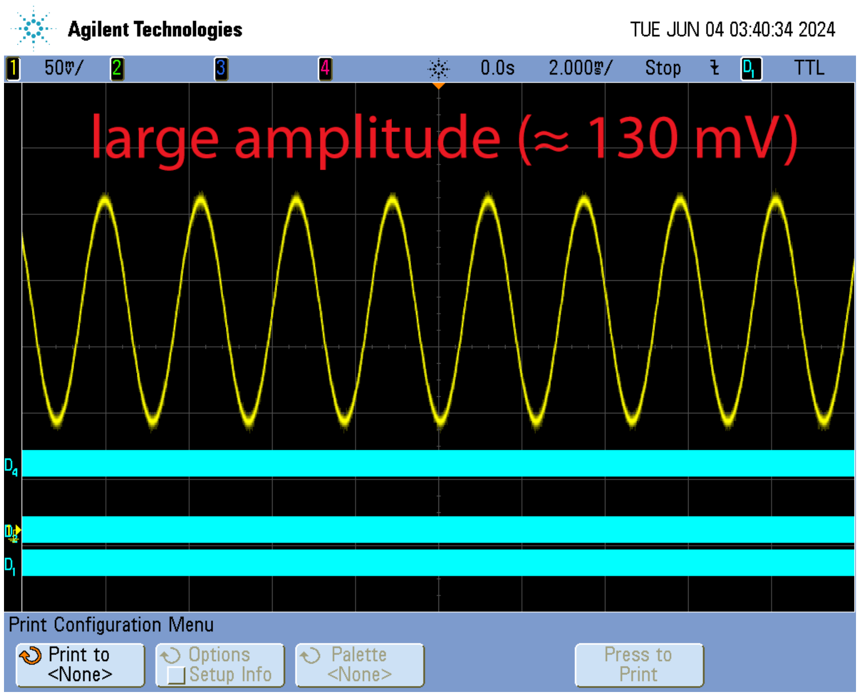
Key Press, Annotated



Key Release, Annotated



Low Velocity Keypress, Annotated



High Velocity Keypress, Annotated

4. Analysis of the Design

Overall, we were successful in our goal of creating a simple MIDI receiver which can handle a single key press at a time. We even exceeded our initial goal of supporting only a binary volume – on & off – by instead allowing for nine different volume levels including 'off.' Of course not everything went exactly as planned, but overall we had pretty smooth sailing mostly due to our component-by-component based approach. We found it productive to work on a component in isolation, write its testbench, and check that it passed all individual tests before including it in the top level shell. The most complex idea to tackle was that of The most frustrating bug was a simple VHDL error in which we had the *spi_cs_port* and the *spi_data_port* signals swapped in the top level port map, but eventually with the help of our TA, Wyatt, we fixed these mappings and our output sound transformed from noise to an actual note.

4.1. Resource Utilization

Our design used 141 total Slice LUTs which is 0.68% of the total capacity, and 254 slice registers which is 0.61% of the total capacity on the FPGA. It uses 5 Bonded IOBs which is a higher percentage of the total available resources on the FPGA at 4.72%. And our largest utilized resource as a percentage of total availability is 2 BUFGCTRLs which is 6.25% of the available resources. Based on the utilization report, it is clear that our design uses a very small percentage of the total resources on the FPGA. The availability of additional resources on the board gives us hope that implementing a multiple-key controller is feasible for this specific FPGA.

4.2. Residual Warnings

- synth_1 (3 warnings)
    - *Reference run did not run incremental synthesis because the design is too small; reverting to default synthesis (1 more like this)*
    - *Parallel synthesis criteria is not met*
    - Both of these warnings can be ignored as they exist because our design is relatively small, and therefore does not need special optimization steps from Vivado.
- dds_compiler_1_synth_1 (109 warnings)
    - *Port 'debug_axi_pinc_in' is missing in component declaration [dds_compiler_1.vhd: 72] (6 more like this)*
    - *Port CLK in module xbip_pipe_v3_0_6_viv_parameterized12 is either unconnected or has no load (99 more like this)*
    - *Register i_rt.i_quarter_table.i_has_sin.i_addr_mod_stage1.mod_sin_addr_reg driving address of a ROM cannot be packed in BRAM/URAM because of presence of initial value.*
    - *Parallel synthesis criteria is not met*
    - We can safely ignore these warnings because they are from the synthesis of an IP core which we did not actually write, and the warnings stem from the fact that the DDS_compiler block has many extra features and ports which we are not using in our design.

4.3. Division of Labor

We split the work and time evenly amongst ourselves. After designing the top level RTL for the controller together, we were able to tackle components individually without fear of incompatibility between components. Thus, once each component passed its testbench in simulation, it was safe to say that it would work in the top level shell. We used Github to host all code and other files, and Microsoft Visio for collaborating on RTL & FSM designs. Overall, our main goal was for each person to fully understand each minor component in the overall design, one that was certainly met.

4.4. Future Work

The next steps would be to implement multiple key presses so that a user can play chords. We had already started implementing this feature but have more bugs to squash and therefore elected not to include it in our report. Another minor design detail which nagged us was an audible "click" whenever a key was pressed down. We hypothesized that this noise was due to the analog voltage snapping from ground to a random position on the sine wave. A future update could work out how to return the voltage to a consistent resting level, and ensure that when a key is pressed, the amplitude begins at that same level.

5. Acknowledgements

Additional information on MIDI notes and frequencies:
- https://homes.luddy.indiana.edu/donbyrd/Teach/MusicalPitchesTable.htm

Pmod DA2 Technical Specifications:
- https://www.ti.com/lit/ds/symlink/dac121s101.pdf

Another helpful page about the MIDI protocol:
- https://www.cs.cmu.edu/~music/cmsip/readings/MIDI%20tutorial%20for%20programmers.html

6. Conclusions

Thus concludes our final report of the MIDI controller project. Our overall experience with this specific project was overwhelmingly positive, and it was rewarding to play around with the final result. We found that though we had hit our original benchmarks, we were intrigued by the idea of developing the design further and tried to implement features like multiple keys. In the end, we unfortunately ran out of time, but this process has left us ready to keep experimenting with digital design. We would recommend this project to future students, especially those with interest in music or DA/AD conversion. Make sure to check even the silly places in your design for errors! That's where ours were.

Appendix A: VHDL Source Code

1. SYSTEM CLOCK GENERATION - BEHAVIORAL

```
--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 system_clock_generation is
  Port (
    --External Clock:
      hw_clk                  : in std_logic;
    --System Clock:
      sclk                    : out std_logic;
    --Forward Clock
      fwd_clk : out std_logic);
end system_clock_generation;

--==============================================================================
--Architecture Type:
--==============================================================================
architecture behavioral_architecture of system_clock_generation is
--==============================================================================
--Signal Declarations:
--==============================================================================
constant CLOCK_DIVIDER_TC: integer := 50;

--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';
signal system_clk_sig    : std_logic := '0';

--=============================================================================
--Processes:
--=============================================================================
begin
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Clock (frequency) Divider):
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Clock_divider: process(hw_clk)
begin
        if rising_edge(hw_clk) 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 system clock onto the FPGA clocking network
Slow_clock_buffer: BUFG
   port map (I => system_clk_tog,
             O => system_clk_sig);
   sclk <= system_clk_sig;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-- Clock Forwarding
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

clock_forwarding_ODDR : ODDR
generic map(
        DDR_CLK_EDGE => "SAME_EDGE", -- "OPPOSITE_EDGE" or "SAME_EDGE"
        INIT => '0', -- Initial value for Q port ('1' or '0')
        SRTYPE => "SYNC") -- Reset Type ("ASYNC" or "SYNC")
port map (
        Q => fwd_clk, -- 1-bit DDR output
        C => system_clk_sig, -- 1-bit clock input
        CE => '1', -- 1-bit clock enable input
        D1 => '1', -- 1-bit data input (positive edge)
        D2 => '0', -- 1-bit data input (negative edge)
        R => '0', -- 1-bit reset input
        S => '0' -- 1-bit set input
);
end behavioral_architecture;
```

2. MIDI RECEIVER - BEHAVIORAL

```
--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_receiver is
```

```
  Port (
    -- inputs
    -- 1MHz clock
    sclk : in std_logic;
    -- serial midi bit
    MIDI_in : in std_logic;

    -- outputs
    -- byte of data
    byte_out : out std_logic_vector(7 downto 0);
    -- done receiving signal
    rx_done   : out std_logic);
end MIDI_receiver;

--=============================================================
--Architecture + Component Declarations
--=============================================================
architecture Behavioral of MIDI_receiver is
  component counter is
    generic (
      MAX_COUNT : integer);
    port (
      --timing
      clk   : in std_logic;
      -- sync clear port
      clr   : in std_logic;
      -- enable counting
      en    : in std_logic;
      tc    : out std_logic);
  end component;

--=============================================================
--Local Signal Declaration
--=============================================================

-- controller
type state_type is (idle, wait0, shift0, wait1, shift1, byte_ready);
signal cs, ns : state_type := idle;
-- control signals
signal shift_en, clr_baud, clr_bit, bit_tc, rx_done_sig, baud_tc, TC2 : std_logic := '0';

-- shift register
signal MSB_in : std_logic := '1'; -- sanitized input
signal shift_reg : std_logic_vector(9 downto 0) := (others => '0');

-- intermediate flip flop signal
signal inputFF : std_logic := '1';

-- baud_counter
signal baud_count : integer := 0;
constant BAUD_MAX_COUNT : integer := 32;

--=============================================================
--Port Mapping + Processes:
--=============================================================
begin
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Bit Counter:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
bit_counter : counter
generic map (
  MAX_COUNT => 10)
port map(
  clk => sclk,
  clr => clr_bit,
  en => shift_en,
  tc => bit_tc);

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Baud Counter:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
baud_count_logic : process(sclk, baud_tc, clr_baud)
```

```
begin
  if rising_edge(sclk) then
    baud_count <= baud_count + 1;
    if baud_tc = '1' or clr_baud = '1' then
      baud_count <= 0;
    end if;
  end if;
end process baud_count_logic;

baud_tc_logic : process(baud_count)
begin
  baud_tc <= '0';
  TC2 <= '0';
  if baud_count = BAUD_MAX_COUNT / 2 - 1 then
    TC2 <= '1';
  elsif baud_count = BAUD_MAX_COUNT - 1 then
    TC2 <= '1';
    baud_tc <= '1';
  end if;
end process baud_tc_logic;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Update State:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
update_state : process(sclk, ns)
begin
  if rising_edge(sclk) then
    cs <= ns;
  end if;
end process update_state;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Next State Logic:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
next_state_logic : process(cs, MSB_in, baud_tc, bit_tc, TC2)
begin
  ns <= cs;
  case cs is
    when idle =>
      if MSB_in = '0' then
        ns <= wait0;
      end if;
    when wait0 =>
      if TC2 = '1' then
        ns <= shift0;
      end if;
    when shift0 => ns <= wait1;
    when wait1 =>
      if baud_tc = '1' then
        ns <= shift1;
      end if;
    when shift1 =>
      if bit_tc = '1' then
        ns <= byte_ready;
      else
        ns <= wait1;
      end if;
    when byte_ready => ns <= idle;
  end case;
end process next_state_logic;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Output Logic:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
output_logic : process(cs)
begin
  clr_bit <= '0';
  shift_en <= '0';
  rx_done_sig <= '0';
  clr_baud <= '0';
  case cs is
    when idle => clr_bit <= '1'; clr_baud <= '1';
    when shift0 => shift_en <= '1'; clr_baud <= '1';
    when shift1 => shift_en <= '1';
    when byte_ready => rx_done_sig <= '1';
    when others =>
  end case;
end process output_logic;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Shift Register Logic:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
shift_reg_logic : process(sclk, MSB_in, shift_en)
begin
  if rising_edge(sclk) then
    if shift_en = '1' then
      shift_reg <= MSB_in & shift_reg(9 downto 1);
    end if;
  end if;
end process shift_reg_logic;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Synchronizer Logic:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
sanitize_input_logic : process(sclk)
begin
  if rising_edge(sclk) then
    inputFF <= MIDI_in;
    MSB_in <= inputFF;
  end if;
end process sanitize_input_logic;

-- tie pins to outputs
byte_out <= shift_reg(8 downto 1);
rx_done <= rx_done_sig;

end Behavioral;
```

3. DATAPATH - BEHAVIORAL

```
--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 datapath is
  port (
    sclk    : in  std_logic;
    byte_in : in  std_logic_vector(7 downto 0);
    rx_done : in  std_logic;
    key_down : out std_logic;
    m_out   : out std_logic_vector(13 downto 0);
    velocity_out : out std_logic_vector(2 downto 0)
  );
end datapath;

--===============================================================
--Architecture + Component Declarations
--===============================================================
architecture behavioral of datapath is

-- block rom for m values
component datapath_m_BROM is
  port (
    clka : IN STD_LOGIC;
    addra : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(13 DOWNTO 0)
  );
end component;

component counter is
  generic (
    MAX_COUNT : integer
  );
  port (
    clk : in std_logic;
    clr : in std_logic;
    en  : in std_logic;
    tc  : out std_logic
  );
end component;

--=============================================================
--Local Signal Declaration
--=============================================================

-- registers
signal B3_reg : std_logic_vector(7 downto 0) := (others => '0');
signal B2_reg : std_logic_vector(7 downto 0) := (others => '0');
signal B1_reg : std_logic_vector(7 downto 0) := (others => '0');
signal status_reg : std_logic := '0';
signal m_reg : std_logic_vector(13 downto 0) := (others => '0');
signal velocity_reg : std_logic_vector(2 downto 0) := (others => '0');

-- controller
type state_type is (idle, shifting, checkStatus, delay, enableOut);
signal ns, cs : state_type := shifting;

-- control signals
signal status_byte : std_logic := '0';
signal shift_enable : std_logic := '0';
signal delay_count_enable : std_logic := '0';
signal delay_count_clear : std_logic := '1';
signal delay_count_tc : std_logic := '0';
signal enable_out_registers : std_logic := '0';

-- signals to output regs
signal brom_out : std_logic_vector(13 downto 0) := (others => '0');
signal key_down_signal : std_logic := '0';

-- constants
constant KEY_DOWN_CODE : std_logic_vector(3 downto 0) := "1001";
constant KEY_UP_CODE   : std_logic_vector(3 downto 0) := "1000";

--=============================================================
--Port Mapping + Processes:
--=============================================================
begin

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Block ROM:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
brom : datapath_m_BROM
  port map (
    clka => sclk,
    addra => B2_reg(6 downto 0),
    douta => brom_out);

delay_counter : counter
  generic map (
    MAX_COUNT => 3
           )
  port map (
    clk => sclk,
    en => delay_count_enable,
    clr => delay_count_clear,
    tc => delay_count_tc);
```

```
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Update State:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
update_state : process (sclk, ns)
begin
  if rising_edge(sclk) then
    cs <= ns;
  end if;
end process update_state;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Next State Logic:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
next_state_logic : process(cs, rx_done, status_byte, delay_count_tc)
begin
  ns <= cs;
  case cs is
    when idle =>
      if rx_done = '1' then
        ns <= shifting;
      end if;
    when shifting => ns <= checkStatus;
    when checkStatus =>
      if status_byte = '1' then
        ns <= delay;
      else
        ns <= idle;
      end if;
    when delay =>
      if delay_count_tc = '1' then
        ns <= enableOut;
      end if;
    when enableOut => ns <= idle;
  end case;
end process next_state_logic;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Output Signal Logic
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
output_signal_logic : process(cs)
begin
  delay_count_clear <= '1';
  delay_count_enable <= '0';
  enable_out_registers <= '0';
  shift_enable <= '0';
  case cs is
    when shifting => shift_enable <= '1';
    when delay => delay_count_enable <= '1'; delay_count_clear <= '0';
    when enableOut => enable_out_registers <= '1';
    when others =>
  end case;
end process output_signal_logic;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Logic for Shifting Bytes through Registers:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
byte_shift_logic : process(sclk, rx_done, byte_in, B3_reg, B2_reg, B1_reg)
begin
  if rising_edge(sclk) then
    if shift_enable = '1' then
      B3_reg <= byte_in;
      B2_reg <= B3_reg;
      B1_reg <= B2_reg;
    end if;
  end if;
end process byte_shift_logic;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Logic Output Registers
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
output_reg_logic : process(sclk, enable_out_registers, brom_out, key_down_signal)
begin
```

```
  if rising_edge(sclk) then
    if enable_out_registers = '1' then
      status_reg <= key_down_signal;
      m_reg <= brom_out;
      velocity_reg <= B3_reg(6 downto 4); -- 3 MSBs
    end if;
  end if;
end process output_reg_logic;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Async Status Signals Logic:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
status_signals_logic : process(B1_reg)
begin
  key_down_signal <= '0';
  status_byte <= '0';
  if B1_reg(7 downto 4) = KEY_UP_CODE then
    status_byte <= '1';
  elsif B1_reg(7 downto 4) = KEY_DOWN_CODE then
    status_byte <= '1';
    key_down_signal <= '1';
  end if;
end process status_signals_logic;

-- tie registers to output ports
m_out <= m_reg;
key_down <= status_reg;
velocity_out <= velocity_reg;

end behavioral;
```

4. DDS - BEHAVIORAL

```
--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 DDS is
  port (
    sclk    : in  std_logic;
    m_in    : in  std_logic_vector(13 downto 0);
    amp_out : out std_logic_vector(11 downto 0);
    take_sample : out std_logic
    );
end DDS;

--=============================================================
--Architecture + Component Declarations
--=============================================================
architecture Behavioral of DDS is
  COMPONENT counter is
    generic (
      MAX_COUNT : integer);
    port (
      --timing
      clk   : in std_logic;
      -- sync clear port
      clr   : in std_logic;
      -- enable counting
      en    : in std_logic;
      tc    : out std_logic);
  END COMPONENT;
  COMPONENT dds_compiler_1
  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;

--=============================================================
--Local Signal Declaration
--=============================================================
signal sample_tc : std_logic; --Terminal count sample rate counter
signal m        : unsigned(13 downto 0);
signal addr_count : unsigned(14 downto 0) := "000000000000000";
signal full_amp_sig : std_logic_vector(15 downto 0);  --LUT gives a 16 bit signal, will take first 12 bits to DAC
signal padded_addr : std_logic_vector(15 downto 0);

--=============================================================
--Port Mapping + Processes:
--=============================================================
begin
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Sample rate Counter:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
sample_counter : counter
generic map (
  MAX_COUNT => 23)
port map(
  clk => sclk,
  clr => '0', --Maybe tie to something else
  en => '1', --Maybe tie to something else
  tc => sample_tc);

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--LUT
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Sine_LUT : dds_compiler_1
  PORT MAP (
    aclk => sclk,
    s_axis_phase_tvalid => '1', --Enable input
    s_axis_phase_tdata => padded_addr,
    m_axis_data_tvalid => open,
    m_axis_data_tdata => full_amp_sig
  );
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--LUT Address Counter: (unsigned adder that will rollover)
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
addr_count_logic : process(sclk, sample_tc, m)
begin
  if rising_edge(sclk) then
    if sample_tc = '1' then
     addr_count <= addr_count + m;           --Increment by M
    end if;
  end if;
end process addr_count_logic;
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Asynchronous
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
amp_out <= not full_amp_sig(11) & full_amp_sig(10 downto 0);  --Tie first 12 to amplitude output
m <= unsigned(m_in);                    --M tied to M input, made unsigned
take_sample <= sample_tc;

padded_addr <= '0' & std_logic_vector(addr_count);

end Behavioral;
```

5. DAC INTERFACE - BEHAVIORAL

```
--Library
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 DAC_interface is
  Port (

    --timing:
    sclk                        : in std_logic;

    --:
    key_down          : in std_logic; --signal that key has been pressed
    data_in          : in std_logic_vector(11 downto 0);
    velocity_in         : in std_logic_vector(2 downto 0);
    take_sample         : in std_logic; -- signal for 44 kHz sampler


    --outputs
    spi_CS           : out std_logic;
    s_data                   : out std_logic);

end DAC_interface;

--==============================================================================
--Architecture Type:
--==============================================================================
architecture behavioral_architecture of DAC_interface is
--==============================================================================
--Component Declarations:
--==============================================================================
component counter is
    generic (
      MAX_COUNT : integer);
    port (
      --timing
      clk   : in std_logic;
      -- sync clear port
      clr   : in std_logic;
      -- enable counting
      en    : in std_logic;
      tc    : out std_logic);
  end component;
--==============================================================================
--Signal Declarations:
--==============================================================================
--Control signals
    signal load_en            : std_logic; -- FSM to Register
    signal shift_en           : std_logic;
    signal bit_tc             : std_logic; --Bit counter to FSM
    signal bit_clr            : std_logic;

--Register signals
    signal reg              : std_logic_vector(15 downto 0);

--Controller signals
    type state_type is (sIdle, sLoad, sShift);
    signal next_state, current_state        : state_type := sIdle;

--Counter signals
    signal shiftNum           : integer := 0;

    signal data_w_volume : std_logic_vector(11 downto 0) := (others => '0');
    signal data_w_volume_padded : std_logic_vector(15 downto 0) := (others => '0');

    BEGIN

--========================== FSM Controller ==================================
stateUpdate : process(sclk)
    begin
    if rising_edge(sclk) then
      current_state <= next_state;
    end if;
end process stateUpdate;

nextStateLogic : process(current_state, bit_tc, key_down, take_sample)
    begin
    next_state <= current_state; --Default
    case current_state is
      when sIdle =>
        if key_down = '1' and take_sample = '1' then --Wait for start bit
          next_state <= sLoad;
        end if; --Else stay Idle
      when sLoad =>
        next_state <= sShift;
      when sShift =>
        if bit_tc = '1' then --When done shifting all the bits
          next_state <= sIdle;
        end if;
      when others =>
    end case;
end process nextStateLogic;

outputLogic : process(current_state)
    begin
    load_en <= '0';
    spi_cs <= '1'; --Idles high, transmits low
    shift_en <= '0';
    bit_clr <= '1';
    case current_state is
      when sIdle => --Do default
        null;
      when sLoad => --Do one cycle of load
        load_en <= '1';
      when sShift => --Shift and transmit low
        shift_en <= '1';
        spi_cs <= '0';
        bit_clr <= '0';
    end case;
end process outputLogic;
--=============================================================================
--===========================  Sub Count Proc ==================================
bit_counter : counter
generic map (
  MAX_COUNT => 16)
port map(
  clk => sclk,
  clr => bit_clr,
  en => shift_en,
  tc => bit_tc);

--==============================================================================
shift_Register : process(sclk)
    begin
    if rising_edge(sclk) then
      if load_en = '1' then
        reg <= "0000" & data_w_volume;
      elsif shift_en = '1' then --Should not be both on at any time
        reg <= reg(14 downto 0) & '0';
      end if;
    end if;
end process shift_Register;

--==============================================================================

  -- divide by 8 and multiply by 1-8
  data_w_volume_padded <= std_logic_vector(shift_right(unsigned(data_in), 3) * (1 + unsigned('0' & velocity_in)));


--Tie s_data to MSB of shift Register
```

```
    s_data <= reg(15);
data_w_volume <= data_w_volume_padded(11 downto 0);
end behavioral_architecture;
```

6. DDS + DAC Top Level - Behavioral

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;                          -- needed for arithmetic
use ieee.math_real.all;                                    -- needed for automatic register sizing
library UNISIM;                                                    -- needed for the BUFG component
use UNISIM.Vcomponents.ALL;

--=============================================================
--Shell Entitity Declarations
--=============================================================
entity dds_top_level is
  port (
    hw_clk_port : in std_logic; -- 100 MHz clock
    spi_cs_port : out std_logic;                      -- spi chip select
           spi_data_port       : out std_logic;                    -- spi data out
    spi_sclk_port    : out std_logic;     -- sclk out for spi
    take_sample_port : out std_logic);
end dds_top_level;

--=============================================================
--Architecture + Component Declarations
--=============================================================
architecture Behavioral of dds_top_level is
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--System Clock Generation:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
component system_clock_generation is
    port (
        hw_clk                         : in  std_logic;
        sclk      : out std_logic;
        fwd_clk   : out std_logic);
end component;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--DDS:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
component DDS is
  port (
    --inputs
    sclk : in std_logic;
    m_in : in std_logic_vector(13 downto 0);
    --outputs
    amp_out : out std_logic_vector(11 downto 0);
    take_sample : out std_logic
  );
end component;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--DAC Interface:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
component DAC_interface is
  Port (
    -- inputs
    -- 1MHz clock
    sclk : in std_logic;
    -- start bit
    key_down : in std_logic;
    take_sample : in std_logic;
    -- parallel data input
    data_in : in std_logic_vector(11 downto 0);

    -- outputs
    -- bit of serial data out
    s_data : out std_logic;
    -- Chip select
    spi_CS : out std_logic
  );
end component;

--=============================================================
--Local Signal Declaration
--=============================================================
signal sclk_sig : std_logic := '0';
signal data_sig  : std_logic_vector(11 downto 0) := (others => '0');
signal take_sample_sig : std_logic := '0';

constant m_value : std_logic_vector(13 downto 0) := "00000101000111";
constant key_down : std_logic := '1';

--=============================================================
--Port Mapping + Processes:
--=============================================================
begin
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Timing:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
clocking: system_clock_generation
port map(
		hw_clk  => hw_clk_port,
		sclk		=> sclk_sig,
  fwd_clk => spi_sclk_port);

DDS_blk : DDS
port map(
  sclk => sclk_sig,
  m_in => m_value,
  amp_out => data_sig,
  take_sample => take_sample_sig
);

DAC : DAC_Interface
port map(
   sclk => sclk_sig,
   data_in => data_sig,
   key_down => key_down,
   take_sample => take_sample_sig,
   s_data => spi_data_port,
   spi_CS => spi_cs_port
);

take_sample_port <= take_sample_sig;

end Behavioral;
```

7. TOP LEVEL SHELL

```
--Sam Barton and Grant Foley
--ES31/CS56
--Provides top level shell for MIDI Controller system
--=============================================================

--=============================================================
--Library Declarations
--=============================================================
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;				-- needed for arithmetic
use ieee.math_real.all;					-- needed for automatic register sizing
library UNISIM;						-- needed for the BUFG component
use UNISIM.Vcomponents.ALL;

--=============================================================
--Shell Entitity Declarations
--=============================================================
entity midi_top_level is
port (
		hw_clk_port		: in  std_logic;			-- ext 100 MHz clock
		midi_in_port			: in  std_logic;				-- async midi signal
  spi_cs_port                : out std_logic;                -- spi chip select
              spi_data_port                          : out std_logic;                              -- spi data out
  spi_sclk_port        : out std_logic);     -- sclk out for spi
end midi_top_level;

--=============================================================
--Architecture + Component Declarations
--=============================================================
architecture Behavioral of midi_top_level is
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--System Clock Generation:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
component system_clock_generation is
    port (
        hw_clk                          : in  std_logic;
        sclk      : out std_logic;
        fwd_clk   : out std_logic);
end component;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--MIDI Receiver:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
component MIDI_receiver is
  Port (
    -- inputs
    -- 1MHz clock
    sclk : in std_logic;
    -- serial midi bit
    MIDI_in : in std_logic;

    -- outputs
    -- byte of data
    byte_out : out std_logic_vector(7 downto 0);
    -- done receiving signal
    rx_done   : out std_logic);
end component;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Datapath:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
component datapath is
  port (
    sclk    : in  std_logic;
    byte_in : in  std_logic_vector(7 downto 0);
    rx_done : in  std_logic;
    key_down : out std_logic;
    m_out   : out std_logic_vector(13 downto 0);
    velocity_out : out std_logic_vector(2 downto 0)
  );
end component;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--DDS:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
component DDS is
  port (
    --inputs
    sclk : in std_logic;
    m_in : in std_logic_vector(13 downto 0);
    --outputs
    amp_out : out std_logic_vector(11 downto 0);
    take_sample : out std_logic
  );
end component;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--DAC Interface:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
component DAC_interface is
  Port (
    -- inputs
    -- 1MHz clock
    sclk : in std_logic;
    -- start bit
    key_down : in std_logic;
    -- parallel data input
    data_in : in std_logic_vector(11 downto 0);
    -- 3 bit velocity from dpath
    velocity_in : in std_logic_vector(2 downto 0);
     -- signal for 44 kHz sampler
    take_sample         : in std_logic;

    -- outputs
    -- bit of serial data out
    s_data : out std_logic;
    -- Chip select
    spi_CS : out std_logic
  );
end component;

--===============================================================
--Local Signal Declaration
--===============================================================
signal sclk_sig         : std_logic := '0';
signal rx_done_sig      : std_logic := '0';
signal byte_sig         : std_logic_vector(7 downto 0) := (others => '0');
signal key_down_sig     : std_logic := '0';
signal m_sig            : std_logic_vector(13 downto 0) := (others => '0');
signal ampl_sig         : std_logic_vector(11 downto 0) := (others => '0');
signal take_sample_sig  : std_logic := '0';
signal velocity_sig     : std_logic_vector(2 downto 0) := (others => '0');

--===============================================================
--Port Mapping + Processes:
--===============================================================
begin
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--Timing:
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
clocking: system_clock_generation
port map(
                hw_clk  => hw_clk_port,
                sclk          => sclk_sig,
  fwd_clk => spi_sclk_port);

receiver : MIDI_receiver
port map(
  sclk => sclk_sig,
  MIDI_in => MIDI_in_port,
  byte_out => byte_sig,
  rx_done => rx_done_sig);

dpath : datapath
port map(
  sclk => sclk_sig,
  byte_in => byte_sig,
  rx_done => rx_done_sig,
  key_down => key_down_sig,
  m_out => m_sig,
  velocity_out => velocity_sig
);

DDS_blk : DDS
port map(
  sclk => sclk_sig,
  m_in => m_sig,
  amp_out => ampl_sig,
  take_sample => take_sample_sig

);

DAC : DAC_Interface
port map(
   sclk => sclk_sig,
   data_in => ampl_sig,
   velocity_in => velocity_sig,
   key_down => key_down_sig,
   take_sample => take_sample_sig,
   s_data => spi_data_port,
   spi_CS => spi_cs_port
);

end Behavioral;
```

Appendix B: VHDL Testbenches

Include copies of your testbenches here. Follow the same order as Section 3. Again, please try to preserve nice formatting.

1. MIDI RECEIVER - Testbench

```
--Library Declarations
--==============================================================
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

 --==============================================================
--Testbench Entity Declaration
--==============================================================
ENTITY MIDI_receiver_tb IS
END MIDI_receiver_tb;

--==============================================================
--Testbench declarations
--==============================================================
architecture testbench of MIDI_receiver_tb is

component MIDI_receiver is
  Port (
    -- inputs
    -- 1MHz clock
    sclk : in std_logic;
    -- serial midi bit
    MIDI_in : in std_logic;

    -- outputs
    -- byte of data
    byte_out : out std_logic_vector(7 downto 0);
    -- done receiving signal
    rx_done   : out std_logic);
end component;

--==============================================================
--Local Signal Declaration
--==============================================================
signal system_clk     : std_logic := '0';
signal MIDI_in        : std_logic := '1';
signal byte_out       : std_logic_vector(7 downto 0) := (others => '0');
signal rx_done        : std_logic := '0';

constant CLK_PERIOD   : time := 1 us; -- 1 MHz clock
constant BAUDRATE     : time := 32 us; -- 31.25 kb/s baudrate

begin

uut: MIDI_receiver
port map (
  sclk => system_clk,
  MIDI_in => MIDI_in,
  byte_out => byte_out,
  rx_done => rx_done);


-- clock signal
clk_process : process
begin
  system_clk <= '0';
  wait for CLK_PERIOD / 2;
  system_clk <= '1';
  wait for CLK_PERIOD / 2;
end process clk_process;

stim_proc : process
begin
  wait for 3 * BAUDRATE;

  -- send status message for key down
  MIDI_in <= '0'; -- start bit
  wait for BAUDRATE;
  -- channel bytes
  MIDI_in <= '1';
  wait for BAUDRATE;
  MIDI_in <= '1';
  wait for BAUDRATE;
  MIDI_in <= '1';
  wait for BAUDRATE;
  MIDI_in <= '1';
  wait for BAUDRATE;
  -- status bits
  MIDI_in <= '1';
  wait for BAUDRATE;
  MIDI_in <= '0';
  wait for BAUDRATE;
  MIDI_in <= '0';
  wait for BAUDRATE;
  MIDI_in <= '1';
  wait for BAUDRATE;
  -- stop bit
  MIDI_in <= '1';
  wait for BAUDRATE;

  -- send status message for PITCH=100
  MIDI_in <= '0'; -- start bit
  wait for BAUDRATE;
  -- 7 pitch bits
  MIDI_in <= '0';
  wait for BAUDRATE;
  MIDI_in <= '0';
  wait for BAUDRATE;
  MIDI_in <= '1';
  wait for BAUDRATE;
  MIDI_in <= '0';
  wait for BAUDRATE;
  MIDI_in <= '0';
  wait for BAUDRATE;
  MIDI_in <= '1';
  wait for BAUDRATE;
  MIDI_in <= '1';
  wait for BAUDRATE;
  -- data bit signifier
  MIDI_in <= '0';
  wait for BAUDRATE;
  -- stop bit
  MIDI_in <= '1';
  wait for BAUDRATE;

  -- send status message for VOLUME=50
  MIDI_in <= '0'; -- start bit
  wait for BAUDRATE;
  -- 7 pitch bits
  MIDI_in <= '0';
  wait for BAUDRATE;
  MIDI_in <= '1';
  wait for BAUDRATE;
  MIDI_in <= '0';
  wait for BAUDRATE;
  MIDI_in <= '0';
  wait for BAUDRATE;
  MIDI_in <= '1';
  wait for BAUDRATE;
  MIDI_in <= '1';
  wait for BAUDRATE;
  MIDI_in <= '0';
  wait for BAUDRATE;
  -- data bit signifier
  MIDI_in <= '0';
  wait for BAUDRATE;
  -- stop bit
  MIDI_in <= '1';
  wait for BAUDRATE;
  wait;
end process stim_proc;

end;
```

2. DATAPATH - Testbench

```
--Library Declarations
--==============================================================
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

 --==============================================================
--Testbench Entity Declaration
--==============================================================
ENTITY datapath_tb IS
END datapath_tb;

--==============================================================
--Testbench declarations
--==============================================================
architecture testbench of datapath_tb is

component datapath is
  Port (
    sclk    : in  std_logic;
    byte_in : in  std_logic_vector(7 downto 0);
    rx_done : in  std_logic;
    key_down : out std_logic;
    m_out   : out std_logic_vector(8 downto 0)
  );
end component;

--==============================================================
--Local Signal Declaration
--==============================================================
--inputs
signal system_clk     : std_logic := '0';
signal byte_in_sig    : std_logic_vector(7 downto 0) := (others => '0');
signal rx_done_sig    : std_logic := '0';
--outputs
signal key_down       : std_logic := '0';
signal m_out         : std_logic_vector(8 downto 0) := (others => '0');

constant CLK_PERIOD   : time := 1 us; -- 1 MHz clock

begin

uut: datapath
port map (
  sclk => system_clk,
  byte_in => byte_in_sig,
  rx_done => rx_done_sig,
  key_down => key_down,
  m_out => m_out);


-- clock signal
clk_process : process
begin
  system_clk <= '0';
  wait for CLK_PERIOD / 2;
  system_clk <= '1';
  wait for CLK_PERIOD / 2;
end process clk_process;

stim_proc : process
begin
  -- key 100 down
  -- status byte
  wait for 10 * CLK_PERIOD;
  byte_in_sig <= "10011111";
  wait for 2 * CLK_PERIOD;
  rx_done_sig <= '1';
  wait for CLK_PERIOD;
  rx_done_sig <= '0';
  -- pitch byte
  wait for 10 * CLK_PERIOD;
  byte_in_sig <= "01100100";
  wait for 2 * CLK_PERIOD;
  rx_done_sig <= '1';
  wait for CLK_PERIOD;
  rx_done_sig <= '0';
  -- volume
  wait for 10 * CLK_PERIOD;
  byte_in_sig <= "00110010";
  wait for 2 * CLK_PERIOD;
  rx_done_sig <= '1';
  wait for CLK_PERIOD;
  rx_done_sig <= '0';

  -- weird bytes
  wait for 10 * CLK_PERIOD;
  byte_in_sig <= "11101111";
  wait for 2 * CLK_PERIOD;
  rx_done_sig <= '1';
  wait for CLK_PERIOD;
  rx_done_sig <= '0';
  wait for 10 * CLK_PERIOD;
  byte_in_sig <= "00111111";
  wait for 2 * CLK_PERIOD;
  rx_done_sig <= '1';
  wait for CLK_PERIOD;
  rx_done_sig <= '0';

  -- key 100 up
  -- status byte
  wait for 10 * CLK_PERIOD;
  byte_in_sig <= "10001111";
  wait for 2 * CLK_PERIOD;
  rx_done_sig <= '1';
  wait for CLK_PERIOD;
  rx_done_sig <= '0';
  -- pitch byte
  wait for 10 * CLK_PERIOD;
  byte_in_sig <= "01100100";
  wait for 2 * CLK_PERIOD;
  rx_done_sig <= '1';
  wait for CLK_PERIOD;
  rx_done_sig <= '0';
  -- volume
  wait for 10 * CLK_PERIOD;
  byte_in_sig <= "00000000";
  wait for 2 * CLK_PERIOD;
  rx_done_sig <= '1';
  wait for CLK_PERIOD;
  rx_done_sig <= '0';

  -- key 52 down
  -- status byte
  wait for 10 * CLK_PERIOD;
  byte_in_sig <= "10011111";
  wait for 2 * CLK_PERIOD;
  rx_done_sig <= '1';
  wait for CLK_PERIOD;
  rx_done_sig <= '0';
  -- pitch byte
  wait for 10 * CLK_PERIOD;
  byte_in_sig <= "00110100";
  wait for 2 * CLK_PERIOD;
  rx_done_sig <= '1';
  wait for CLK_PERIOD;
  rx_done_sig <= '0';
  -- volume
  wait for 10 * CLK_PERIOD;
  byte_in_sig <= "00110010";
  wait for 2 * CLK_PERIOD;
  rx_done_sig <= '1';
  wait for CLK_PERIOD;
  rx_done_sig <= '0';
  wait;
end process stim_proc;

end;
```

3. DDS - Testbench

```
--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 DDS_tb is
end entity;

--==============================================================================
--Architecture
--==============================================================================
architecture testbench of DDS_tb is

--==============================================================================
--Component Declaration
--==============================================================================
component DDS is
  port (
    sclk    : in  std_logic;
    m_in    : in  std_logic_vector(13 downto 0);
    amp_out : out std_logic_vector(11 downto 0)
    );
end component;

--==============================================================================
--Signals
--==============================================================================
signal sclk : std_logic;
signal m_data : std_logic_vector(13 downto 0);
signal amp_to_DAC    : std_logic_vector(11 downto 0);

CONSTANT clk_period : time := 1 us;
CONSTANT sample_period : time := 23 * clk_period;

begin

--==============================================================================
--Port Map
--==============================================================================
uut: DDS
        port map(
                sclk        => sclk,
                m_in    => m_data,
                amp_out     => amp_to_DAC
    );

--==============================================================================
--clk_4MHz generation
--==============================================================================
clkgen_proc: process
begin
sclk <= '0';
wait for clk_period/2;
sclk <= '1';
wait for clk_period/2;

end process clkgen_proc;

--==============================================================================
--Stimulus Process
--==============================================================================
stim_proc: process
begin
m_data <= "00000101000111"; --Equal to 327, should generate around 440Hz
wait for 120 * sample_period;
m_data <= "00001010001110";  --Equal to 654, should generate around 880Hz (one octave up)
wait for 30 * sample_period;
m_data <= "11111111111111"; --Very high frequency
wait for 30 * sample_period;
m_data <= "00000000000001"; --Base frequency, 1.6
wait for 60 * sample_period;

end process;
end testbench;
```

4. DAC Interface - Testbench

```
--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 DAC_interface_tb is
end entity;

--==============================================================================
--Architecture
--==============================================================================
architecture testbench of DAC_interface_tb is

--==============================================================================
--Component Declaration
--==============================================================================
component DAC_interface is
  Port (
    --timing:
    sclk                : in std_logic;
    --control inputs:
    start             : in std_logic;
    data_in      : in std_logic_vector(11 downto 0);

    --outputs:
    spi_CS            : out std_logic;
    s_data            : out std_logic)
  ;
end component;

--==========================================================================
--Signals
--==========================================================================

signal sclk       : std_logic := '0'; --Clock
signal key_down   : std_logic := '0'; --Simulated signal when key goes down, from datapath
signal p_data     : std_logic_vector(11 downto 0); --12 bits from Direct Digital Synthesis, loaded into reg

begin

--==========================================================================
--Port Map
--==========================================================================
uut: DAC_interface
        port map(
                sclk        => sclk,
                start       => key_down,
                data_in     => p_data
    );

--==========================================================================
--clk_4MHz generation
--==========================================================================
clkgen_proc: process
begin
sclk <= '0';
wait for 20 ns;
sclk <= '1';
wait for 20 ns;

end process clkgen_proc;

--==========================================================================
--Stimulus Process
--==========================================================================
stim_proc: process
begin

--Try loading in data from data_in
p_data <= "110011001100";
wait for 40 ns;
key_down <= '1';
wait for 720 ns; --Should load 1 cycle, then shift out 15 bits, starting with 3 leading zeroes and 12 data bits by MSB
key_down <= '0';
wait for 1000 ns; --Should complete cycle it is on before stop, idle

--Try loading in some new data twice in a row while holding key down
p_data <= "111100001010";
wait for 40 ns;
key_down <= '1';
wait for 1280 ns;
key_down <= '0';
wait for 100ns;
  wait;
end process stim_proc;

end testbench;
```

5. Top Shell - Testbench

```
--Library Declarations
```

```
--==============================================================
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;


--Testbench Entity Declaration
--==============================================================
ENTITY midi_shell_tb IS
END midi_shell_tb;

--==============================================================
--Testbench declarations
--==============================================================
architecture testbench of midi_shell_tb is

component midi_top_level is
  Port (
                hw_clk_port          : in  std_logic;                    -- ext 100 MHz clock
                midi_in_port                          : in  std_logic;                           -- async midi signal
                spi_cs_port                  : out std_logic;                -- spi chip select
                spi_data_port                                    : out std_logic;                           -- spi data out
    spi_sclk_port     : out std_logic       -- sclk out for spi
  );
end component;

--==============================================================
--Local Signal Declaration
--==============================================================
--inputs
signal midi_in_sig : std_logic := '1';
signal hw_clk : std_logic := '0';
signal spi_cs_port : std_logic := '0';
signal spi_data_port : std_logic := '0';
signal spi_sclk_port : std_logic := '0';

constant CLK_PERIOD   : time := 10 ns; -- 100 MHz hw clock
constant BAUDRATE     : time := 32 us; -- 31.25 kb/s baudrate

begin

uut: midi_top_level
port map (
              hw_clk_port => hw_clk,
              midi_in_port => midi_in_sig,
  spi_cs_port            => spi_cs_port,
              spi_data_port          => spi_data_port,
  spi_sclk_port => spi_sclk_port);


-- clock signal
clk_process : process
begin
  hw_clk <= '0';
  wait for CLK_PERIOD / 2;
  hw_clk <= '1';
  wait for CLK_PERIOD / 2;
end process clk_process;

stim_proc : process
begin
  wait for 3 * BAUDRATE;

  -- send status message for key down
  midi_in_sig <= '0'; -- start bit
  wait for BAUDRATE;
  -- channel bytes
  midi_in_sig <= '1';
  wait for BAUDRATE;
  midi_in_sig <= '1';
  wait for BAUDRATE;
  midi_in_sig <= '1';
  wait for BAUDRATE;
  midi_in_sig <= '1';
  wait for BAUDRATE;
  -- status bits
  midi_in_sig <= '1';
  wait for BAUDRATE;
  midi_in_sig <= '0';
  wait for BAUDRATE;
  midi_in_sig <= '0';
  wait for BAUDRATE;
  midi_in_sig <= '1';
  wait for BAUDRATE;
  -- stop bit
  midi_in_sig <= '1';
  wait for 5*BAUDRATE;

  -- send status message for PITCH=100
  midi_in_sig <= '0'; -- start bit
  wait for BAUDRATE;
  -- 7 pitch bits
  midi_in_sig <= '0';
  wait for BAUDRATE;
  midi_in_sig <= '0';
  wait for BAUDRATE;
  midi_in_sig <= '1';
  wait for BAUDRATE;
  midi_in_sig <= '0';
  wait for BAUDRATE;
  midi_in_sig <= '0';
  wait for BAUDRATE;
  midi_in_sig <= '1';
  wait for BAUDRATE;
  midi_in_sig <= '1';
  wait for BAUDRATE;
  -- data bit signifier
  midi_in_sig <= '0';
  wait for BAUDRATE;
  -- stop bit
  midi_in_sig <= '1';
  wait for 5*BAUDRATE;

  -- send status message for VOLUME=50
  midi_in_sig <= '0'; -- start bit
  wait for BAUDRATE;
  -- 7 pitch bits
  midi_in_sig <= '0';
  wait for BAUDRATE;
  midi_in_sig <= '1';
  wait for BAUDRATE;
  midi_in_sig <= '0';
  wait for BAUDRATE;
  midi_in_sig <= '0';
  wait for BAUDRATE;
  midi_in_sig <= '1';
  wait for BAUDRATE;
  midi_in_sig <= '1';
  wait for BAUDRATE;
  midi_in_sig <= '0';
  wait for BAUDRATE;
  -- data bit signifier
  midi_in_sig <= '0';
  wait for BAUDRATE;
  -- stop bit
  midi_in_sig <= '1';
  wait for 5*BAUDRATE;
  wait;
end process stim_proc;

end;
```

Constraints

```
## This file is a general .xdc for the Basys3 rev B board for ENGS31/CoSc56
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project

##=======================================================================
## External_Clock_Port
##=======================================================================
set_property PACKAGE_PIN W5 [get_ports hw_clk_port]
	set_property IOSTANDARD LVCMOS33 [get_ports hw_clk_port]
	create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports hw_clk_port]

##=======================================================================
## Pmod Header JA
##=======================================================================
#Sch name = JA1
set_property PACKAGE_PIN J1 [get_ports {spi_cs_port}]
	set_property IOSTANDARD LVCMOS33 [get_ports {spi_cs_port}]
#Sch name = JA2
set_property PACKAGE_PIN L2 [get_ports {spi_data_port}]
	set_property IOSTANDARD LVCMOS33 [get_ports {spi_data_port}]
#Sch name = JA3
#set_property PACKAGE_PIN J2 [get_ports {JA_port[2]}]
#	set_property IOSTANDARD LVCMOS33 [get_ports {JA_port[2]}]
#Sch name = JA4
set_property PACKAGE_PIN G2 [get_ports {spi_sclk_port}]
	set_property IOSTANDARD LVCMOS33 [get_ports {spi_sclk_port}]

#sch name = JA7
#set_property PACKAGE_PIN H1 [get_ports {take_sample_port}]
#set_property IOSTANDARD LVCMOS33 [get_ports {take_sample_port}]

##=======================================================================
## Pmod Header JB
##=======================================================================
##Sch name = JB1
set_property PACKAGE_PIN A14 [get_ports {midi_in_port}]
	set_property IOSTANDARD LVCMOS33 [get_ports {midi_in_port}]

##=======================================================================
## Implementation Assist
##=======================================================================
## These additional constraints are recommended by Digilent, do not remove!
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]

set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]

set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]

##Pmod Header JB


##Sch name = JB7
#set_property PACKAGE_PIN A15 [get_ports {JB[4]}]
	#set_property IOSTANDARD LVCMOS33 [get_ports {JB[4]}]
##Sch name = JB8
#set_property PACKAGE_PIN A17 [get_ports {JB[5]}]
	#set_property IOSTANDARD LVCMOS33 [get_ports {JB[5]}]
##Sch name = JB9
#set_property PACKAGE_PIN C15 [get_ports {JB[6]}]
	#set_property IOSTANDARD LVCMOS33 [get_ports {JB[6]}]
##Sch name = JB10
#set_property PACKAGE_PIN C16 [get_ports {JB[7]}]
	#set_property IOSTANDARD LVCMOS33 [get_ports {JB[7]}]
```