Polyphonic Digital Synthesizer

Grant Larson and Catherine Slaughter

Abstract

For our project, we created a polyphonic digital synthesizer with a MIDI interface for MIDI keyboard input. The synthesizer receives 3-byte packages of MIDI data from a MIDI keyboard using the UART serial protocol. For each of the 25 keys on the keyboard, the FPGA stores whether the note is on or off. For each note that is on, a sinusoidal waveform is generated. The resultant waveforms are added and output to a speaker. The synthesizer can play up to 25 notes simultaneously, pitched from C3 through C5.

Table of Contents

1. Introduction……...3
2. Design Solution…….....3-12
    2.1. Specifications
    2.2. Operating Instructions
    2.3. Theory of Operation
    2.4. Construction and Debugging
3. Justification and Evaluation……..12
4. Conclusions…….....12-13
5. Acknowledgements…….….13-14
6. References……...14-15
7. Appendices……..……...16-79
    7.1. Appendix A : Front Panel
    7.2. Appendix B: Block Diagrams
    7.3. Appendix C: State Diagrams
    7.4. Appendix D: Simulation Waveforms
    7.5. Appendix E: Parts List
    7.6. Appendix F: VHDL Code
    7.7. Appendix G: VHDL Testbench Code
    7.8. Appendix H: Resource Utilization
    7.9. Appendix I: Memory Map
    7.10. Appendix J: Analysis of Residual Warnings
    7.11. Appendix K: MATLAB Code

1. Introduction

Our project is to build a polyphonic digital synthesizer with a MIDI interface. The synth takes MIDI keyboard input and outputs the sine wave corresponding to each note. If multiple notes are playing, the synthesizer adds the appropriate sine waves.

2. Design Solution

2.1. Specifications

Our circuit converts MIDI keyboard input to audio output (Appendix A; Appendix B, Figure 1). The MIDI keyboard is connected to the Basys 3 FPGA via a 5-pin MIDI connector. MIDI data is received as UART serial at a baud rate of 31250, the MIDI standard. The audio is output from the FPGA to Pmod DA2 as 16-bit SPI-like serial. The analog output of the DA converter is transmitted to a speaker, which outputs the synthesized audio. The system is clocked at 1 MHz. Audio is sampled at 50 kHz.

2.2. Operating Instructions

Operation of our polyphonic MIDI synth is simple. Once the keyboard is plugged in and turned on and the FPGA is powered up and programmed, it is good to go. Pressing the keys will cause the corresponding notes to be played as expected, with the most recently changed (on or off) note number showing up in hexadecimal format on the FPGA 7-segment display. There is no need to flip any switches on the FPGA. It is important that the user does not change any of the default settings on the MIDI keyboard (such as the pitch wheel or octave setting) while the synth is in use. Such

features do not send MIDI codes in the 3-byte packages that our MIDI interface expects. Doing so will throw the controller out of sync and the synth will need to be reset.

2.3. Theory of Operation

We designed our polyphonic synth to be as modular as possible. Doing so made it easier to test each part of our design as we went along, and also made building and adding new features much simpler. Our final design has four independent modules under the same shell (Appendix B, Figure 2). The first is the UART interface, which deals with taking in and synchronizing the serial MIDI data coming from the keyboard, packaging it up into 8-bit bytes, and parallel loading it into our MIDI interface. The MIDI interface takes in the 3 bytes, interprets each as necessary (i.e. reading a note on/note off signal from the message byte), and passes the important info from each on to our sound engine. In the case of our synth, only the first two bytes (the message signal and note number) are actually passed on. However, our design is such that the third byte (the velocity) is also registered, so it would be reasonably simple to add on new modules and features that utilize the velocity data. The next module in our design is our sound engine, which deals with interpreting the note on/off and note number signals from the MIDI interface, keeping track of which notes are pressed, and generating and adding sine waves. It passes this data to our final module, the DA interface, with turns our parallel sine wave data into serial data to be passed along a SPI interface that the digital to analog (DA) converter Pmod can understand. A more in-depth description of each module can be found below.

UART INTERFACE (Appendix B, Figure 3)

The UART interface loads serial data in from the keyboard to be output along an 8-bit bus to the MIDI interface. We use a typical interface here, not entirely unlike that used in the SPI bus lab. First, the serial data gets pushed into a shift register. Then, the shift register parallel-loads its data into an output register, at which point the midi_done_tick is asserted for the MIDI interface. The difficult part of UART is that unlike the SPI bus interface we used in lab, UART does not pass along a clock, so we had to synchronize. To do this, we used a double-flop synchronizer and a clock divider. The clock was divided down based on the agreed-upon baud rate of the MIDI UART data, which is 31250 bits/second. When the serial input goes low, the UART controller (Appendix C, Figure 1) enters a calibration state and identifies the center of the start bit using the middle count of the clock divider counter. The controller then moves to the shift state and continues shifting at the center of each bit, ensuring that a small timing error would not impact the values read in by the UART interface. When 10 shifts are complete (1 start bit, 8 data bits, 1 stop bit), the serial data line returns high, and the shift register is parallel loaded into the output register.

MIDI INTERFACE (Appendix B, Figure 3)

The MIDI interface works by taking in 3 bytes of MIDI data from the UART interface one at a time, and multiplexing them into the appropriate registers for the message, note number, or velocity signal. A counter (which goes from 0-2) outputs the

current count as the select bit to the multiplexer and informs the controller of which byte is being loaded next. When the controller receives an rx_done_tick signal from the UART interface, it checks what the current count is. If the count is 0, the message register loads. If the count is 1, the note number register loads. If the count is 2, the velocity register loads. Each time a register is enabled, the count is incremented by 1. The count rolls back to 0 after the velocity byte has been loaded. Once all three registers have been updated with new data, the MIDI controller (Appendix C, Figure 2) outputs a midi_done_tick for the sound engine. For our synth in particular, the interface runs the message signal through a lookup table to decode the note on/off message[1] to a one-bit signal -- 1 for on, 0 for off. It outputs this new signal and the 8-bit note number to the sound engine.

SOUND ENGINE (Appendix B, Figure 4)

The sound generator module takes in the note on/off signal and note number from the MIDI interface, along with a take_sample tick that is generated in the shell. This signal is generated by dividing the 1MHz clock down to a 50kHz signal. It is crucial to the sound engine's timing that the take_sample tick has a frequency smaller than that of the master clock by several orders of magnitude.

The sound engine begins with a note on register that functions similarly to a map or a dictionary. When the engine receives a midi_done_tick signal from the MIDI interface, it stores the value of the note_on signal at the location along the register

[1] By MIDI convention, the first 4 bits of a note on message are 1001 (9 in binary), and the first 4 bits of a note off message are 1000 (8 in binary)

corresponding to the note_num signal coming from the MIDI interface. The note_num is the key and the note_on signal is the value. As the data comes in from the MIDI interface, a count of the number of keys currently being pressed is kept, increasing with each note on and decreasing with each note off. This is used later to decrease the volume of the final output. The note numbers sent by the keyboard range from 48 (C3) to 72 (C5). We subtract the value of the lowest note number, 48, from the note number coming in, so our note on register is indexed from 0 to 24.

Each bit on the register acts as the enable bit for one of 25 phase accumulators, one for each key of the keyboard. The phase accumulators output a value between 0 and 4095 (a 12-bit number), where each possible output corresponds to a value in the sine lookup table. In our code, we were able to build the phase accumulators using a VHDL generate statement, instead of hard coding each one individually. The specifics of each phase accumulator can be found in the **Phase Accumulators** section below.

In order to achieve polyphony, the sine waves for each of the pressed keys must be added together. Each key corresponds to a sine wave of a given frequency, and the sound of several keys is the superposition of all the corresponding waves. To implement polyphony, we take advantage of the several-order-of-magnitude difference between the frequencies of our system clock and our take_sample signal. Because the individual note phase accumulators only update on the take_sample signal, we can cycle through and sum up the values in a sine accumulator in between take_sample ticks using the system clock.

The sound generator controller is in charge of running this whole process (Appendix C, Figure 3). The controller contains a counter that runs from 0 to 24, incrementing on the 1MHz clock. Each time a take_sample tick is received, the counter begins incrementing, and stops after terminal count is reached. The count of the counter acts as the select bit (called count_mux on our block diagram) for a multiplexer, which takes in the current values of each phase accumulator and outputs them to the sine wave LUT (Appendix I). From here, the output of the sine wave LUT is input into a sine accumulator. After terminal count is reached, the controller goes into a load state. The total value stored in the sine accumulator is divided by the number of keys that are currently being pressed -- such that the volume does not increase as the number of keys pressed does -- before being stored in a register to be output to the DA interface. The sine accumulator sums 25 12-bit numbers from the phase accumulators, so it outputs an 18-bit number. However, dividing by the number of keys pressed ensures that the output always ends up being 12 bits, matching the width of the DA converter. All of these operations (multiplexing, sine lookup, accumulating, and loading into the register) are run on the 1MHz clock and occur before the next take_sample tick comes, so they do not impact the way the final output sounds. The DA interface is also run on the slow take_sample clock. The final output register is absolutely crucial because the sine accumulator must be reset to 0 before the next take_sample tick, but the data also needs to be available at the same take_sample tick.

While the logic for adding up the sine values of all 25 phase accumulators is not too difficult, it is a bit tricky to clock. This is because there is an inherent latency present

in the sine wave LUT. In our case, the latency is 6 clock cycles. In order to deal with this, we have to delay any outputs from the controller that go to parts of the data path after the LUT, specifically our accum_en and load_en signals. In order to do this, we fed the outputs from the controller through shift registers of length 6. For each signal, we pushed the value from the controller into the MSB of each register and popped the LSB as our accum_en_delay and load_en_delay, respectively. By making the length of the registers equal to the latency of the LUT, we ensured that the signals reached their respective places six clock cycles after the controller updated them. Doing so ensured that we did not begin to accumulate data too early (thereby accidentally adding old data), nor did we stop accumulating too late or load too early (missing the last 6 sine values). This made our controller design simpler, as we didn't need to include weird waiting states where counting had finished but we weren't ready to load the data.

Phase Accumulator

The individual phase accumulators in our sound engine take in the value of note_num_reg at the index for their corresponding key on the board and the take_sample signal from the shell. Each accumulator outputs a value from 0 to 4095 to be put through the sine LUT. For each accumulator, although it is being clocked on the 1Mhz clock, it will only update when both its corresponding note_on signal is high and when the 50 KHz take_sample signal is up (which it only is for a single 1MHz clock cycle). The step value the accumulator increases by (called M) is passed in as a generic from the sound generator at the generate statement and does not change. The M values

are hard-coded into a constant array of integers in our sound generator program. They were calculated according to the Direct Digital Synthesis method for generating sine waves, as outlined on the Canvas page "How to make a sine wave (revised)" written by Professor Hansen. For our synthesizer, our sampling frequency is 50kHz, our N is 4096, and our F_0 (the frequency of the first note on the keyboard) is 130.81Hz. The calculations of our 25 M values were done in MatLab (Appendix K).

We found that our chosen values of N and F_s gave us one or two places where the M values calculated for two neighboring notes was the same, because M has to be an integer. To retain precision and eliminate this issue, we multiplied all our N values by 1024 (a 10-bit shift) and made our phase accumulators 22 bits wide. The accumulated value is then shifted back 10 bits before being output to the sound generator multiplexer.

DA INTERFACE (Appendix B, Figure 5)

The DA interface turns our 12-bit bus of data from the sine wave accumulator and turns it into a SPI-like data stream, which that DA Pmod converts to analog data for the speaker. It loads in the data on the take_sample tick, before shifting the bits out one at a time on the 1MHz clock, well before the next take_sample tick comes in. The low-true spi_cs signal is handled by the DA controller (Appendix C, Figure 4), and the 1MHz spi_clk is passed on to the DA converter by the shell.

2.4. Construction and Debugging

After spending a lot of time designing three of our four modules, we first built our project in three overarching pieces, our input modules, output modules, and sound generator. We first built our input modules, the UART and MIDI interfaces, wired them together and simulated each section to verify that data was being read, interpreted, and passed on correctly (Appendix D, Figures 1-4; Appendix G). We did not do a physical test until later in the process. Next, we wrote the output modules, beginning with the DA interface. To test the DA interface, we also wrote the PA module that we were planning on using in our sound engine, and wired the two together in a shell such that the speaker would play middle C. We then conducted a physical test of this section, using the oscilloscope to make sure data shifted out correctly (Appendix D, Figures 5-7). Once every other piece of our synth was verified, we started building the sound engine. We were exceptionally meticulous in writing the module code and while wiring all our modules up in our top-level shell, so much so that we did not end up needing to testbench that piece of the project.

Thankfully, we did not have to spend too much time debugging any one part of our project. Obviously, there were many small typos and such that were easy fixes, so we will outline only the most significant and/or frustrating bugs here. For the input modules, our physical test revealed a bug that had not surfaced in our simulation. We did not perform a physical test of this piece until after we had built and tested our output modules, and during this time we decided to change our master clock from

10MHz to 1MHz. This caused an error in the clock divider for the UART interface. This bug was easily fixed once discovered. We spent more time debugging the output modules. Ultimately, we found two primary bugs. The first being that we had failed to instantiate the phase accumulator with a starting value of 0, so when it went to accumulate it was adding M to undefined. The second was an error in our XDC file where we tried to map the digital audio data to two different ports, so the DA converter could not receive it properly. When we went to run our final project, the first two things we checked were the sine accumulator's initial value and the XDC file. We had once again made errors in each, but these were the only remaining bugs present. Had we encountered more issues, we would have simulated our sound engine module, but through meticulous coding we were able to avoid doing so.

3. Justification and Evaluation

We found our design to be generally efficient, with minimal resource utilization and robust polyphonic functionality. Through use of a generate statement in the sound engine module, we were able to efficiently generate the 25 accumulators necessary for the full range of a 25 key MIDI keyboard. The less efficient alternative would be to instantiate each accumulator separately. We made some trade-offs between efficiency and sound quality in our phase accumulators, where we added in the 10-bit shift on the M values then divided back down. Obviously this division takes some computational power, but we decided it was worth it so that each note had a unique frequency.

The biggest improvement that could be made upon our design is dividing the added sine waves by the square root of the number of keys pressed rather than number of keys pressed, so the volume is not decreased by pressing multiple keys at once. As is mentioned in the comments of the sound engine code in Appendix G, we attempted to handle the square root by creating a coarse LUT. However, we found that the dividing by less than N caused the sine sum to overflow, resulting in noise. At that point, we decided in the interest of time that the square root functionality was best left for a future iteration.

4. Conclusions

In the end, our final project went above and beyond the expectations we set in our original design. Our initial design was a simple monophonic synthesizer without any bells or whistles, designed to be intentionally modular so we could potentially try for polyphony later in the process. We spent the majority of our time on this project trying to design and implement the datapath and controller for our polyphonic sound engine module, so finally getting it to work was very satisfying. Although we achieved more than we had hoped to for the purposes of this project, it is important to note that our design is intentionally modular so new functionality could easily be swapped in. Given all the time in the world, we could potentially add in features such as an interpretation of the velocity byte to augment volume, string sounds, or a low-pass filter to clean up the sound output. These are just a few of a seemingly endless number of additional features that could be added to our design.

Given the chance, our strongest recommendations to future groups considering a polyphonic synth would be to make your design as modular as possible, and to start early designing the logic for your sound engine. Don't be afraid to ask lots of questions. In general, we'd recommend groups take the time to sit down and work through the data path(s), controller(s), and *clocking* of your system very carefully before you go about writing any code. Additionally, we recommend you code systematically, with lots of comments. Most of our modules were carefully coded moving from left-to-right on our block diagrams. Both of these practices slashed the time and frustration we had to spend on debugging and writing our report.

5. Acknowledgements

Thank you to our project TA Evan for his time, willingness to answer questions, and general support from the very first steps of our project. We would also like to thank Dave and Dave for their patience and expertise in the lab. Finally, we would like to thank Professor Hansen for his guidance, insight, and enthusiasm throughout the duration of ENGS 31 and the project process.

Throughout the project we generally worked on things together, as opposed to dividing up the work. All of our data paths and state diagrams were designed by both of us with much, sometimes heated, discussion. When in the lab, we would take turns typing the code and guiding the typer, so as not to get too distracted on any one section. Grant

wrote the UART and DA interfaces, which we both worked to debug. Working and coding together in this way was important for the overall flow of our project, because it retained consistency in how the different modules interact with each other. We both contributed to writing this report. Grant made the data path diagrams, and Catherine made the state machine diagrams. In general, Grant pulled most of the appendix together and Catherine focused on writing most of the report itself, with plenty of collaboration and overlap.

6. References

Our most important references we used in this project were the Canvas page "How to make a sine wave (revised)" written by Professor Hansen, and the old UART interface lab that was used in previous years of ENGS 31. Any resources from the internet that we used can be found in the bibliography below.

J., Byron. "MIDI Tutorial." *Sparkfun*, 8 Oct. 2015, learn.sparkfun.com/tutorials/midi-tutorial/all.

"MIDI Note Numbers and Center Frequencies." *Inspired Acoustics*, www.inspiredacoustics.com/en/MIDI_note_numbers_and_center_frequencies.

Murphy, Eva, and Colm Slattery. "Ask The Application Engineer-33: All About Direct Digital Synthesis." *Analog Dialouge*, Aug. 2004, www.analog.com/en/analog-dialogue/articles/all-about-direct-digital-synthesis.html.

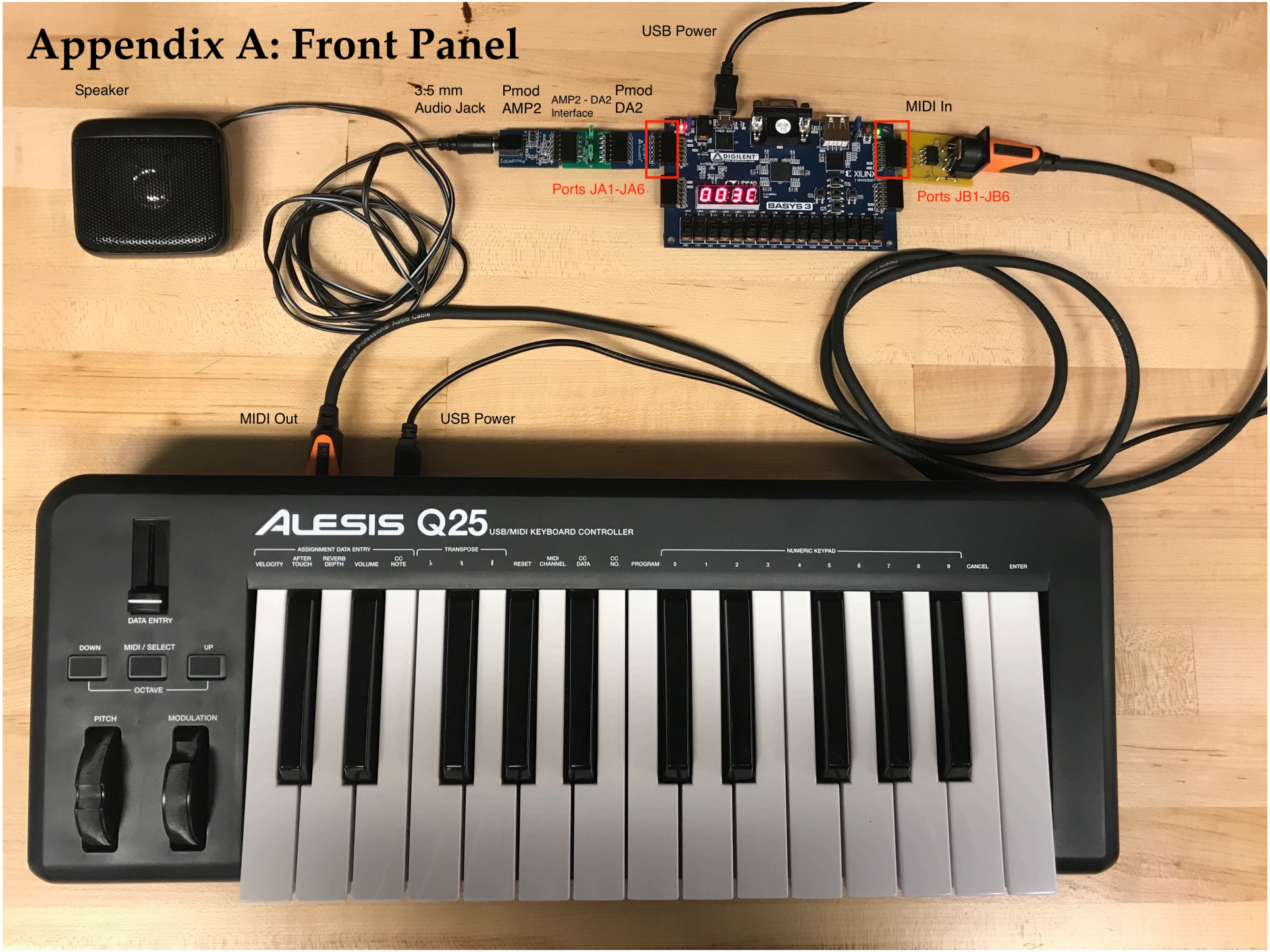
"Musical Notes." *Musical Note Frequencies*, TechLib.com, www.techlib.com/reference/musical_note_frequencies.htm.

Appendix

Polyphonic Digital Synthesizer

Grant Larson and Catherine Slaughter

Appendix A: Front Panel



Appendix B: Block Diagrams

Figure 1: Top Level Block Diagram

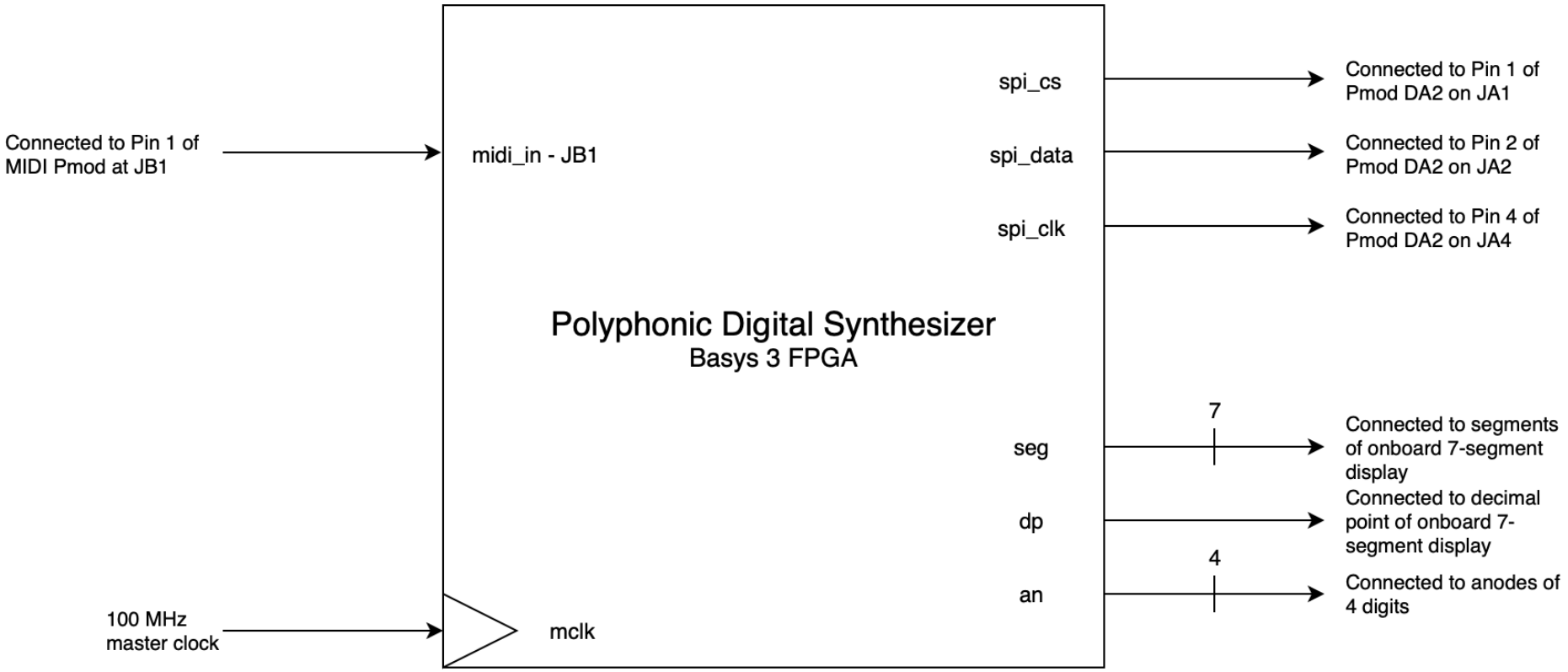


Figure 2 : Top Level Logic Block Diagram

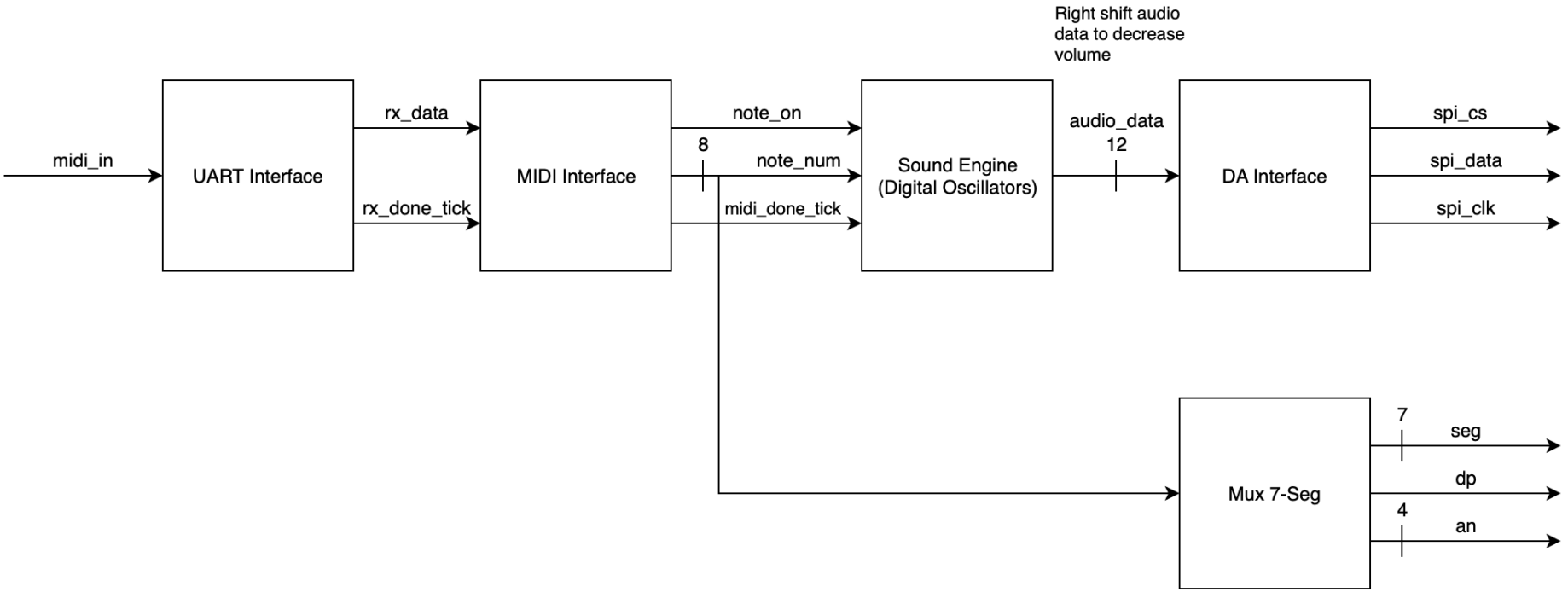


Figure 3: UART and MIDI Interface Block Diagram

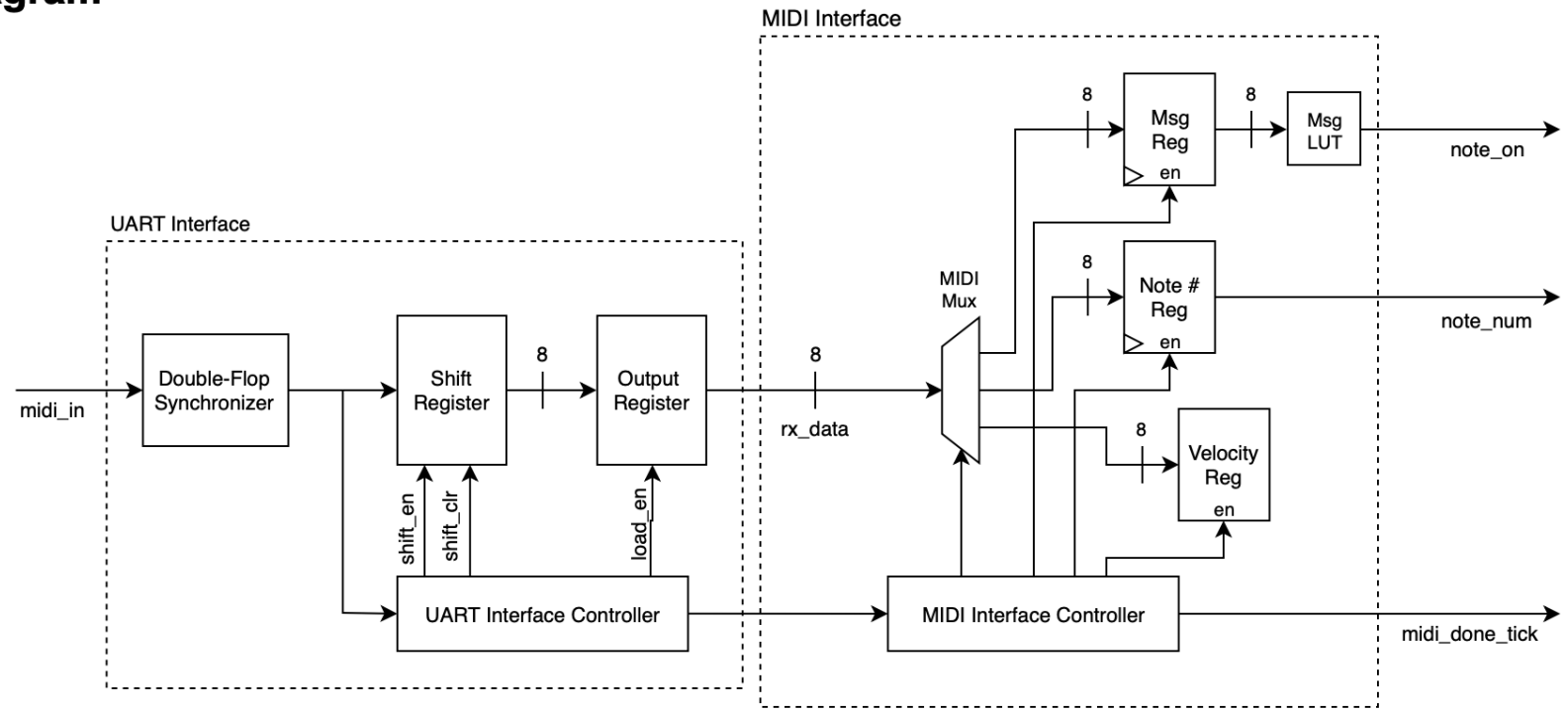


Figure 4: Sound Engine Block Diagram

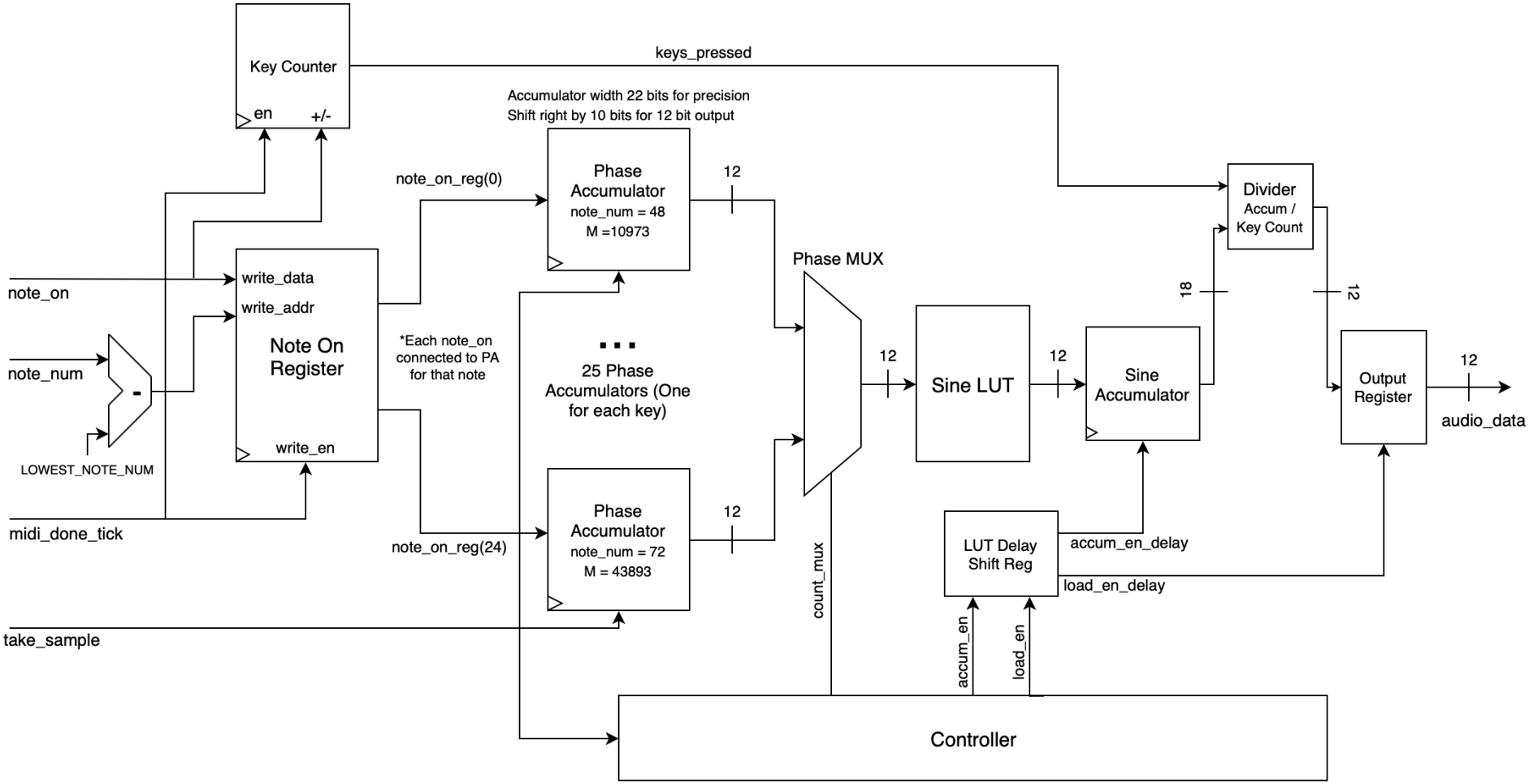
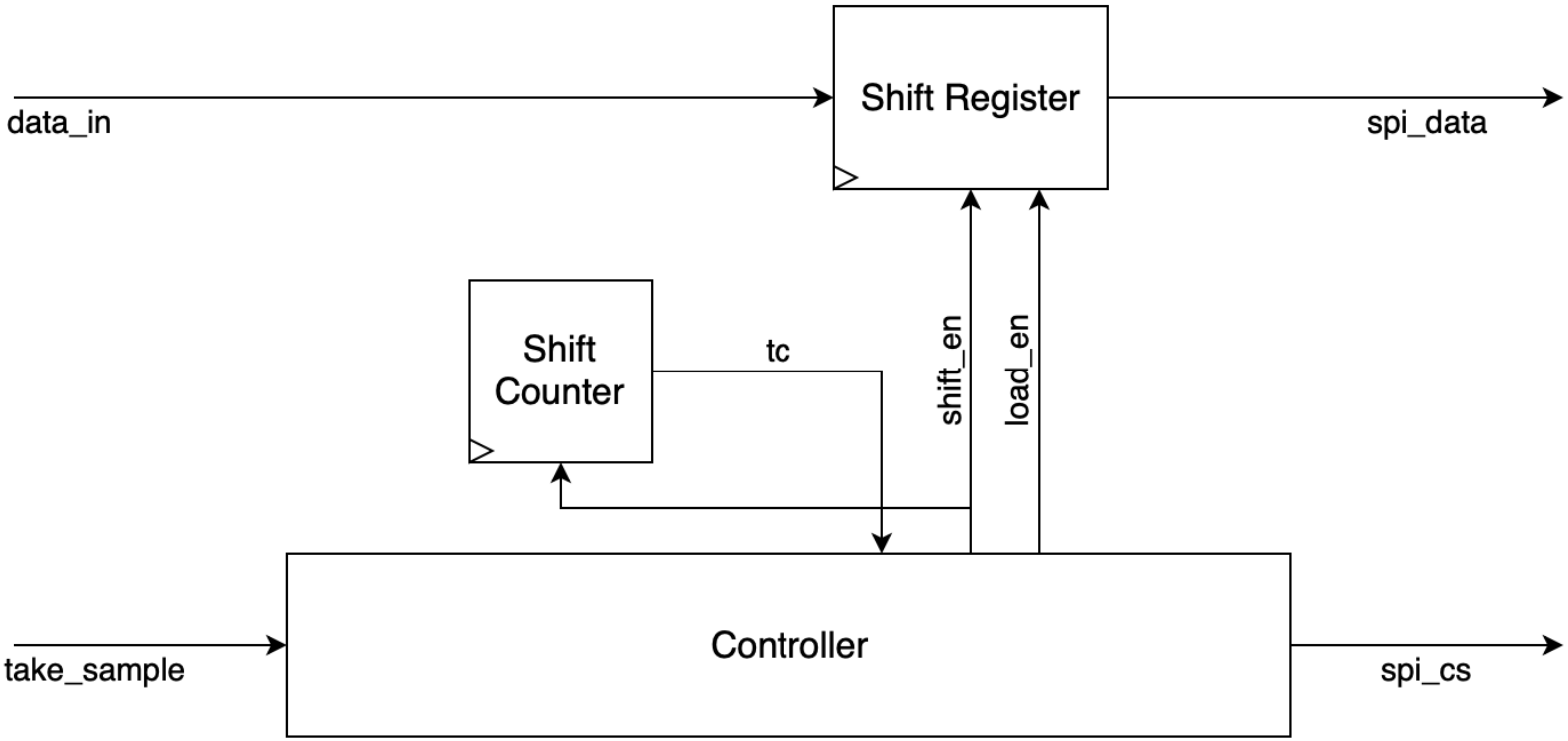


Figure 5: DA Interface Block Diagram



Appendix C: State Diagrams

Figure 1: UART Interface Controller State Diagram

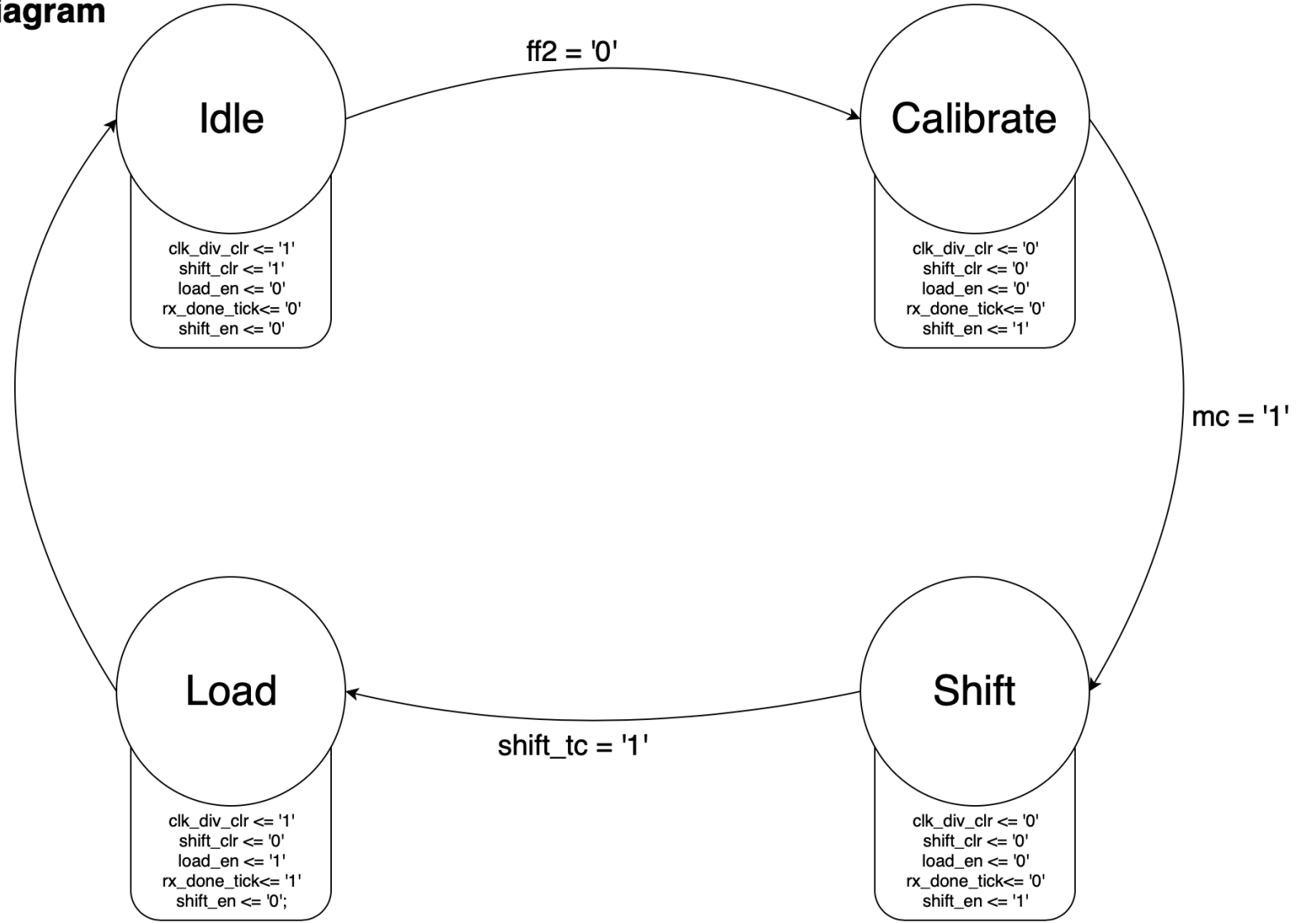


Figure 2: MIDI Interface Controller State Diagram

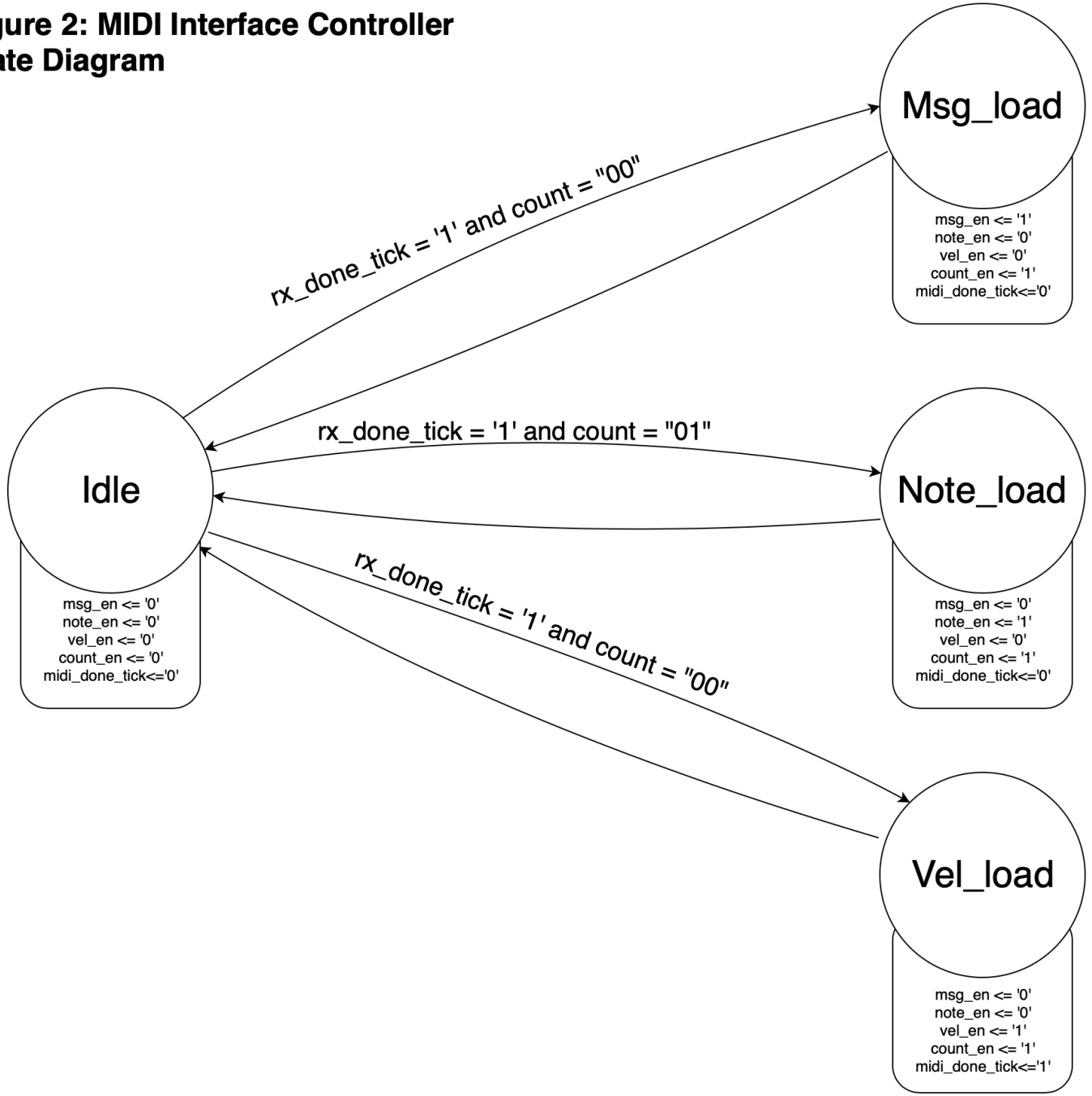


Figure 3: Sound Engine Controller State Diagram

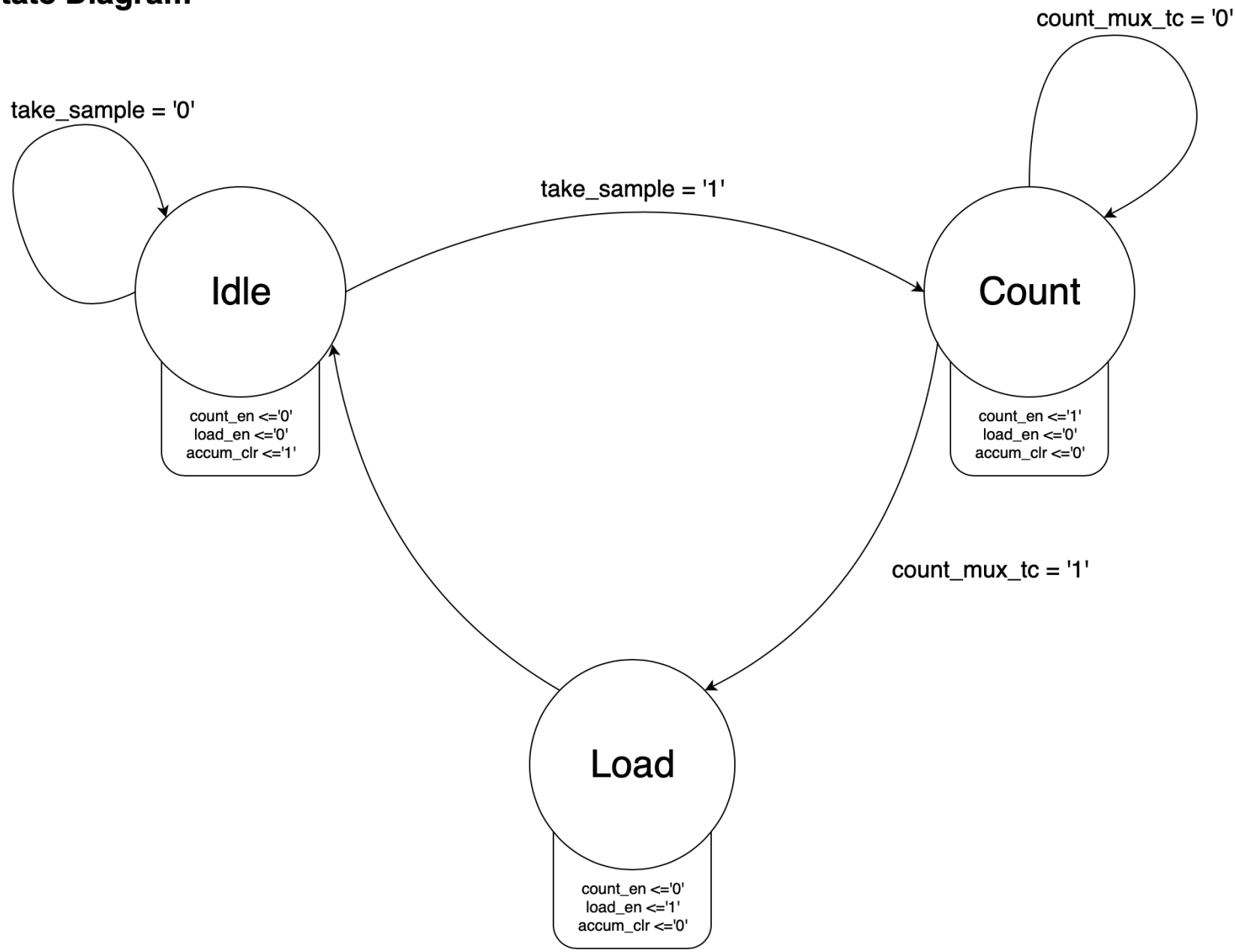
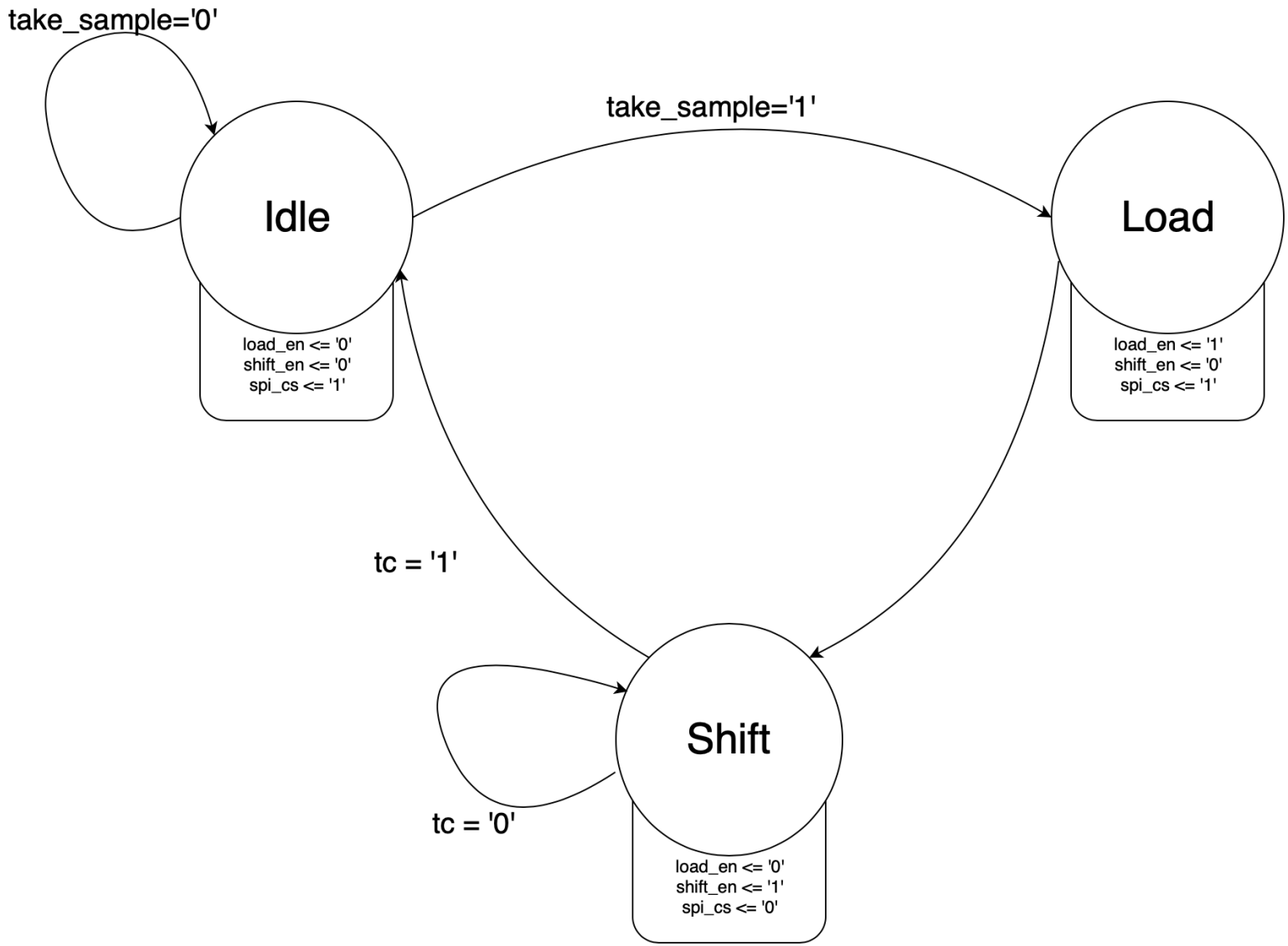


Figure 4: DA Interface Controller State Diagram



Appendix D: Waveforms

Figure 1: UART Interface Testbench

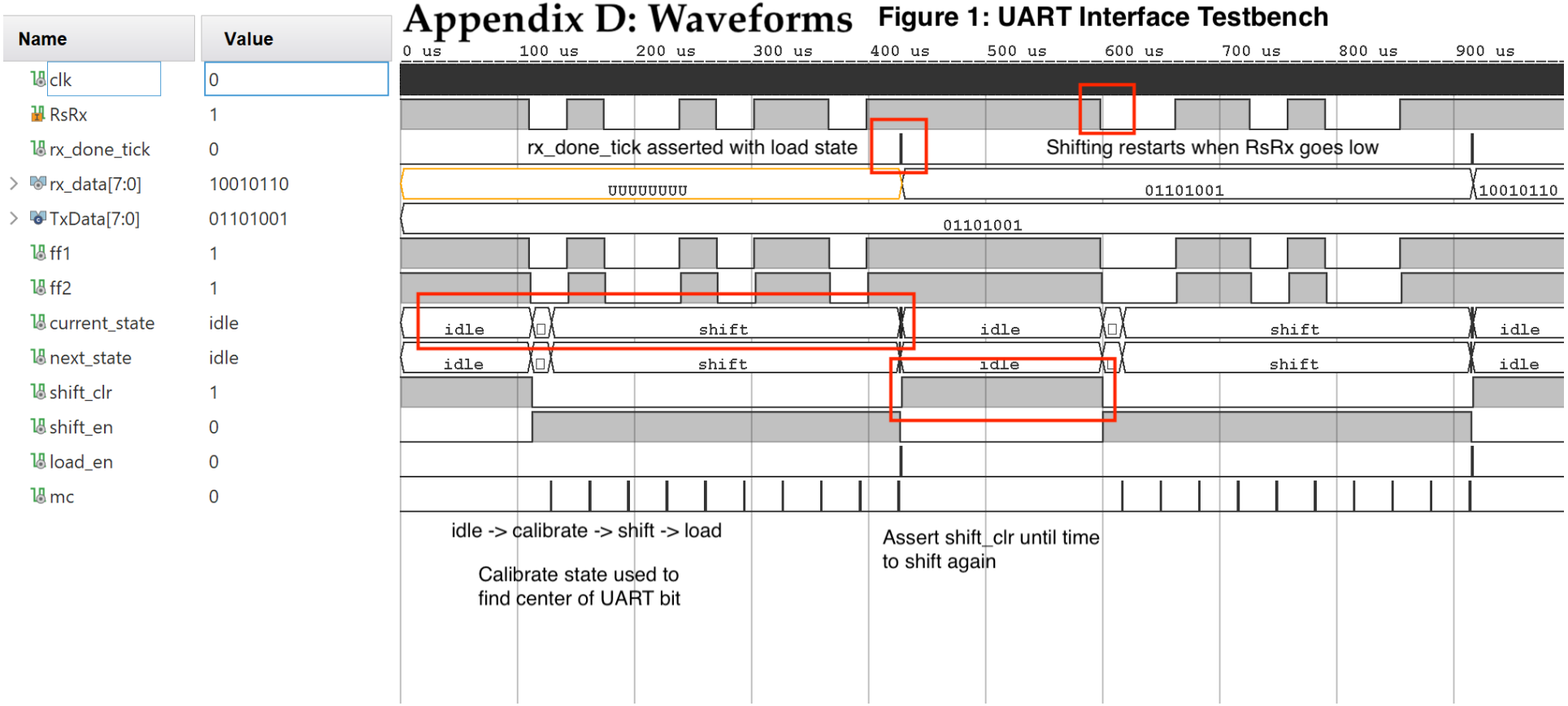


Figure 2: UART Interface Testbench - Load State

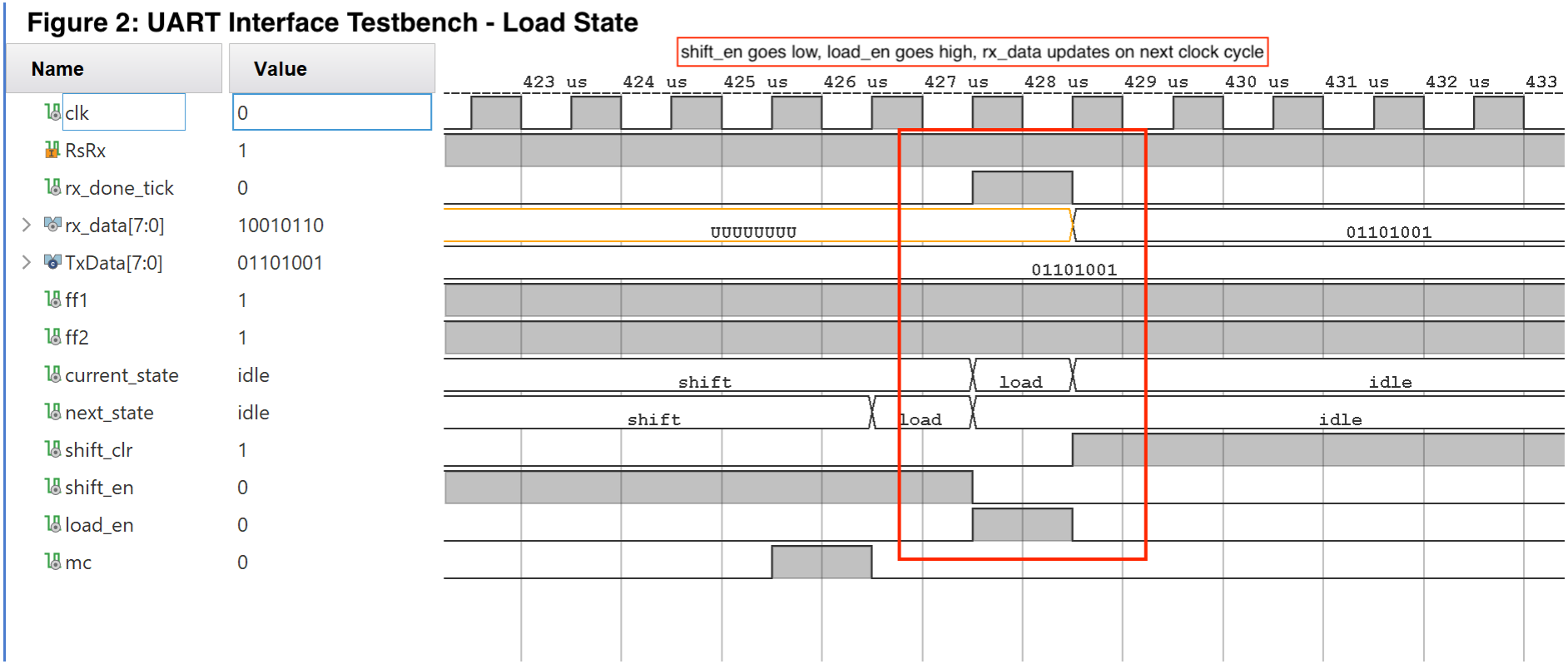


Figure 3: MIDI Interface Testbench

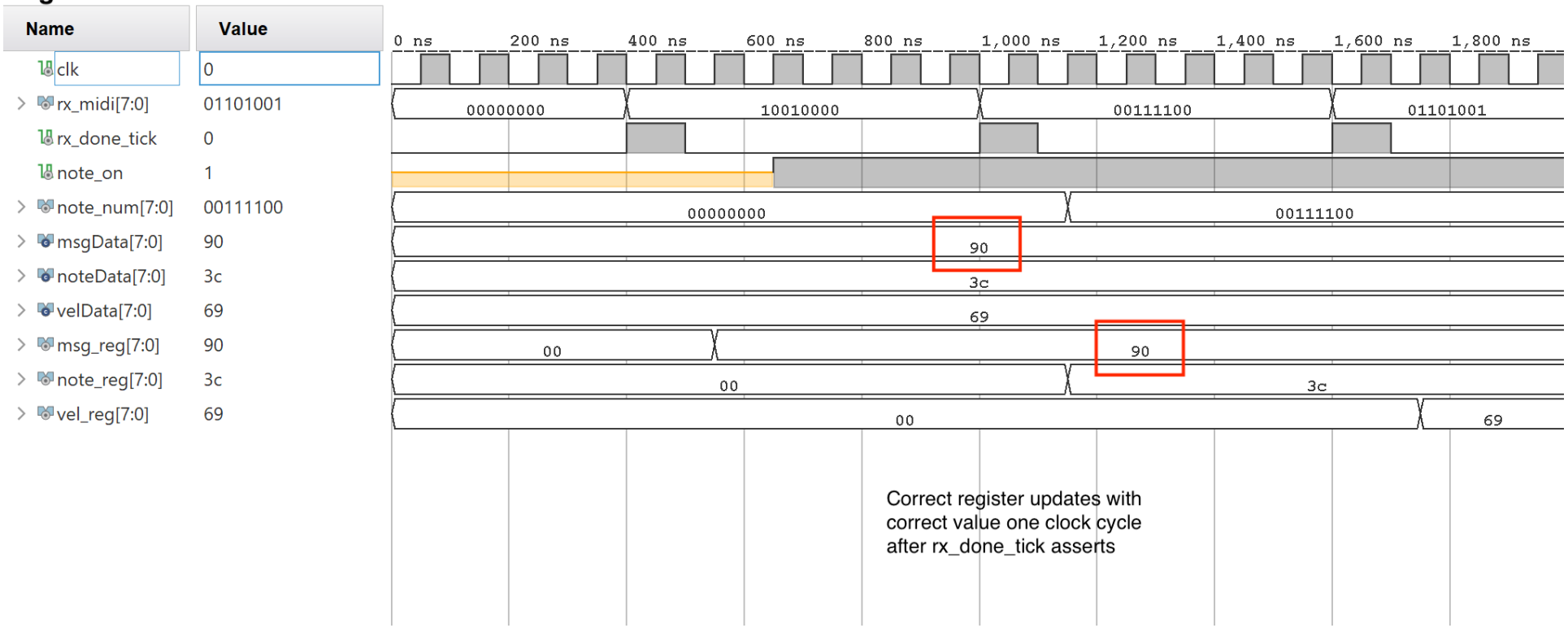
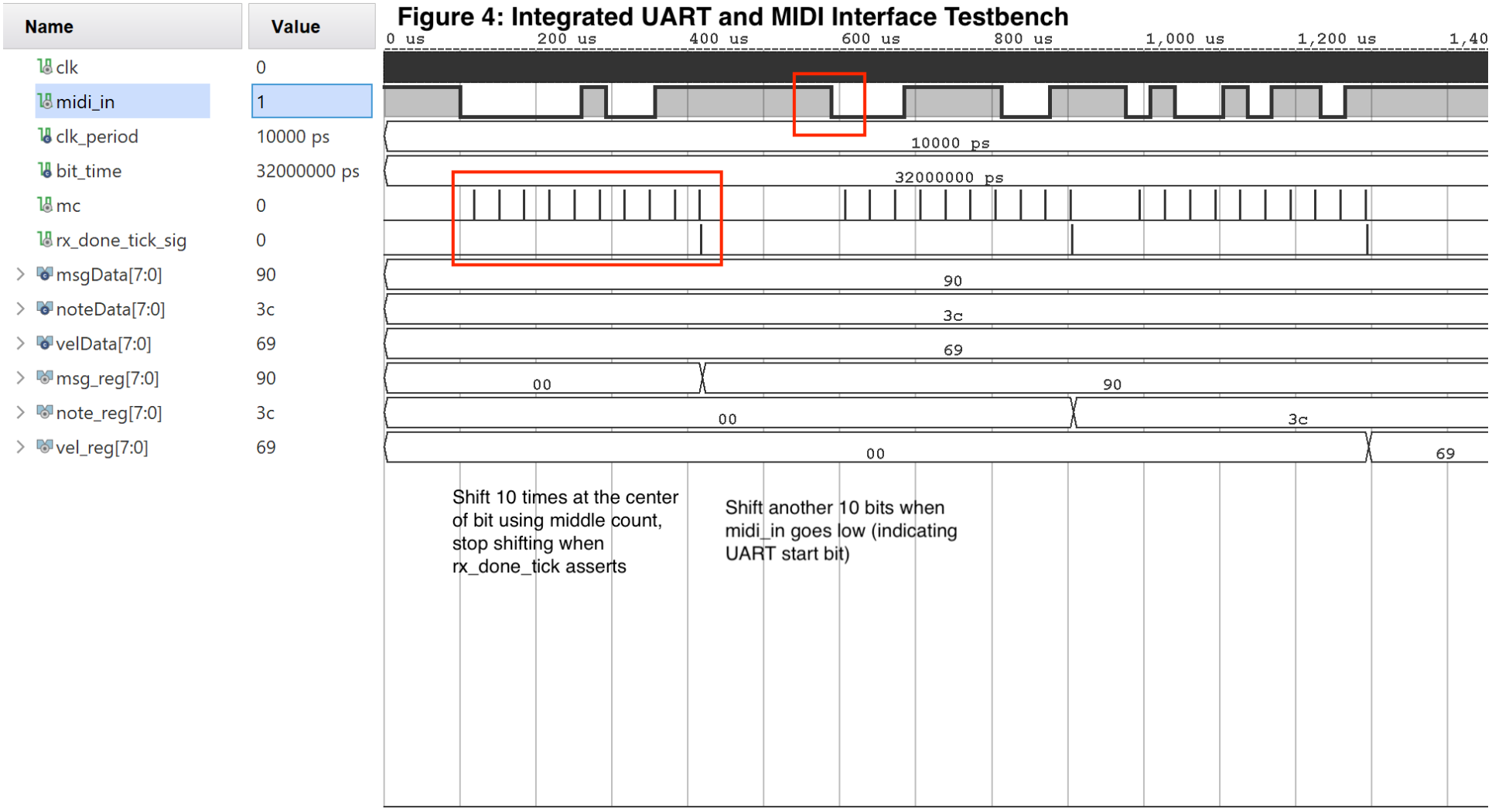
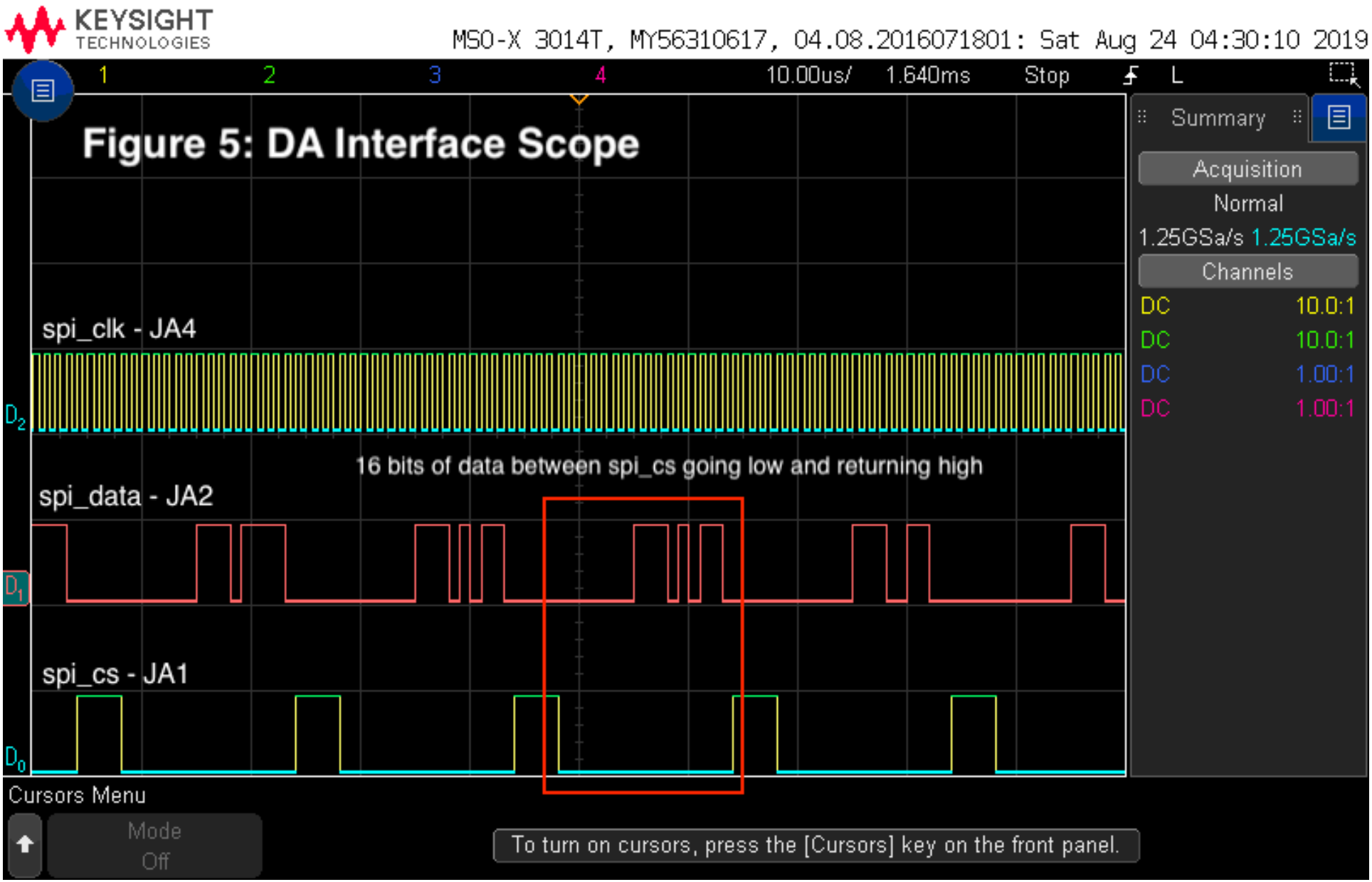
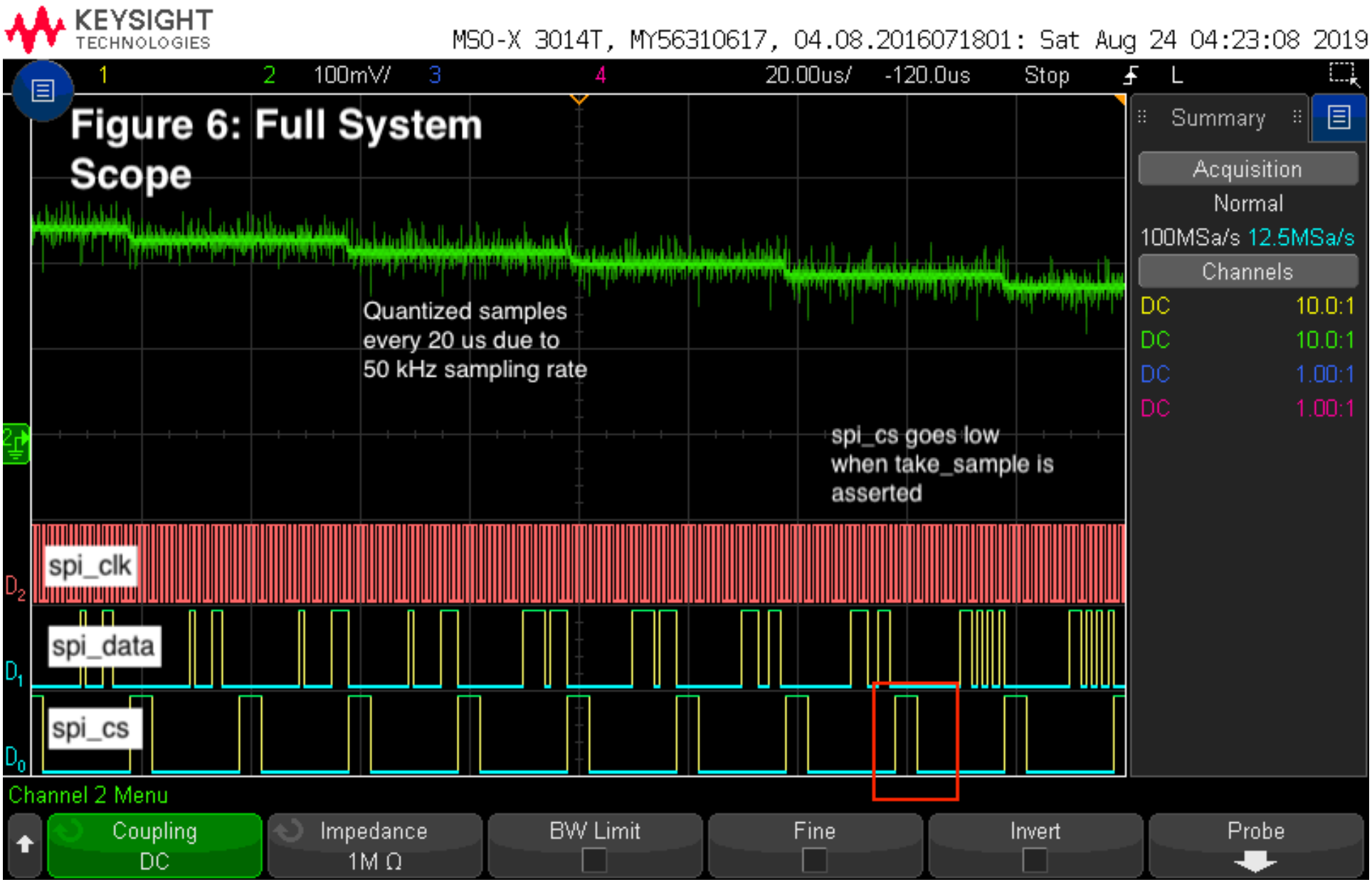
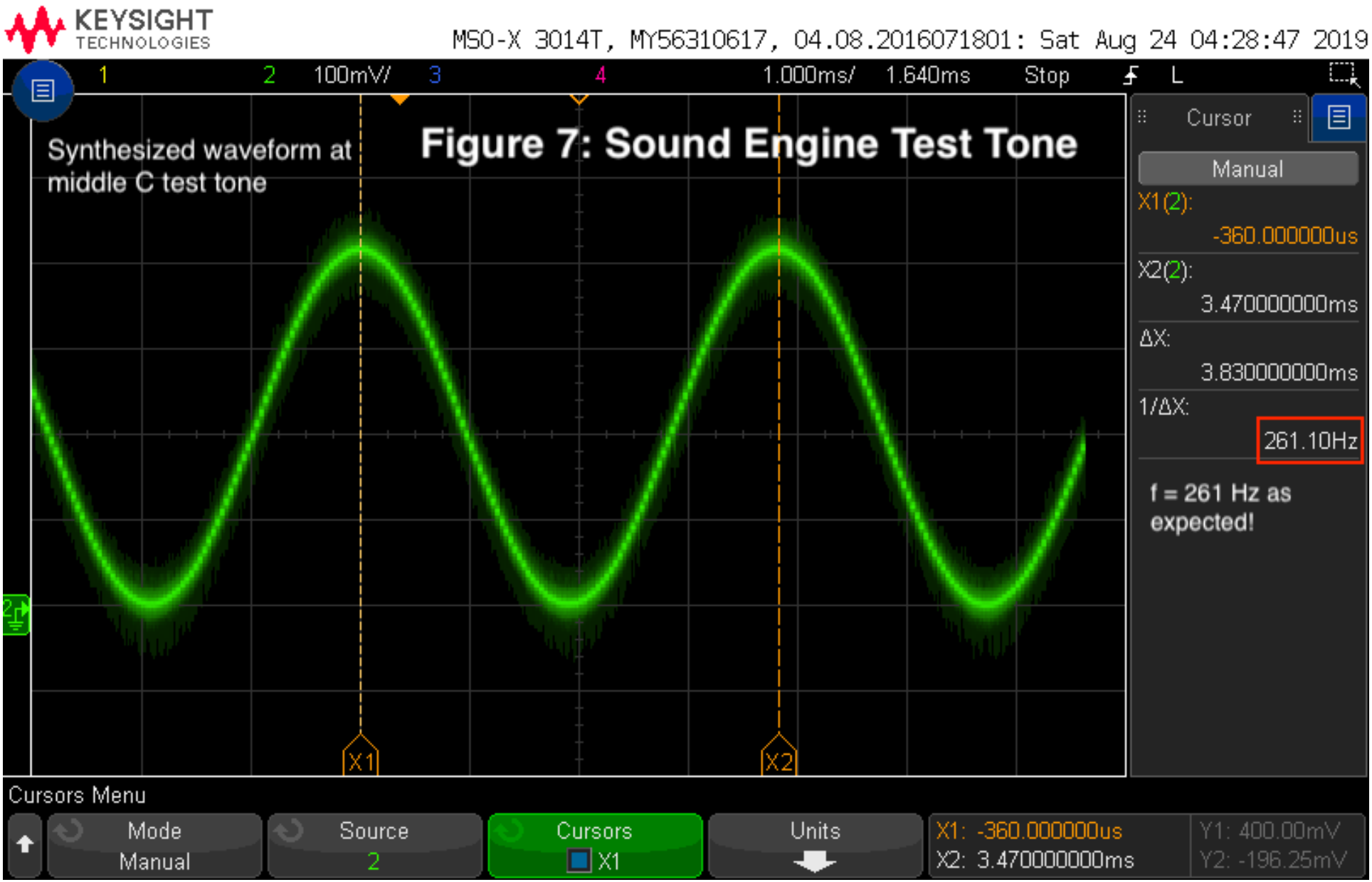


Figure 4: Integrated UART and MIDI Interface Testbench




KEYSIGHT TECHNOLOGIES
MSO-X 3014T, MY56310617, 04.08.2016071801: Sat Aug 24 04:30:10 2019
1
2
3
4
10.00us/
1.640ms
Stop
L
Figure 5: DA Interface Scope
Summary
Acquisition
Normal
1.25GSa/s 1.25GSa/s
Channels
DC 10.0:1
DC 10.0:1
DC 1.00:1
DC 1.00:1
spi_clk - JA4
D2
16 bits of data between spi_cs going low and returning high
spi_data - JA2
D1
spi_cs - JA1
D0
Cursors Menu
Mode
Off
To turn on cursors, press the [Cursors] key on the front panel.


KEYSIGHT TECHNOLOGIES
MSO-X 3014T, MY56310617, 04.08.2016071801: Sat Aug 24 04:23:08 2019
1
2
100mV/
3
4
20.00us/
-120.0us
Stop
L
Figure 6: Full System Scope
Summary
Acquisition
Normal
100MSa/s 12.5MSa/s
Channels
DC 10.0:1
DC 10.0:1
DC 1.00:1
DC 1.00:1
Quantized samples every 20 us due to 50 kHz sampling rate
spi_cs goes low when take_sample is asserted
spi_clk
spi_data
spi_cs
D2
D1
D0
Channel 2 Menu
Coupling
DC
Impedance
1M Ω
BW Limit
Fine
Invert
Probe


KEYSIGHT TECHNOLOGIES
MSO-X 3014T, MY56310617, 04.08.2016071801: Sat Aug 24 04:28:47 2019
1
2
100mV/
3
4
1.000ms/
1.640ms
Stop
L
Synthesized waveform at middle C test tone
Figure 7: Sound Engine Test Tone
Cursor
Manual
X1(2):
-360.000000us
X2(2):
3.470000000ms
ΔX:
3.830000000ms
1/ΔX:
261.10Hz
f = 261 Hz as expected!
X1
X2
Cursors Menu
Mode
Manual
Source
2
Cursors
X1
Units
X1: -360.000000us
X2: 3.470000000ms
Y1: 400.00mV
Y2: -196.25mV

Appendix E: Parts List

Listed below are the parts required to build this design and descriptions of each part. Any part listed as (TH) is built in-house at Thayer School of Engineering.

Part	**Quantity**	**Description**
FPGA Board		
Basys 3	1	FPGA board for implementing design
Pmods		
Pmod DA2	1	Digital to analog converter
Pmod AMP2	1	Audio amp with 3.5 mm port
DA2-AMP2 Interface (TH)	1	Connects DA2 and AMP2
MIDI Pmod (TH)	1	5-pin MIDI jack for Digilent Pmod setup
External Hardware		
25-Key MIDI Controller	1	Standard MIDI Keyboard
MIDI Cable	1	Standard MIDI Cable
Speaker	1	Speaker with 3.5 mm jack

```
1  ----------------------------------------------------------------------------------
…  -
2  -- Company: ENGS 31 / CoSc 56
3  -- Engineer: Grant Larson and Catherine Slaughter
4  --
5  -- Create Date: 08/19/2019 05:03:19 PM
6  -- Design Name: shell
7  -- Module Name: shell - Behavioral
8  -- Project Name: Polyphonic Digital Synth
9  -- Target Devices: Basys 3
10 -- Tool Versions:
11 -- Description: Top level shell for polyphonic digital synthesizer
12 --
13 ----------------------------------------------------------------------------------
…  -
14
15 library IEEE;
16 use IEEE.STD_LOGIC_1164.ALL;
17 use IEEE.NUMERIC_STD.ALL;
18 use IEEE.MATH_REAL.ALL;
19
20 library UNISIM;           -- Needed for the BUFG component
21 use UNISIM.Vcomponents.ALL;
22
23 entity shell is
24     port ( mclk : in std_logic;
25            midi_in : in std_logic;
26
27            -- Outputs of DA interface
28            spi_data : out std_logic;
29            spi_cs : out std_logic;
30            spi_clk : out std_logic;
31
32            -- Outputs for 7-segment displays
33            seg : out  std_logic_vector(0 to 6);
34            dp : out std_logic;
35            an : out  std_logic_vector(3 downto 0) );
36 end shell;
37
38 architecture behavioral of shell is
39
40     -- Declare component for UART interface
41     component uart_interface is
42         generic ( DATA_WIDTH : integer := 8);
43         port ( clk : in std_logic;
44                RsRx : in std_logic;
```

```
45              rx_data : out std_logic_vector(DATA_WIDTH-1 downto 0);
46              rx_done_tick : out std_logic);
47      end component uart_interface;
48
49      -- Declare component for MIDI interface
50      component midi_interface is
51          generic( MIDI_WIDTH : integer := 8;
52                   MSG_WIDTH : integer := 4 );
53          port(  clk : in std_logic;
54                 rx_midi: in std_logic_vector(MIDI_WIDTH-1 downto 0);
55                 rx_done_tick: in std_logic;
56                 note_num: out std_logic_vector(MIDI_WIDTH-1 downto 0);
57                 note_on: out std_logic;
58                 midi_done_tick : out std_logic );
59      end component midi_interface;
60
61      -- Declare component for sound engine
62      component sound_engine is
63          generic ( MIDI_WIDTH : integer := 8;
64                    AUDIO_WIDTH : integer := 12;
65                    PHASE_WIDTH : integer := 12;
66                    NUM_KEYS : integer := 25;
67                    LOWEST_NOTE_NUM : integer := 48;
68                    SINE_LATENCY : integer := 6;
69                    LUT_WIDTH : integer := 16;
70                    SUM_WIDTH : integer := 17 );
71          port ( clk : in std_logic;
72                 midi_done_tick : in std_logic;
73                 note_num : in std_logic_vector(MIDI_WIDTH-1 downto 0);
74                 note_on : in std_logic;
75                 take_sample : in std_logic;
76                 audio_data : out std_logic_vector(AUDIO_WIDTH-1 downto 0) );
77      end component sound_engine;
78
79      -- Declare component for DA interface
80      component da_interface is
81          generic ( INPUT_WIDTH : integer := 12;
82                    DA_WIDTH : integer := 16 );
83          port (    clk : in std_logic ;
84                    data_in : in std_logic_vector (INPUT_WIDTH-1 downto 0);
85                    spi_data : out std_logic;
86                    spi_cs : out std_logic;
87                    take_sample: in std_logic);
88      end component da_interface;
89
90      -- Declare component for multiplexed seven segment display
```

```
91      component mux7seg is
92          port (  clk : in  STD_LOGIC;
93                  y0, y1, y2, y3 : in  STD_LOGIC_VECTOR (3 downto 0);
94                  dp_set : in std_logic_vector(3 downto 0);
95                  seg : out  STD_LOGIC_VECTOR (0 to 6);
96                  dp : out std_logic;
97                  an : out  STD_LOGIC_VECTOR (3 downto 0) );
98      end component;
99
100     -- Constants
101     constant MIDI_WIDTH : integer := 8;
102     constant AUDIO_WIDTH : integer := 12;
103
104     -- Clock divider signals
105     constant CLK_DIVIDER_VALUE: integer := 50; -- Divide by 50 to get 2 MHz, flip
… flop gives 1 MHz
106     constant COUNT_LEN: integer := integer(ceil( log2( real(CLK_DIVIDER_VALUE) )
… ));
107     signal clkdiv: unsigned(COUNT_LEN-1 downto 0) := (others => '0');  -- Clock
… divider counter
108     signal clk_unbuf: std_logic := '0';    -- Unbuffered serial clock
109     signal clk: std_logic := '0';
110
111     -- Take sample signals
112     signal take_sample_sig : std_logic := '0';
113     constant MAX_COUNT : integer := 20; -- Divide 1 MHz down to 50 kHz
114     signal take_sample_count : unsigned(15 downto 0):= (others => '0');
115
116     -- UART to MIDI signals
117     signal rx_data_sig : std_logic_vector(MIDI_WIDTH-1 downto 0) := (others =>
… '0');
118     signal rx_done_tick_sig : std_logic := '0';
119
120     -- MIDI to sound engine signals
121     signal note_num_sig : std_logic_vector(MIDI_WIDTH-1 downto 0) := (others =>
… '0');
122     signal note_on_sig : std_logic := '0';
123     signal midi_done_tick_sig : std_logic := '0';
124
125     -- Sound engine to DA converter signals
126     signal audio_data_sig : std_logic_vector(AUDIO_WIDTH-1 downto 0);
127     signal audio_data_sig_div : std_logic_vector(AUDIO_WIDTH-1 downto 0);
128
129 begin
130     -- Instantiate UART interface
131     get_midi: uart_interface
```

```
132         generic map ( DATA_WIDTH => 8 )
133         port map (
134             clk => clk,
135             RsRx => midi_in,
136             rx_data => rx_data_sig,
137             rx_done_tick => rx_done_tick_sig );
138 
139     -- Instantiate MIDI interface
140     parse_midi: midi_interface
141         generic map ( MIDI_WIDTH => 8, MSG_WIDTH => 4)
142         port map (
143             clk => clk,
144             rx_midi => rx_data_sig,
145             rx_done_tick => rx_done_tick_sig,
146             note_num => note_num_sig,
147             note_on => note_on_sig,
148             midi_done_tick => midi_done_tick_sig );
149 
150     -- Instantiate sound engine
151     osc_bank: sound_engine
152         generic map ( MIDI_WIDTH => 8,
153                       AUDIO_WIDTH => 12,
154                       PHASE_WIDTH => 12,
155                       NUM_KEYS => 25,
156                       LOWEST_NOTE_NUM => 48,
157                       SINE_LATENCY => 6,
158                       LUT_WIDTH => 16,
159                       SUM_WIDTH => 18 )
160         port map ( clk => clk,
161                    note_num => note_num_sig,
162                    note_on => note_on_sig,
163                    midi_done_tick => midi_done_tick_sig,
164                    take_sample => take_sample_sig,
165                    audio_data => audio_data_sig );
166 
167     -- Divide audio data to decrease volume
168     audio_data_sig_div <= "00" & audio_data_sig(AUDIO_WIDTH-1 downto 2);
169 
170     -- Instantiate DA converter
171     da_converter: da_interface
172         generic map ( INPUT_WIDTH => 12,
173                       DA_WIDTH => 16 )
174         port map (
175             clk => clk,
176             data_in => audio_data_sig_div,
177             spi_data => spi_data,
```

```
178             spi_cs => spi_cs,
179             take_sample => take_sample_sig );
180 
181     -- Send along spi_clk
182     spi_clk <= clk;
183 
184     -- Instantiate 7-segment display
185     display: mux7seg port map(
186             clk => clk,       -- runs on the 1 MHz clock
187             y3 => "0000",
188             y2 => "0000",
189             y1 => note_num_sig(7 downto 4), -- Upper 4 bits
190             y0 => note_num_sig(3 downto 0), -- Lower 4 bits
191             dp_set => "0000",           -- decimal points off
192             seg => seg,
193             dp => dp,
194             an => an );
195 
196     -- Instantiate clock buffer for clk
197     -- The BUFG component puts the signal onto the FPGA clocking network
198     slow_clock_buffer: BUFG
199         port map (I => clk_unbuf,
200                   O => clk );
201 
202     -- Clock divider
203     -- Divide the 100 MHz clock down to 2 MHz, then toggling a flip flop gives
… the final
204     -- 1 MHz system clock
205     clock_divider: process(mclk)
206     begin
207         if rising_edge(mclk) then
208             if clkdiv = CLK_DIVIDER_VALUE-1 then
209                 clkdiv <= (others => '0');
210                 clk_unbuf <= NOT(clk_unbuf);
211             else
212                 clkdiv <= clkdiv + 1;
213             end if;
214         end if;
215     end process clock_divider;
216 
217     -- Take sample generator
218     take_sample_gen: process(clk,take_sample_count)
219     begin
220         if rising_edge(clk) then
221             take_sample_count <= take_sample_count + 1;
222             if take_sample_count = MAX_COUNT-1 then
```

```
223                 take_sample_count <= (others => '0');
224             end if;
225         end if;
226 
227         if take_sample_count = MAX_COUNT-1 then
228             take_sample_sig <= '1'; -- Assert take_sample when counter times out
229         else
230             take_sample_sig <= '0';
231         end if;
232     end process;
233 
234 end behavioral;
```

```
1  ----------------------------------------------------------------------------------
   -
2  -- Company: ENGS 31 / CoSc 56
3  -- Engineer: Grant Larson and Catherine Slaughter
4  --
5  -- Create Date: 08/13/2019 09:43:45 PM
6  -- Design Name: uart_interface
7  -- Module Name: uart_interface - Behavioral
8  -- Project Name: Digital Synth
9  -- Target Devices: Basys 3
10 -- Tool Versions:
11 -- Description: UART interface for recieving serial from MIDI keyboard
12 --
13 -- Revision:
14 -- Revision 0.01 - File Created
15 -- Revision 0.1 - Changed shift register to 10-bits and added logic to find
   center of clock edge
16 -- Additional Comments:
17 --
18 ----------------------------------------------------------------------------------
   -
19 library IEEE;
20 use IEEE.STD_LOGIC_1164.ALL;
21 use IEEE.NUMERIC_STD.ALL;
22
23 entity uart_interface is
24     generic ( DATA_WIDTH : integer := 8);
25     port ( clk : in std_logic;
26            RsRx : in std_logic;
27            rx_data : out std_logic_vector(DATA_WIDTH-1 downto 0);
28            rx_done_tick : out std_logic);
29 end uart_interface;
30
31 architecture behavioral of uart_interface is
32
33 -- Constants
34 constant BAUD_RATE : integer := 31250; -- UART baud rate (bits/second)
35 constant CLK_FREQ : integer := 1E6; -- Clock frequency in Hz
36 constant N : integer := CLK_FREQ/BAUD_RATE; -- Number of clock cycles per bit
37
38
39 -- Internal signals
40 signal count : unsigned(9 downto 0) := (others => '0'); -- Shift counter
41 signal shift_count: unsigned(3 downto 0) := (others => '0');
42 signal shift_reg : std_logic_vector(DATA_WIDTH+1 downto 0) := (others => '0'); --
   Include start and stop bit
```

```
43 signal ff1,ff2 : std_logic := '1'; -- Flip flop signals for double flop
44
45 -- Signals for state machine
46 type statetype is (idle,calibrate,shift,load);
47 signal current_state, next_state : statetype;
48
49 -- Control signals
50 signal shift_clr: std_logic := '0'; -- Clear for shift register and shift counter
51 signal shift_tc: std_logic := '0'; -- Terminal count for shift counter
52 signal clk_div_clr: std_logic := '0'; -- Clear for clock divider counter
53 signal shift_en: std_logic := '0'; -- Shift enable for shift register
54 signal load_en: std_logic := '0'; -- Load enable for shift register
55 signal mc: std_logic := '0'; -- Middle count for clock divider
56
57 begin
58
59 -- Double flop synchronizer
60 double_flop: process(clk)
61 begin
62     if rising_edge(clk) then
63         ff1 <= RsRx;
64         ff2 <= ff1;
65     end if;
66 end process;
67
68 -- Shift counter
69 shift_counter: process(clk,shift_count)
70 begin
71     if rising_edge(clk) then
72         if shift_clr = '1' then
73             shift_count <= (others => '0');
74         else
75             -- Increment count at center of bit
76             if mc = '1' then
77                 shift_count <= shift_count + 1;
78             end if;
79         end if;
80     end if;
81
82     -- Terminal count logic
83     if shift_count =10 then
84         shift_tc <= '1';
85     else
86         shift_tc <= '0';
87     end if;
88 end process;
```

```
89
90  -- Shift register
91  shift_register: process(clk)
92  begin
93      if rising_edge(clk) then
94          if shift_en = '1' then
95              if mc = '1' then
96                  -- Right shift in output of double flop
97                  shift_reg <= ff2 & shift_reg(DATA_WIDTH+1 downto 1);
98              end if;
99          end if;
100     end if;
101 end process;
102
103 -- Clock divider
104 clock_div: process(clk,count)
105 begin
106     if rising_edge(clk) then
107         if count = N or clk_div_clr = '1' then
108             count <= (others => '0');
109         else
110             count <= count + 1;
111         end if;
112     end if;
113
114     -- Middle count logic for finding center of bit
115     if count = N/2 then
116         mc <= '1';
117     else
118         mc <= '0';
119     end if;
120 end process;
121
122
123 -- Output register
124 output_register: process(clk)
125 begin
126     if rising_edge(clk) then
127         if load_en = '1' then
128             -- Load bit 8 downto 0 of shift register
129             rx_data <= shift_reg(DATA_WIDTH downto 1);
130         end if;
131     end if;
132 end process;
133
134 -- State update process
```

```
135 fsm_update: process(clk)
136 begin
137     if rising_edge(clk) then
138         current_state <= next_state;
139     end if;
140 end process;
141 
142 -- Next state logic
143 fsm_comb: process(current_state,ff2,RsRx,mc,shift_tc)
144 begin
145     -- Defaults
146     next_state <= current_state;
147     clk_div_clr <= '1';
148     shift_clr <= '0';
149     load_en <= '0';
150     rx_done_tick <= '0';
151     shift_en <= '0';
152 
153     -- State update logic
154     case current_state is
155         when idle =>
156             shift_clr <= '1'; -- Hold shift clear high
157             if ff2 = '0' then
158                 next_state <= calibrate;
159             end if;
160         when calibrate =>
161             clk_div_clr <= '0';
162             shift_en<='1';
163             if mc = '1' then
164                 next_state <= shift; -- Change states at center of bit
165             end if;
166         when shift =>
167             shift_en <= '1';
168             clk_div_clr <= '0';
169             if shift_tc = '1' then
170                 next_state <= load; -- Change states when shifting complete
171             end if;
172         when load =>
173             load_en <= '1';
174             rx_done_tick <= '1'; -- Signal that data byte is ready
175             next_state <= idle;
176     end case;
177 end process;
178 
179 end behavioral;
180 
```

```
1  ----------------------------------------------------------------------------------
   -
2  -- Company: ENGS 31 / CoSc 56
3  -- Engineer: Grant Larson and Catherine Slaughter
4  --
5  -- Create Date: 08/15/2019 04:41:10 PM
6  -- Design Name: midi_interface
7  -- Module Name: midi_interface - Behavioral
8  -- Project Name: Polyphonic Digital Synth
9  -- Target Devices: Basys 3
10 -- Tool Versions:
11 -- Description: Standard MIDI interface for recieving note on and note off
   messages.
12 --
13 ----------------------------------------------------------------------------------
   -
14
15 library IEEE;
16 use IEEE.STD_LOGIC_1164.ALL;
17 use IEEE.NUMERIC_STD.ALL;
18
19 entity midi_interface is
20     generic( MIDI_WIDTH : integer := 8;
21              MSG_WIDTH : integer := 4 );
22     port(  clk : in std_logic;
23            rx_midi: in std_logic_vector(MIDI_WIDTH-1 downto 0);
24            rx_done_tick: in std_logic;
25            note_num: out std_logic_vector(MIDI_WIDTH-1 downto 0);
26            note_on: out std_logic;
27            midi_done_tick : out std_logic );
28 end midi_interface;
29
30 architecture behavioral of midi_interface is
31
32 -- Constants
33 constant ON_MSG: std_logic_vector(MSG_WIDTH-1 downto 0) := "1001"; -- MIDI note
   on msg
34 constant OFF_MSG: std_logic_vector(MSG_WIDTH-1 downto 0) := "1000"; -- MIDI note
   off msg
35
36 -- State machine signals
37 type statetype is (idle,msg_load,note_load,vel_load);
38 signal current_state, next_state : statetype;
39
40 -- Internal signals
41 signal msg_reg: std_logic_vector(MIDI_WIDTH-1 downto 0); -- Store message
```

```
42 signal note_reg: std_logic_vector(MIDI_WIDTH-1 downto 0); -- Store note number
43 signal vel_reg: std_logic_vector(MIDI_WIDTH-1 downto 0); -- Store note velocity
… (unused)
44
45 -- Control signals
46 signal count : unsigned(1 downto 0) := (others => '0'); -- Counts bytes of MIDI
47 signal msg_en : std_logic := '0';
48 signal note_en : std_logic := '0';
49 signal vel_en : std_logic := '0';
50 signal count_en : std_logic := '0';
51
52 begin
53
54 -- Parse the MIDI data into appropriate register
55 midi_parse: process(clk)
56 begin
57     if rising_edge(clk) then
58         if msg_en = '1' then
59             msg_reg <= rx_midi;
60         elsif note_en = '1' then
61             note_reg <= rx_midi;
62         elsif vel_en = '1' then
63             vel_reg <= rx_midi;
64         end if;
65     end if;
66 end process;
67
68 -- Determine note_on value from encoded MIDI message
69 msg_decode: process(clk)
70 begin
71     if rising_edge(clk) then
72         if msg_reg(MIDI_WIDTH-1 downto MSG_WIDTH) = ON_MSG then
73             note_on <= '1';
74         elsif msg_reg(MIDI_WIDTH-1 downto MSG_WIDTH) = OFF_MSG then
75             note_on <= '0';
76         end if;
77     end if;
78 end process;
79
80 -- Count number of bytes to determine correct register to load
81 byte_counter: process(clk)
82 begin
83     if rising_edge(clk) then
84         if count_en = '1' then
85             if count = "10" then -- Only count up to 3 (3 registers)
86                 count<="00";
```

```
87              else
88                  count <= count + 1;
89              end if;
90          end if;
91      end if;
92  end process;
93
94  -- State update logic
95  fsm_update: process(clk)
96  begin
97      if rising_edge(clk) then
98          current_state <= next_state;
99      end if;
100 end process;
101
102 -- Next state logic
103 fsm_comb: process(current_state,rx_done_tick,count)
104 begin
105     -- Defaults
106     next_state <= current_state;
107     msg_en <= '0';
108     note_en <= '0';
109     vel_en <= '0';
110     count_en <= '0';
111     midi_done_tick <= '0';
112
113     -- Next state logic
114     case current_state is
115         when idle =>
116             if rx_done_tick = '1' then
117                 if count = "00" then
118                     next_state <= msg_load;
119                 elsif count = "01" then
120                     next_state <= note_load;
121                 else
122                     -- Need to ensure that count is never "11"
123                     next_state <= vel_load;
124                 end if;
125             end if;
126         when msg_load =>
127             count_en <= '1';
128             msg_en <= '1';
129             next_state <= idle;
130         when note_load =>
131             count_en <= '1';
132             note_en <= '1';
```

```
133             next_state <= idle;
134         when vel_load =>
135             count_en <= '1';
136             vel_en <= '1';
137             next_state <= idle;
138             midi_done_tick <= '1'; -- Packet of data is ready
139     end case;
140 end process;
141 
142 note_num <= note_reg; -- Output value of note_reg
143 
144 end behavioral;
145 
```

```
1  ----------------------------------------------------------------------------------
   -
2  -- Company: ENGS 31 / CoSc 56
3  -- Engineer: Grant Larson and Catherine Slaughter
4  --
5  -- Create Date: 08/19/2019 05:03:19 PM
6  -- Design Name: sound_engine
7  -- Module Name: sound_engine - Behavioral
8  -- Project Name: Polyphonic Digital Synth
9  -- Target Devices: Basys 3
10 -- Tool Versions:
11 -- Description: Polyphonic sound engine for polyphonic digital synthesizer
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
   -
20
21 library IEEE;
22 use IEEE.STD_LOGIC_1164.ALL;
23 use IEEE.NUMERIC_STD.ALL;
24
25 entity sound_engine is
26     generic ( MIDI_WIDTH : integer := 8;
27               AUDIO_WIDTH : integer := 12;
28               PHASE_WIDTH : integer := 12;
29               NUM_KEYS : integer := 25;
30               LOWEST_NOTE_NUM : integer := 48;
31               SINE_LATENCY : integer := 6;
32               LUT_WIDTH : integer := 16;
33               SUM_WIDTH : integer := 18 );
34     port ( clk : in std_logic;
35            midi_done_tick : in std_logic;
36            note_num : in std_logic_vector(MIDI_WIDTH-1 downto 0);
37            note_on : in std_logic;
38            take_sample : in std_logic;
39            audio_data : out std_logic_vector(AUDIO_WIDTH-1 downto 0) );
40 end sound_engine;
41
42 architecture behavioral of sound_engine is
43
44     -- Input register signals
```

```
45      signal note_on_reg : std_logic_vector(NUM_KEYS-1 downto 0) := (others =>
…   '0');
46
47      -- Declare phase accumulator as component
48      component pa is
49          generic (M : integer; -- Change this value for different test tone
50                   PHASE_WIDTH : integer );
51          port (
52                clk : in std_logic;
53                note_on : in std_logic;
54                take_sample : in std_logic;
55                phase : out std_logic_vector(11 downto 0));
56      end component pa;
57
58
59      -- Create k to M array
60      type init_M_arr is array(NUM_KEYS-1 downto 0) of integer;
61      constant M_arr : init_M_arr := (
62          0  => 10973,
63          1  => 11626,
64          2  => 12317,
65          3  => 13049,
66          4  => 13825,
67          5  => 14647,
68          6  => 15518,
69          7  => 16441,
70          8  => 17419,
71          9  => 18455,
72          10 => 19552,
73          11 => 20715,
74          12 => 21946,
75          13 => 23251,
76          14 => 24634,
77          15 => 26099,
78          16 => 27651,
79          17 => 29295,
80          18 => 31037,
81          19 => 32882,
82          20 => 34838,
83          21 => 36909,
84          22 => 39104,
85          23 => 41429,
86          24 => 43893
87          );
88
89      -- Declare signals for state machine
```

```
90      type statetype is (idle,count,load);
91      signal current_state, next_state : statetype;
92
93      -- Control signals
94      signal count_en : std_logic := '0';
95      signal load_en : std_logic := '0';
96      signal accum_clr : std_logic := '0';
97
98      -- Declare signals for phase accumulator array
99      type phase_arr is array(NUM_KEYS-1 downto 0) of
… std_logic_vector(PHASE_WIDTH-1 downto 0);
100     signal phase_mux : phase_arr := (others => (others => '0'));
101
102     -- Counter signals
103     signal count_mux : unsigned(4 downto 0) := (others => '0');
104     signal count_mux_tc : std_logic := '0';
105
106     -- Sine delay signals
107     signal count_en_delay_reg : std_logic_vector(SINE_LATENCY-1 downto 0) :=
… (others => '0');
108     signal load_en_delay_reg : std_logic_vector(SINE_LATENCY-1 downto 0) :=
… (others => '0');
109     signal accum_clr_delay_reg : std_logic_vector(SINE_LATENCY-1 downto 0) :=
… (others => '0');
110     signal count_en_delay : std_logic := '0';
111     signal load_en_delay : std_logic := '0';
112     signal accum_clr_delay : std_logic := '0';
113
114     -- Internal signals
115     signal phase_sig : std_logic_vector(PHASE_WIDTH-1 downto 0) := (others =>
… '0');
116     signal lut_phase_sig : std_logic_vector(LUT_WIDTH-1 downto 0) := (others =>
… '0');
117     signal sine_sig : std_logic_vector(PHASE_WIDTH-1 downto 0) := (others =>
… '0');
118     signal lut_sine_sig : std_logic_vector(LUT_WIDTH-1 downto 0) := (others =>
… '0');
119
120
121     -- Declare component for sine wave LUT
122     component sine_wave
123       port (
124         aclk : IN STD_LOGIC;
125         s_axis_phase_tvalid : in std_logic;
126         s_axis_phase_tdata : in std_logic_vector(15 DOWNTO 0);
127         m_axis_data_tvalid : out std_logic;
```

```
128         m_axis_data_tdata : out std_logic_vector(15 DOWNTO 0) );
129     end component;
130 
131     --Declare signals for sine accumulator
132     signal sine_sum : signed(SUM_WIDTH-1 downto 0) := (others => '0'); --Large
… enough to hold 25 12-bit numbers added together
133     signal sine_sum_reg : signed(SUM_WIDTH-1 downto 0) := (others => '0');
134 
135     --Declare signal for dividing sine accum output
136     signal keys_pressed : integer := 0;
137 
138 begin
139 
140     -- Input register and keys pressed counter
141     input_reg : process(clk)
142     begin
143         if rising_edge(clk) then
144             if midi_done_tick = '1' then
145                 note_on_reg(to_integer(unsigned(note_num)) - LOWEST_NOTE_NUM) <=
… note_on;
146                 if note_on = '1' then
147                     keys_pressed <= keys_pressed + 1; -- Increment keys_pressed
148                 else
149                     keys_pressed <= keys_pressed - 1; -- Decrement keys_pressed
150                 end if;
151             end if;
152         end if;
153     end process;
154 
155     -- Generate statement for phase accumulators
156     accumulators : for k in 0 to (NUM_KEYS-1) generate
157     begin
158 
159         phase_accumulator: pa
160             generic map (  M  => M_arr(k),-- M value for 440 Hz at 50 kHz
… sampling rate
161                            PHASE_WIDTH => 22 ) -- Shift left by 10 to multiply by
… 1024
162             port map (
163                 clk => clk,
164                 note_on => note_on_reg(k),
165                 take_sample => take_sample,
166                 phase => phase_mux(k) );
167     end generate accumulators;
168 
169     -- State update logic
```

```
170     fsm_update: process(clk)
171     begin
172         if rising_edge(clk) then
173             current_state <= next_state;
174         end if;
175     end process;
176 
177     -- Next state logic
178     fsm_comb: process(current_state, take_sample, count_mux_tc)
179     begin
180         -- Defaults
181         next_state <= current_state;
182         count_en <= '0';
183         load_en <= '0';
184         accum_clr <= '0';
185 
186         -- Next state logic
187         case current_state is
188             when idle =>
189                 accum_clr <= '1';
190                 if take_sample = '1' then
191                     next_state <= count;
192                 end if;
193             when count =>
194                 count_en <= '1';
195                 if count_mux_tc = '1' then
196                     next_state <= load;
197                 end if;
198             when load =>
199                 load_en <= '1';
200                 next_state <= idle;
201         end case;
202     end process;
203 
204     -- Counter for multiplexer
205     counter_mux: process(clk,count_mux)
206     begin
207         if rising_edge(clk) then
208             if count_en = '1' then
209                 if count_mux = NUM_KEYS-1 then
210                     count_mux <= (others => '0');
211                 else
212                     count_mux <= count_mux + 1;
213                 end if;
214             end if;
215         end if;
```

```
216
217         -- Count up to number of keys
218         if count_mux = NUM_KEYS-1 then
219             count_mux_tc <= '1';
220         else
221             count_mux_tc <= '0';
222         end if;
223     end process;
224
225     -- Shift register for waiting out sine LUT latency
226     sine_delay: process(clk)
227     begin
228         if rising_edge(clk) then
229             count_en_delay_reg <= count_en & count_en_delay_reg(SINE_LATENCY-1
… downto 1);
230             load_en_delay_reg <= load_en & load_en_delay_reg(SINE_LATENCY-1
… downto 1);
231             accum_clr_delay_reg <= accum_clr & accum_clr_delay_reg(SINE_LATENCY-1
… downto 1);
232         end if;
233     end process;
234
235     -- Get LSB of delay register
236     count_en_delay <= count_en_delay_reg(0);
237     load_en_delay <= load_en_delay_reg(0);
238     accum_clr_delay <= accum_clr_delay_reg(0);
239
240     -- Multiplex phase based on counter
241     phase_sig <= phase_mux(to_integer(count_mux));
242
243     -- Instantiate the sine wave ROM
244     sine_wave_LUT : sine_wave
245         port map (
246             aclk => clk,
247             s_axis_phase_tvalid => '1',
248             s_axis_phase_tdata => lut_phase_sig,
249             m_axis_data_tdata => lut_sine_sig );
250
251     -- Adapt LUT inputs and outputs to desired width
252     lut_phase_sig <= "0000" & phase_sig;
253     sine_sig <= lut_sine_sig(PHASE_WIDTH-1 downto 0);
254
255     -- Add up sine values produced by each accumulator
256     sine_accumulator: process(clk)
257     begin
258         if rising_edge(clk) then
```

```
259             if count_en_delay = '1' then
260                 sine_sum <= sine_sum + signed(sine_sig);
261             end if;
262             if accum_clr_delay = '1' then
263                 sine_sum <= (others => '0');
264             end if;
265         end if;
266     end process;
267 
268     sine_accum_reg: process(clk)
269     begin
270         if rising_edge(clk) then
271             if load_en_delay = '1' then
272 
273 -- The commented code below is a coarse method of dividing by sqrt(keys_pressed). We
274 -- decide not to implement this because dividing by a number less than keys_pressed
275 -- caused our sine_sum_reg to overflow and produce noise. Implementing this would be a
276 -- desirable improvement to our design.
277 --                 case keys_pressed is
278 --                     when 1 =>
279 --                         sine_sum_reg <= sine_sum;
280 --                     when 2 to 4 =>
281 --                         sine_sum_reg <= shift_right(sine_sum,1);
282 --                     when 5 to 16 =>
283 --                         sine_sum_reg <= shift_right(sine_sum,2);
284 --                     when 17 to 25 =>
285 --                         sine_sum_reg <= shift_right(sine_sum,3);
286 --                     when others =>
287 --                         sine_sum_reg <= (others => '0'); -- Clear the register
288 --                 end case;
289 
290                 sine_sum_reg <= sine_sum/keys_pressed; -- Divide by keys_pressed
291             end if;
292         end if;
293     end process;
294 
295     -- Flip MSB for desired sine wave output
296     audio_data <= not(std_logic(sine_sum_reg(AUDIO_WIDTH-1))) & std_logic_vector(sine_sum_reg(AUDIO_WIDTH-2 downto 0));
297 
298 end behavioral;
299 
```

```
1  ----------------------------------------------------------------------------------
2  -- Company: ENGS 31 / CoSc 56
3  -- Engineer: Grant Larson and Catherine Slaughter
4  --
5  -- Create Date: 08/14/2019 06:14:16 PM
6  -- Design Name: phase_accumulator
7  -- Module Name: pa - Behavioral
8  -- Project Name: Digital Synth
9  -- Target Devices: Basys 3
10 -- Tool Versions:
11 -- Description: Phase accumulator module for digital synth
12 --
13 -- Additional Comments: Conversion from 22 to 12 bit precision happens within the
…  phase.
14 --
15 ----------------------------------------------------------------------------------
16
17 library IEEE;
18 use IEEE.STD_LOGIC_1164.ALL;
19 use IEEE.NUMERIC_STD.ALL;
20
21 entity pa is
22     generic ( M : in integer;
23               PHASE_WIDTH : integer );
24     port ( clk : in std_logic;
25            note_on : in std_logic;
26            take_sample : in std_logic;
27            phase : out std_logic_vector(11 downto 0)); -- Hard code phase as
…  12-bit
28 end entity pa;
29
30 architecture behavioral of pa is
31
32 -- Internal signals
33 -- Wide phase for high precision M values
34 signal phase_sig : unsigned(PHASE_WIDTH-1 downto 0) := (others => '0');
35
36 begin
37
38 -- Phase accumulator
39 accumulator: process(clk)
40 begin
41     if rising_edge(clk) then
42         if note_on = '1' then
43             if take_sample = '1' then
44                 phase_sig <= phase_sig + to_unsigned(M,PHASE_WIDTH);
```

```
            end if;
        else
            phase_sig <= (others => '0');
        end if;
    end if;
end process;

-- Cast accumulator value as std_logic_vector for output
-- Shift right 10 bits to divide by 1024 and retain 12 bit output
phase <= std_logic_vector(phase_sig(PHASE_WIDTH-1 downto 10));

end behavioral;
```

```
1  ----------------------------------------------------------------------------------
…  -
2  -- Company: ENGS 31 / CoSc 56
3  -- Engineer: Grant Larson and Catherine Slaughter
4  --
5  -- Create Date: 08/18/2019 01:47:12 PM
6  -- Design Name: da_interface
7  -- Module Name: da_interface - Behavioral
8  -- Project Name: Polyphonic Digital Synth
9  -- Target Devices: Basys 3, PMOD DA2
10 -- Tool Versions:
11 -- Description: Serial interface for Digilent PMOD DA2
12 --
13 ----------------------------------------------------------------------------------
…  -
14
15 library IEEE;
16 use IEEE.STD_LOGIC_1164.ALL;
17 use IEEE.NUMERIC_STD.ALL;
18
19 entity da_interface is
20     generic ( INPUT_WIDTH : integer := 12;
21               DA_WIDTH : integer := 16 );
22     port (    clk : in std_logic ;
23               data_in : in std_logic_vector (INPUT_WIDTH-1 downto 0);
24               spi_data : out std_logic;
25               spi_cs : out std_logic;
26               take_sample : in std_logic );
27 end da_interface;
28
29 architecture behavioral of da_interface is
30
31 -- State machine signals
32 type statetype is (idle,load,shift);
33 signal current_state, next_state: statetype;
34
35 -- Control signals
36 signal load_en : std_logic := '0';
37 signal shift_en : std_logic := '1';
38 signal tc : std_logic := '0';
39
40 -- Internal signals
41 signal count : unsigned(7 downto 0) := (others => '0'); -- For shift counter
42 signal shift_reg : std_logic_vector(DA_WIDTH-1 downto 0) := (others => '0');
43
44 begin
```

```
45
46  shift_register: process(clk)
47  begin
48      if rising_edge(clk) then
49          if shift_en = '1' then
50              shift_reg <= shift_reg(DA_WIDTH-2 downto 0) & '0'; -- Shift MSB first
51          end if;
52
53          if load_en = '1' then
54              shift_reg <= "0000" & data_in; -- Parallel load input data
55          end if;
56      end if;
57  end process;
58
59  shift_counter: process(clk,count)
60  begin
61      if rising_edge(clk) then
62          if shift_en = '1' then
63              count <= count + 1;
64          else
65              count <= (others => '0');
66          end if;
67      end if;
68
69      -- Shift DA_WIDTH times
70      if count = DA_WIDTH-1 then
71          tc <= '1';
72      else
73          tc <= '0';
74      end if;
75  end process;
76
77  -- State update logic
78  fsm_update: process(clk)
79  begin
80      if rising_edge(clk) then
81          current_state <= next_state;
82      end if;
83  end process;
84
85  -- Next state logic
86  fsm_comb: process(current_state,tc,take_sample)
87  begin
88      -- Defaults
89      next_state <= current_state;
90      load_en <= '0';
```

```
91      shift_en <= '0';
92      spi_cs <= '1'; -- Low-true chip select (should be 1!)
93
94      -- Next state logic
95      case current_state is
96          when idle =>
97              if take_sample='1' then
98                  next_state <= load;
99              end if;
100         when load =>
101             load_en <= '1';
102             next_state <= shift;
103         when shift =>
104             spi_cs <= '0';
105             shift_en <= '1';
106             if tc = '1' then
107                 next_state <= idle; -- Return to idle when shifting complete
108             end if;
109     end case;
110 end process;
111
112 -- Output MSB on serial data line
113 spi_data <= shift_reg(DA_WIDTH-1);
114
115 end behavioral;
116
```

```
## This file is a .xdc for Grant Larson and Catherine Slaughter's polyphonic synthesizer

# Clock signal
#Bank = 34, Pin name = CLK,         Sch name = CLK100MHZ
set_property PACKAGE_PIN W5 [get_ports mclk]
set_property IOSTANDARD LVCMOS33 [get_ports mclk]
create_clock -period 20.000 -name sys_clk_pin -waveform {0.000 10.000} -add [get_ports mclk]


##7 segment display
set_property PACKAGE_PIN W7 [get_ports {seg[0]}]
  set_property IOSTANDARD LVCMOS33 [get_ports {seg[0]}]
set_property PACKAGE_PIN W6 [get_ports {seg[1]}]
  set_property IOSTANDARD LVCMOS33 [get_ports {seg[1]}]
set_property PACKAGE_PIN U8 [get_ports {seg[2]}]
  set_property IOSTANDARD LVCMOS33 [get_ports {seg[2]}]
set_property PACKAGE_PIN V8 [get_ports {seg[3]}]
  set_property IOSTANDARD LVCMOS33 [get_ports {seg[3]}]
set_property PACKAGE_PIN U5 [get_ports {seg[4]}]
  set_property IOSTANDARD LVCMOS33 [get_ports {seg[4]}]
set_property PACKAGE_PIN V5 [get_ports {seg[5]}]
  set_property IOSTANDARD LVCMOS33 [get_ports {seg[5]}]
set_property PACKAGE_PIN U7 [get_ports {seg[6]}]
  set_property IOSTANDARD LVCMOS33 [get_ports {seg[6]}]

set_property PACKAGE_PIN V7 [get_ports dp]
  set_property IOSTANDARD LVCMOS33 [get_ports dp]

set_property PACKAGE_PIN U2 [get_ports {an[0]}]
  set_property IOSTANDARD LVCMOS33 [get_ports {an[0]}]
set_property PACKAGE_PIN U4 [get_ports {an[1]}]
  set_property IOSTANDARD LVCMOS33 [get_ports {an[1]}]
set_property PACKAGE_PIN V4 [get_ports {an[2]}]
  set_property IOSTANDARD LVCMOS33 [get_ports {an[2]}]
set_property PACKAGE_PIN W4 [get_ports {an[3]}]
  set_property IOSTANDARD LVCMOS33 [get_ports {an[3]}]


##Pmod Header JA
##Sch name = JA1
set_property PACKAGE_PIN J1 [get_ports spi_cs]
  set_property IOSTANDARD LVCMOS33 [get_ports spi_cs]
##Sch name = JA2
set_property PACKAGE_PIN L2 [get_ports spi_data]
```

```
  set_property IOSTANDARD LVCMOS33 [get_ports spi_data]
##Sch name = JA3
#set_property PACKAGE_PIN J2 [get_ports spi_data]
  #set_property IOSTANDARD LVCMOS33 [get_ports spi_data]
##Sch name = JA4
set_property PACKAGE_PIN G2 [get_ports spi_clk]
  set_property IOSTANDARD LVCMOS33 [get_ports spi_clk]


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

##Pmod Header JC
##Sch name = JC1
#set_property PACKAGE_PIN K17 [get_ports note_on]
  #set_property IOSTANDARD LVCMOS33 [get_ports note_on]

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]

```

```
1  ----------------------------------------------------------------------------------
…  -
2  -- Company: ENGS 31 / CoSc 56
3  -- Engineer: Grant Larson and Catherine Slaughter
4  --
5  -- Create Date: 08/13/2019 09:43:45 PM
6  -- Design Name: uart_interface
7  -- Module Name: uart_interface_tb
8  -- Project Name: Digital Synth
9  -- Target Devices: Basys 3
10 -- Tool Versions:
11 -- Description: Testbench for standard UART interface
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments: Adapted from SPI testbench by Eric Hansen
18 --
19 ----------------------------------------------------------------------------------
…  -
20
21 LIBRARY ieee;
22 USE ieee.std_logic_1164.ALL;
23 USE ieee.numeric_std.all;
24
25 entity uart_interface_tb is
26 end uart_interface_tb;
27
28 architecture testbench of uart_interface_tb is
29
30     component uart_interface is
31         generic ( DATA_WIDTH : integer := 8);
32         port ( clk : in std_logic;
33                RsRx : in std_logic;
34                rx_data : out std_logic_vector(DATA_WIDTH-1 downto 0);
35                rx_done_tick : out std_logic);
36     end component uart_interface;
37
38     -- Inputs
39     signal clk : std_logic := '0';
40     signal RsRx : std_logic := '1';
41
42     -- Outputs
43     signal rx_done_tick : std_logic := '0';
44     signal rx_data : std_logic_vector(7 downto 0) := (others => '0');
```

```
45
46      -- Clock period definitions
47      constant clk_period : time := 1 us;   -- 1 us for 1 MHz clock
48
49    -- Data definitions
50    -- constant bit_time : time := 104us;   -- 9600 baud
51    -- constant bit_time : time := 8.68us;    -- 115,200 baud
52    constant bit_time : time := 32us;
53    constant TxData : std_logic_vector(7 downto 0) := "01101001";
54
55  begin
56
57      -- Instantiate the unit under test
58      uut: uart_interface
59          port map(
60             clk => clk,
61             RsRx => RsRx,
62             rx_data => rx_data,
63             rx_done_tick => rx_done_tick );
64
65     -- Clock process definitions
66     clk_process :process
67     begin
68      clk <= '0';
69      wait for clk_period/2;
70      clk <= '1';
71      wait for clk_period/2;
72     end process;
73
74     -- Stimulus process
75     stim_proc: process
76     begin
77      wait for 100 us;
78      wait for 10.25*clk_period;
79
80      RsRx <= '0';    -- Start bit
81      wait for bit_time;
82
83      -- Shift in 8 bits of arbitrary data
84      for bitcount in 0 to 7 loop
85        RsRx <= TxData(bitcount);
86        wait for bit_time;
87      end loop;
88
89      RsRx <= '1';    -- Stop bit
90      wait for 200 us;
```

```
91
92      RsRx <= '0';     -- Start bit
93      wait for bit_time;
94
95      -- Shift in another 8 bits of arbitrary data
96      for bitcount in 0 to 7 loop
97        RsRx <= not( TxData(bitcount) );
98        wait for bit_time;
99      end loop;
100
101     RsRx <= '1';     -- Stop bit
102
103     wait;
104    end process;
105 end testbench;
```

```
1  ----------------------------------------------------------------------------------
…  -
2  -- Company: ENGS 31 / CoSc 56
3  -- Engineer: Grant Larson and Catherine Slaughter
4  --
5  -- Create Date: 08/13/2019 09:43:45 PM
6  -- Design Name: midi_interface
7  -- Module Name: midi_interface_tb
8  -- Project Name: Digital Synth
9  -- Target Devices: Basys 3
10 -- Tool Versions:
11 -- Description: Testbench for standard MIDI interface
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments: Adapted from SPI testbench by Eric Hansen
18 --
19 ----------------------------------------------------------------------------------
…  -
20
21 LIBRARY ieee;
22 USE ieee.std_logic_1164.ALL;
23 USE ieee.numeric_std.all;
24
25 entity midi_interface_tb is
26     generic( MIDI_WIDTH : integer := 8;
27              MSG_WIDTH : integer := 4 );
28 end midi_interface_tb;
29
30 architecture testbench of midi_interface_tb is
31
32     component midi_interface is
33         generic( MIDI_WIDTH : integer := 8;
34                  MSG_WIDTH : integer := 4 );
35         port(  clk : in std_logic;
36                rx_midi: in std_logic_vector(MIDI_WIDTH-1 downto 0);
37                rx_done_tick: in std_logic;
38                note_num: out std_logic_vector(MIDI_WIDTH-1 downto 0);
39                note_on: out std_logic );
40     end component midi_interface;
41
42     -- Inputs
43     signal clk : std_logic := '0';
44     signal rx_midi : std_logic_vector(MIDI_WIDTH-1 downto 0) := (others => '0');
```

```
45      signal rx_done_tick : std_logic := '0';
46
47      -- Outputs
48      signal note_on : std_logic := '0';
49      signal note_num : std_logic_vector(MIDI_WIDTH-1 downto 0) := (others => '0');
50
51      -- Clock period definitions
52      constant clk_period : time := 100 ns;   -- 100 ns for 10 MHz clock
53
54    -- Data definitions
55    -- constant bit_time : time := 104us;    -- 9600 baud
56    -- constant bit_time : time := 8.68us;    -- 115,200 baud
57    constant bit_time : time := 32us;
58    constant msgData : std_logic_vector(7 downto 0) := "10010000";
59    constant noteData : std_logic_vector(7 downto 0) := "00111100";
60    constant velData : std_logic_vector(7 downto 0) := "01101001";
61
62  begin
63    -- Instantiate the Unit Under Test (UUT)
64
65      uut: midi_interface
66          port map(
67             clk => clk,
68             rx_midi => rx_midi,
69             rx_done_tick => rx_done_tick,
70             note_num => note_num,
71             note_on => note_on);
72
73     -- Clock process definitions
74     clk_process :process
75     begin
76      clk <= '0';
77      wait for clk_period/2;
78      clk <= '1';
79      wait for clk_period/2;
80     end process;
81
82     -- Stimulus process
83     stim_proc: process
84     begin
85      wait for 4*clk_period;
86
87      -- Test overall MIDI functionality
88      rx_midi <= msgData;
89      rx_done_tick <= '1';
90      wait for clk_period;
```

```
91      rx_done_tick <= '0';
92      wait for clk_period*5;
93
94      rx_midi <= noteData;
95      rx_done_tick <= '1';
96      wait for clk_period;
97      rx_done_tick <= '0';
98      wait for clk_period*5;
99
100     rx_midi <= velData;
101     rx_done_tick <= '1';
102     wait for clk_period;
103     rx_done_tick <= '0';
104     wait for clk_period*5;
105
106     wait;
107    end process;
108 end testbench;
```

```
----------------------------------------------------------------------------------
-
-- Company: Grant Larson and Catherine Slaughter
-- Engineer: ENGS 31 / CoSc 56
--
-- Create Date: 08/19/2019 03:01:48 PM
-- Design Name: midi_shell
-- Module Name: midi_shell_tb - Behavioral
-- Project Name: Polyphonic Digital Synthesizer
-- Target Devices:
-- Tool Versions:
-- Description: Testbench for standard MIDI interface
--
----------------------------------------------------------------------------------
-

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity midi_shell_tb is
end midi_shell_tb;

architecture testbench of midi_shell_tb is

    -- Declare MIDI shell as component
    component midi_shell is
        generic ( MIDI_WIDTH : integer := 8 );
        port ( mclk : in std_logic;
           midi_in : in std_logic;
           note_on : out std_logic;
           seg : out  STD_LOGIC_VECTOR (0 to 6);
           dp : out std_logic;
           an : out  STD_LOGIC_VECTOR (3 downto 0) );
    end component midi_shell;

    -- Constants
    constant clk_period : time := 10 ns; -- 100 MHz master clock
    constant bit_time : time := 32us;
    constant msgData : std_logic_vector(7 downto 0) := "10010000"; -- Note on
message
    constant noteData : std_logic_vector(7 downto 0) := "00111100"; -- Arbitrary
    constant velData : std_logic_vector(7 downto 0) := "01101001"; -- Arbitrary

    -- Inputs
    signal mclk : std_logic := '0';
    signal midi_in : std_logic := '1';
```

```
44
45      -- Outputs
46      signal note_on : std_logic := '0';
47      signal seg : std_logic_vector(0 to 6) := (others => '0');
48      signal dp : std_logic := '0';
49      signal an : std_logic_vector(3 downto 0) := (others => '0');
50
51  begin
52
53    -- Instantiate the Unit Under Test (UUT)
54      uut: midi_shell
55          port map(
56             mclk => mclk,
57             midi_in => midi_in,
58             note_on => note_on,
59             seg => seg,
60             dp => dp,
61             an => an
62   );
63
64     -- Clock process definitions
65     clk_process :process
66     begin
67      mclk <= '0';
68      wait for clk_period/2;
69      mclk <= '1';
70      wait for clk_period/2;
71     end process;
72
73     -- Stimulus process
74     stim_proc: process
75     begin
76
77        wait for 100 us;
78        wait for 10.25*clk_period;
79
80        midi_in <= '0';   -- Start bit
81        wait for bit_time;
82
83        -- Shift in message data
84        for bitcount in 0 to 7 loop
85          midi_in <= msgData(bitcount);
86          wait for bit_time;
87        end loop;
88
89        midi_in <= '1';   -- Stop bit
```

```
90
91        wait for 200 us;
92
93        midi_in <= '0';   -- Start bit
94        wait for bit_time;
95
96        -- Shift in note data
97        for bitcount in 0 to 7 loop
98          midi_in <= noteData(bitcount);
99          wait for bit_time;
100       end loop;
101
102       midi_in <= '1';   -- Stop bit
103
104       wait for 100 us;
105
106       midi_in <= '0';   -- Start bit
107       wait for bit_time;
108
109       -- Shift in velocity data
110       for bitcount in 0 to 7 loop
111         midi_in <= velData(bitcount);
112         wait for bit_time;
113       end loop;
114
115       midi_in <= '1';   -- Stop bit
116       wait for 200 us;
117
118       wait;
119    end process;
120 end testbench;
121
```

Appendix H: Resource Utilization

The table below outlines the resource utilization of our design on the Basys 3 FPGA.

Resource	Utilization	Available	Utilization %
LUT	1134	20800	5.4519234
LUTRAM	5	9600	0.052083336
FF	801	41600	1.9254807
BRAM	0.5	50	1.0
IO	17	106	16.037735
BUFG	2	32	6.25

Appendix I: Memory Map

The only block memory utilized in this design was BROM for the sine wave LUT. The LUT was generating using Vivado's built-in DDS compiler. The dimensions of the LUT are *4096 locations x 12 bits per location*. Therefore, the memory address is a 12-bit unsigned.

Appendix J: Analysis of Residual Warnings

Listed below are the residual warnings from our final design. The synthesis warnings all prove benign. Vel_reg is unused because our system does not process the note velocity data it stores in the MIDI interface in any way. However, it is still necessary to receive that data in order to maintain a 3-byte MIDI input cycle, so we keep the unused register. Sin_sum_reg is reduced in size because after division by keys_pressed, the sine_sum_reg is never more than 12 bits wide. The register is 18-bits to accommodate the possibility of dividing by the square root of keys_pressed, which would yield results more than 12 bits wide. The no constraints selected for write error is erroneous - we verified that our inputs and outputs mapped correctly by scoping the utilized ports on the Basys 3.

The rest of the errors come from the Out-of-Context Module Run for creating the sine_wave LUT. We removed unnecessary ports from our component declaration and port map for the sine_wave BROM in the shell. The synthesis tool for the BROM warns of the unconnected/missing ports. Unnecessary ports were left unconnected intentionally, so these warnings are benign. Scope readings demonstrate that the sine_wave LUT functions properly. See Appendix D, Figure 7 for a waveform of the properly output sine wave.

Synthesis

- [Synth 8-6014] Unused sequential element vel_reg_reg was removed. ["P:/19summer/engs031/PG17/final_project/final_project.srcs/sources_1/new/midi_interface.vhd":68]
- [Synth 8-3936] Found unconnected internal register 'sine_sum_reg_reg' and it is trimmed from '18' to '12' bits. ["P:/19summer/engs031/PG17/digital_synth/digital_synth.srcs/sources_1/new/sound_engine.vhd":282]
- [Constraints 18-5210] No constraints selected for write. Resolution: This message can indicate that there are no constraints for the design, or it can indicate that the used_in flags are set such that the constraints are ignored. This later case is used when running synth_design to not write synthesis constraints to the resulting checkpoint. Instead, project constraints are read when the synthesized design is opened.

Out-of-Context Module Runs

- [Synth 8-5640] Port 'debug_axi_pinc_in' is missing in component declaration ["p:/19summer/engs031/PG17/digital_synth/digital_synth.srcs/sources_1/ip/sine_wave/synth/sine_wave.vhd":72]

- [Synth 8-5640] Port 'debug_axi_poff_in' is missing in component declaration ["p:/19summer/engs031/PG17/digital_synth/digital_synth.srcs/sources_1/ip/sine_wave/synth/sine_wave.vhd":72]
- [Synth 8-5640] Port 'debug_axi_resync_in' is missing in component declaration ["p:/19summer/engs031/PG17/digital_synth/digital_synth.srcs/sources_1/ip/sine_wave/synth/sine_wave.vhd":72]
- [Synth 8-5640] Port 'debug_axi_chan_in' is missing in component declaration ["p:/19summer/engs031/PG17/digital_synth/digital_synth.srcs/sources_1/ip/sine_wave/synth/sine_wave.vhd":72]
- [Synth 8-5640] Port 'debug_core_nd' is missing in component declaration ["p:/19summer/engs031/PG17/digital_synth/digital_synth.srcs/sources_1/ip/sine_wave/synth/sine_wave.vhd":72]
- [Synth 8-5640] Port 'debug_phase' is missing in component declaration ["p:/19summer/engs031/PG17/digital_synth/digital_synth.srcs/sources_1/ip/sine_wave/synth/sine_wave.vhd":72]
- [Synth 8-5640] Port 'debug_phase_nd' is missing in component declaration ["p:/19summer/engs031/PG17/digital_synth/digital_synth.srcs/sources_1/ip/sine_wave/synth/sine_wave.vhd":72]
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized0 has unconnected port CE
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized0 has unconnected port SSET
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized0 has unconnected port SINIT
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv has unconnected port SSET
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv has unconnected port SINIT
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized12 has unconnected port CLK
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized12 has unconnected port CE
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized12 has unconnected port SCLR
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized12 has unconnected port SSET
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized12 has unconnected port SINIT
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized8 has unconnected port SCLR
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized8 has unconnected port SSET
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized8 has unconnected port SINIT
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized4 has unconnected port SCLR
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized4 has unconnected port SSET
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized4 has unconnected port SINIT
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized6 has unconnected port SCLR
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized6 has unconnected port SSET
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized6 has unconnected port SINIT
- [Synth 8-3331] design sin_cos has unconnected port SCLR
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized10 has unconnected port CLK
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized10 has unconnected port CE
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized10 has unconnected port SCLR
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized10 has unconnected port SSET
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized10 has unconnected port SINIT
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized2 has unconnected port SCLR
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized2 has unconnected port SSET
- [Synth 8-3331] design xbip_pipe_v3_0_5_viv__parameterized2 has unconnected port SINIT
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port RFD
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port COSINE[11]

- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port COSINE[10]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port COSINE[9]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port COSINE[8]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port COSINE[7]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port COSINE[6]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port COSINE[5]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port COSINE[4]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port COSINE[3]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port COSINE[2]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port COSINE[1]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port COSINE[0]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port ADDR[0]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port REG_SELECT
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port SCLR
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port WE
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port DATA[11]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port DATA[10]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port DATA[9]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port DATA[8]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port DATA[7]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port DATA[6]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port DATA[5]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port DATA[4]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port DATA[3]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port DATA[2]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port DATA[1]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port DATA[0]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port PINC_IN[11]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port PINC_IN[10]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port PINC_IN[9]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port PINC_IN[8]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port PINC_IN[7]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port PINC_IN[6]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port PINC_IN[5]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port PINC_IN[4]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port PINC_IN[3]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port PINC_IN[2]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port PINC_IN[1]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port PINC_IN[0]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port POFF_IN[11]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port POFF_IN[10]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port POFF_IN[9]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port POFF_IN[8]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port POFF_IN[7]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port POFF_IN[6]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port POFF_IN[5]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port POFF_IN[4]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port POFF_IN[3]

- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port POFF_IN[2]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port POFF_IN[1]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port POFF_IN[0]
- [Synth 8-3331] design dds_compiler_v6_0_17_core has unconnected port RESYNC_IN
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port m_axis_data_tlast
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port m_axis_data_tuser[0]
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port m_axis_phase_tdata[0]
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port m_axis_phase_tlast
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port m_axis_phase_tuser[0]
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port aclken
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port aresetn
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port s_axis_phase_tdata[15]
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port s_axis_phase_tdata[14]
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port s_axis_phase_tdata[13]
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port s_axis_phase_tdata[12]
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port s_axis_phase_tlast
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port s_axis_phase_tuser[0]
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port s_axis_config_tvalid
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port s_axis_config_tdata[0]
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port s_axis_config_tlast
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port m_axis_data_tready
- [Synth 8-3331] design dds_compiler_v6_0_17_viv has unconnected port m_axis_phase_tready
- [Synth 8-6040] Register i_rtl.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.
- [Constraints 18-5210] No constraints selected for write. Resolution: This message can indicate that there are no constraints for the design, or it can indicate that the used_in flags are set such that the constraints are ignored. This later case is used when running synth_design to not write synthesis constraints to the resulting checkpoint. Instead, project constraints are read when the synthesized design is opened.
- [Constraints 18-5210] No constraints selected for write. Resolution: This message can indicate that there are no constraints for the design, or it can indicate that the used_in flags are set such that the constraints are ignored. This later case is used when running synth_design to not write synthesis constraints to the resulting checkpoint. Instead, project constraints are read when the synthesized design is opened.

```
% Grant Larson and Catherine Slaughter
% Script for generating frequency to M value LUT
% 8/14/19

% Declare constants
N = 4096; % Number of phase values
Fref = 130.81; % Frequency of first note on keyboard
Fs = 50E3; % 50 kHz sampling rate

% Declare column vector of note numbers from 0-127
note_num = (48:72)';

% Calculate frequency with reference to note 0
freq = Fref * 2.^((note_num-48)/12);

% Calculate accumulator interval M based on frequency
M = round(1024 * freq * N / Fs); % Multiply by 1024 for precision
```

Published with MATLAB® R2019a