EE108 Lab 4

Note Player

Lab Due: October 21st at 11:59 pm

Objective: The purpose of lab 4 is to build a single-note music synthesizer in Verilog to be extended in lab 5 and the final project.

1. Introduction

So, what are we doing here?

In this lab you will implement a wave-table music synthesizer. The music player will use data from songs stored in a ROM consisting of notes and durations, look up the frequency for the note in another ROM, and then play that note through a speaker using a sine ROM (i.e. a wave table) to synthesize a sine wave of the appropriate frequency. The sine wave output will give you pure tones. In the final project you can add harmonics and dynamics to create more realistic voices.

You will be reusing the modules you develop for lab 4 in both lab 5 and your final project, so it is very important that you stick to the interfaces we give you. If you think you need a different interface, chances are you're not implementing the proper functionality, in which case you should talk to us.

This is a *much* larger lab than any of the labs you've done already, so we're providing you with a bunch of starter files. Make sure you look through them thoroughly before you get started. Since the lab is very large, you'll probably want to work in parallel with your partners most of the time. The interfaces between all major modules are well-defined, which makes working in parallel easier, but you still need to do a very thorough job of sketching how your modules will work to avoid misunderstandings between your and your partners' code.

Instructions on what exactly you have to do can be found on page under the "*Deliverables*" header. The remainder of this introduction reviews the architecture of the starter code and the interfaces you must adhere to.

Wait, before we go on, what are ROMs again?

ROMs, short for **read only memories**, are special modules designed to hold a large set of data to be looked up. The ROM organizes its data by n-bit **words** which each have a unique **address**. The ROMs we use in this lab all have one input port and one output port. We feed the address of the word we want into the input port of the ROM and **automatically on the next cycle** that word will appear at the output. As long as we hold the input address constant, the output word will be constant.

In this lab we use a ROM to store the pre-calculated values of a sine wave changing over time, so we don't have to calculate them ourselves. We also use a different ROM to store the series of notes that make up our songs, and yet another ROM to store what audio frequency each note corresponds to.

ROMs are close cousins of RAMs, **random access memories**, which are similar but allow us to write values to addresses at run-time as well as read them.

A note on musical notation

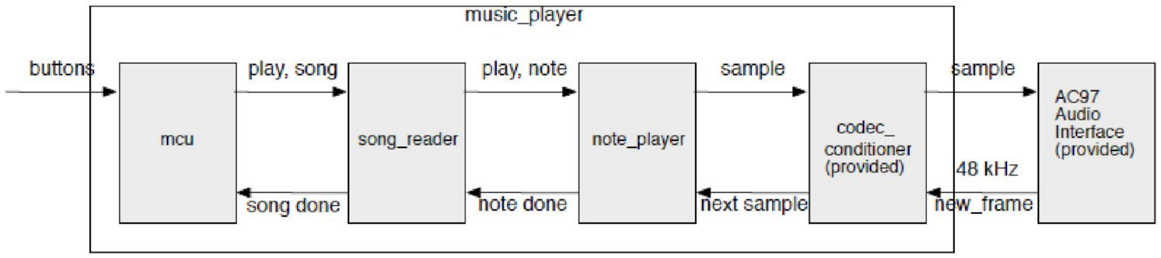
You do not need any prior knowledge of music or musical notation to do this lab, but regardless you may encounter some confusing terminology. Notes are just names that correspond to various pitches of a sound. The name of the note starts with a number from 1 to 6 and ends with a letter from A to G, with pitch increasing as the numbers get higher and letters get further along in the alphabet. Additionally, some notes end with a '#' or a '*b*' symbol, which are pronounced *sharp* and *flat*, respectively. Sharp and flat notes are duplicates of each other. That is 3C# is the same pitch as 3D*b*. They're just synonyms for the same sound. This is all just terminology and you don't have to have any understanding of why notes are organized this way. This is a list of notes our music player will play, in order of increasing pitch:

1A	2A	3A	4A	5A	6A
1 A#/Bb	2A#/Bb	3 A#/Bb	4A#/Bb	5 A#/Bb	6A#/Bb
1 B	2B	3 B	4B	5 B	6B
1 C	2C	3 C	4C	5 C	
1 C#/Db	2C#/Db	3 C#/Db	4C#/Db	5 C#/Db	
1 D	2D	3 D	4D	5 D	
1 D#/Eb	2D#/Eb	3 D#/Eb	4D#/Eb	5 D#/Eb	
1 E	2E	3 E	4E	5 E	
1 F	2F	3 F	4F	5 F	
1 F#/Gb	2F#/Gb	3 F#/Gb	4F#/Gb	5 F#/Gb	
1 G	2G	3 G	4G	5 G	
1G#/Ab	2G#/Ab	3G#/Ab	4G#/Ab	5G#/Ab	

2. Architectural overview

The music_player module is the top-level module[1] of your design and contains 4 sub-modules, the master control unit (mcu), the song_reader, the note_player, and the codec_conditioner. It is important to understand the flow of events and data in this system so you will understand the role of each module.

There are two types of events that drive your system: button presses (play or next_song) and new_frame signals from the audio chip on the board. In response to these three events your music_player will synthesize the correct audio samples at the correct time.



Let's run through a sample set of inputs and responses from music_player to figure out the behavior of the system. Make sure you see how the steps correspond with the arrows in the diagram.

First, examine the diagram from left to right to see how samples, songs and notes are selected:

1. The user pushes the play button, which is observed by the mcu.
2. The mcu tells the song_reader to play one of the four available songs.
3. The song reader looks up the first note of the selected song from an internal song_rom module, and passes the first note along with a play signal to the note_player.
4. The note_player sees the play signal and the note, and looks up the frequency associated with that note from an internal frequency_rom module. It passes the frequency into an internal sine_reader module that calculates the actual audio sample for a sine wave of that frequency. It then waits for codec_conditioner to request the next sample, and then passes the next sample to the codec_conditioner.

So, now we've output a single sample of a single note from a single song. When do the samples, notes, and songs change? To understand this, follow the diagram from the right to left:

1. A 48 kHz new_frame input from the audio codec on the FPGA goes high for a cycle.
2. The codec_conditioner sees the new_frame input and asks note_player to pass it the next 48 kHz sample of the current note.
3. When the note_player finishes the current note (which happens when the

[1] When you synthesize your design for the FPGA you will use the lab4_top.v file which contains all the inputs and outputs for the actual FPGA.

codec_conditioner has asked it for enough samples) it tells the song_reader that it is done with the note.

4. The song_reader then tells the note_player what the next note is.
5. When the song_reader is done with a song (which happens when the note_player has asked for all of the notes in the song) it tells the mcu that the song is done.
6. The mcu is then responsible for telling the song_reader to go on to the next song when the appropriate button is pressed.

This event flow is critical for understanding how this system works together. Remember that the audio codec requests new samples and the buttons control the state of the system. Note how nicely this breaks down your design. Once we have defined the interface between each of these blocks you can write and test them independently before you hook them up.

The Master Control Unit (MCU)

The MCU has two tasks: controlling the state of the overall system in response to the button presses and keeping track of which song is currently being played.

MCU Interface:

Signal	Direction	Description
clk	Input	Clock signal
reset	Input	Reset signal
play_button	Input	A one-cycle pulse indicating the play button has been pressed
next_button	Input	A one-cycle pulse indicating the next_button has been pressed
play	Output	True if the system should be playing, false if no audio output should be generated
reset_player	Output	High when the player is moving on to the next song. Resets the other parts of the system so they aren't in the middle of their jobs.
song_done	Input	From the song_reader indicating that the current song has finished.
song[1:0]	Output	The song to play.

The MCU is responsible for starting off paused when the system is reset. Whenever the system is paused it should output no audio (just hold the value of the current sample steady—since this will be a DC signal, you'll hear only silence). When the play button is pressed, the MCU will signal to begin playing the current song. If the play button is pressed while playing the song, the song will pause, and it will resume at the same point in the song when play is pressed again (this is what the play output to the song_reader is for). When the song finishes, the MCU should wait in the paused state at the beginning of the next song. If the next button is pressed at any time, the MCU will instruct the song_reader to go to the beginning of the next song and pause there. After song 3 the MCU should return to song 0. This is trivial, just let your song counter wrap around.

The Song Reader

The song_reader is responsible for reading the song's sequence of notes out of the provided song_rom. The song_rom has 128 entries of 12 bits each. Each address contains a note, in the format of {6'note, 6'duration}, where the note represents a note frequency in the frequency_rom in the note_player, and the duration is the length of the note in 48ths of a second. The song_rom contains 4 songs, each with 32 notes. A song that is shorter than 32 notes should be filled with 0-length notes at the end. Take a look at the song_rom file for the details. The rom is generated using a spreadsheet, which is also included (rom_generator.xls).

The song_reader takes in the number of the song to play from the MCU and looks up the first note in the song_rom. (Note that the song_rom takes one cycle to return the data, similar to a flipflop!) It then sends the new note and the duration on to the note_player. The song_reader then waits for the note_player to tell it that it has finished the note. When the note_player finishes playing the current note, the song_reader looks up the next note and sends it to the note_player. This repeats until the song_reader has finished the current song, at which point it tells the MCU that the song is done.

Song Reader Interface:

Signal	Direction	Description
clk	Input	Clock signal
reset	Input	Reset signal
play	Input	True if the song reader should be playing.
song[1:0]	Input	The song to play.
note_done	Input	From the note_player to indicate that the note has finished and that it is ready for the next note.
song_done	Output	True if the song has finished.
note[5:0]	Output	The note from the song_rom to play now. The frequency_rom looks up this note to find the step size to use to generate the sine wave output.
duration[5:0]	Output	The duration for the note from the song_rom to play now.
new_note	Output	One cycle pulse that tells the note_player to latch in the values on note and duration and start playing that note.

When you implement your song_reader it is essential that you draw a timing diagram that shows what states your FSM is going through and when the data is ready from the song_rom. Otherwise, you will end up sending the wrong note to the note_player (see page for instructions on drawing these diagrams). You should instantiate the song_rom within the song_reader since this is the only place it will be used.

Here are some entries from the song_rom we're providing:

Address (7 bits)	**Value (12 bits)**	**Note Value**	**Duration Value**
7'd90 = 7'b **10** *11010* song **2**, note *26*	{6'd43, 6'd6}	Note 43 = D♯ or E♭ 4	6/48ths
7'd91 = 7'b **10** *11011* song **2**, note *27*	{6'd44, 6'd14}	Note 44 = E 4	14/48ths
7'd92 = 7'b **10** *11100* song **2**, note *28*	{6'd0, 6'd28}	Note 0 = rest (silence)	28/48ths

Note how in the address column, the two MSBs map to the song we're playing, and the bottom 5 bits map to the current note we're playing. You should convince yourself that addressing the song_rom in this manner really splits the 128-entry song_rom into a 4 song by 32 note matrix of songs.

In this example, when the second song is playing and it gets to the 26th note, it will tell the note_player to play the 4th D sharp or E flat (they're the same note) for 6/48ths of a second. The following note (song 2, note 27) will be the 4th E for 14/48ths of a second, followed by silence (a rest, denoted by note value = 0) for 28/48ths of a second.

Both the song_rom and the frequency_rom are generated using the provided spreadsheet (rom_generator.xls), so you can easily modify them. (Indeed you need to create a fourth song to complete the lab, but it doesn't have to be any good—if you're lazy or don't have much time, just make it a scale.)

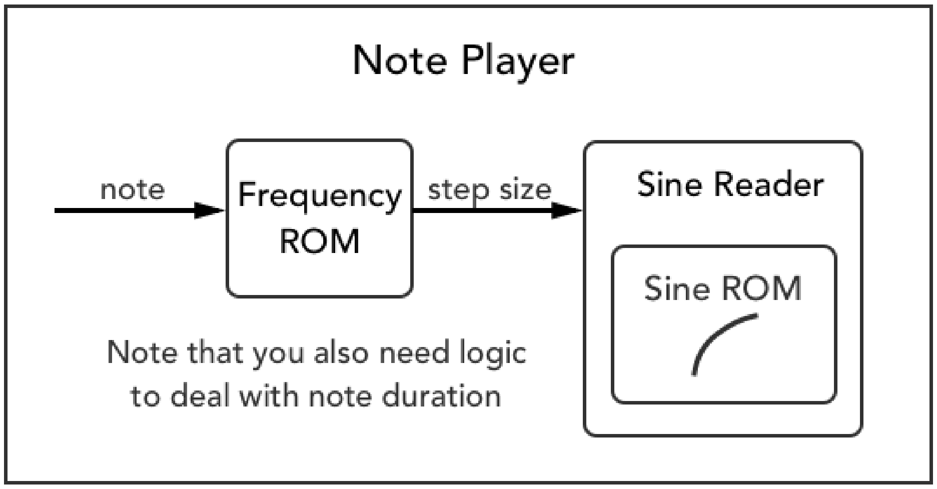
If you have a state in your song reader that you go through for one cycle whenever you change notes, you can add a $display statement to it which could output something like:

```
$display("Playing note %d of song %d, which is note %d duration %d",
          note_address, song, note, duration);
```

This way, when debugging your song_reader, the simulator will tell you what note it was playing. $display statements are ignored by the Xilinx synthesizer, so you don't have to worry about the fact that the $display statement is not synthesizable.

Note Player

Here is a sketch of some of the internal components and connections you'll need in your note player:



The note_player is the central part of this lab. Its responsibility is to take in a note, look up the step size required to generate the correct frequency for that note, and then synthesize a sine wave at that frequency for as long as the note is playing.

The note player must store a note and its duration from the song_reader, output a sine wave of the correct frequency for the provided duration, and then tell song_reader it has completed the note. The sine wave is synthesized by looking up the note's step size in a frequency rom, and feeding this value to the sine reader.

The note_player also needs a counter to keep track of the duration of the current note. This counter ticks every 48th of a second while the note is playing. If we pause the song, the counter pauses as well. The 48th beats are generated by a beat_generator module which we provide for you.

Note Player Interface:

Signal	Direction	Description
clk	Input	Clock signal
reset	Input	Reset signal
play_enable	Input	True if the note should be playing
note_to_load[5:0]	Input	The note to load
duration_to_load[5:0]	Input	The duration to load
load_new_note	Input	Goes high when we have a new note to load
done_with_note	Output	Goes high when we have finished playing our note
beat	Input	Goes high for one cycle at 48Hz
generate_next_sample	Input	From the codec_conditioner telling us to generate and output the next sample
sample_out[15:0]	Output	The 16-bit audio sample output
new_sample_ready	Output	Tells the codec_conditioner that we have a new sample ready for it.

Sine Reader

The sine_reader is a sub-module of the note_player which takes in a step size (determined by the frequency of the note) and steps through the sine ROM, generating a sine wave with frequency determined by the step size.

Our sine ROM takes in an address and produces a sample from the sine wave. If we address the sine ROM correctly, we can generate sine waves of arbitrary frequency. For every generate_next pulse that we get, we have to increment the address pointer into our sine ROM by a certain step size; this represents the frequency. If we increment our address pointer by a larger step size, we walk through the sine wave faster, and we generate a higher frequency sine wave. If we increment our address pointer by a smaller step size, we walk through the sine wave slower, and generate a lower frequency sine wave. The module output is generated from the sine ROM.

Sine Reader Interface:

Signal	**Direction**	**Description**
clk	Input	Clock signal
reset	Input	Reset signal
step_size[19:0]	Input	The step by which we count each time. This is the frequency.
generate_next	Input	True when we should generate the next sample.
sample_ready	Output	True if we have a sample ready to output. (Remember the sine_rom takes a cycle!)
sample[15:0]	Output	The sample we've generated.

This sounds relatively simple, but there are three complications with the sine reader. First, the sine_rom takes one cycle to output its value, so you need to make sure your logic is designed around this timing. Draw a timing diagram to figure this out before you start coding.

Secondly, the sine_rom only has 1024 samples, which means it has a 10 bit address signal. However, we need more than 10 bits of precision to make precise tones. For example, we might want to increment the address of the sine ROM by 10.5 for every 48 kHz pulse, rather than by just 10 (if we truncate the bits). So after two pulses, we want to increment the address by 21 (10.5 * 2), rather than by 20 (10 * 2). To do this, we must keep extra bits of precision around. We will use a 10.10 binary fixed point signal for our step size. We will keep our address counter 20 bits wide, but we'll only use the upper 10 bits to actually address the sine ROM.

Let's run through the above example of using a step size of 10.5 (0000001010.1000000000). Our first addition will give us address 10:

current address	+	step size	=	next address
0000000000.0000000000	+	0000001010.1000000000	=	0000001010.1000000000

We only use the top 10 bits (0000001010 = 10'd10) as the address to the sine_rom.

Our second addition will give us 21:

current address	+	step size	=	next address
0000001010.1000000000	+	0000001010.1000000000	=	0000010101.0000000000

Again, we only use the top 10 bits as the address to the sine_rom, which are 0000010101 = 10'd21.

Since we can interpret bits however we want, there's nothing special about treating the 20-bit step size input as a 10.10 number, as long as the input was calculated as a 10.10 number. The frequency_rom that provides the step_size is calculated such that the output is a 10.10 step size, so this works for you out of the box.

So in the end, we basically end up using a 20 bit counter and take the top 10 bits off as the address—except that there's one more complication.

The sine ROM does not store an entire sine wave. Instead, it stores one quarter of a sine wave. From that quarter, you have to generate the other three quadrants.

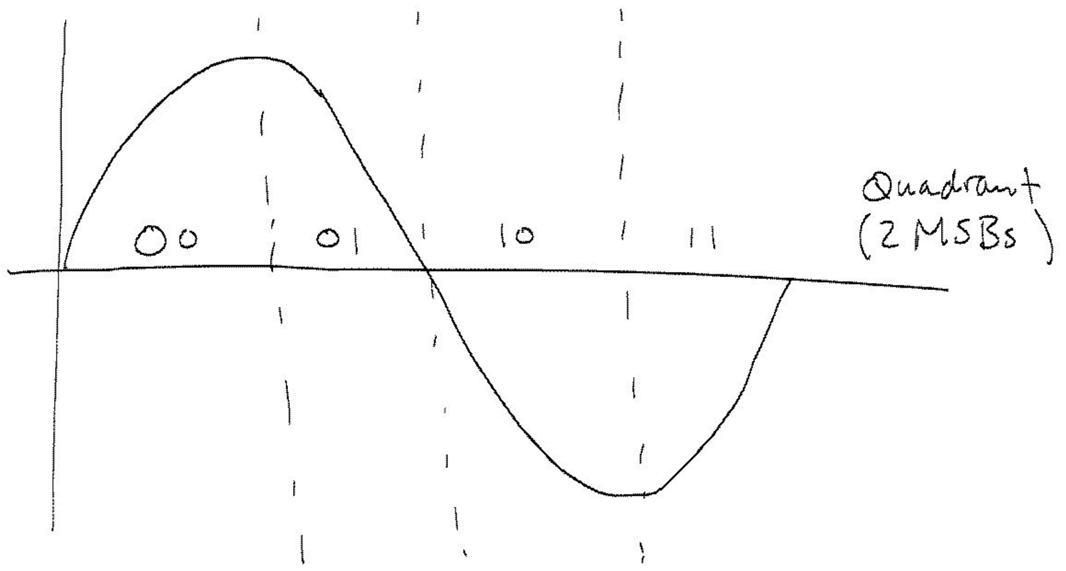
We can deal with this the same way we've dealt with the step size—by using extra bits of precision. We add two new most significant bits to our 20 bit counter to make it 22 bits wide. We use the top two bits to denote which quadrant of the sine wave we're in:

```
Q  Raw addr    Precision
01 0000001010  1000000000
```

Q – 2 bits that specify the quadrant of the sine wave that we're in
Raw address – 10 bits that specify the address we need
Precision – 10 bits that we keep around just for precision.

Therefore we need a 22 bit flip flop to store the address bits for our counter, and 22 bit next address and 22 bit current address signals going into the D and Q ports of the DFF, respectively.

Depending on the quadrant of the sine wave we're in, we change either the address into the sine ROM or the result out of the sine ROM accordingly. If we are in the first quadrant of the sine wave, we can feed raw address into the sine ROM and leave the output of the quarter sine ROM unmodified. For the other quadrants, we see that the data coming out is either flipped horizontally or vertically, or both. So we need to modify the address going into the sine ROM (to flip the sine wave horizontally), or the data coming out of the sine ROM (if we want to flip the sine wave vertically). Here is a sketch and a chart to get you started.



Quadrant	Address into Sine ROM	Output of Module
00	raw_address	sine_rom_output
01	<fill out>	<fill out>
10	<fill out>	<fill out>
11	<fill out>	<fill out>

If you need to negate a signal remember 0 minus a number gives you the additive inverse of that value. You should not use multipliers anywhere in this lab. (You may, however, use them on the final project as necessary.)

The sine_reader is the most complicated module of this project, but it's also the most fun to debug. You'll want to check the appendix under "Simulation Notes" for more info on how to display the sine wave as an analog waveform, which will be immensely useful in debugging.

Codec Conditioner and adau1761

The codec conditioner is provided for you and interfaces your design with the audio hardware on the development board, controlled by the (also provided) adau1761 module. The codec conditioner will automatically output to you a 'generate_new_sample' signal which will tell your note player to grab the next value out of its sine_reader. You'll then raise its input 'latch_new_sample_in' once you have that value and have fed it to the 'new_sample_in' port.

The purpose of this module is to hide all of the nasty timing issues from us that pop up when our design interfaces with logic that does not run at the same clock frequency. You are generating audio at 48kHz. The clock on the FPGA runs at 100MHz. That means you have 2083 cycles between each new_frame. With the codec_conditioner you have 2083 cycles to generate the next sample before anything bad happens. This should be plenty of time.

We count down note durations using beat_generator, which is driven by the audio chip's clock much as your bike light was driven by beat32 in lab 3. This is convenient because it means that if the audio chip's clock drifts a little bit we'll still be in sync with it, and instead of having to divide the FPGA clock's 100MHz down to 48Hz we only have to divide 48kHz down to 48Hz.

For more information about how this works under the surface, ask your TAs.

Codec Conditioner Interface:

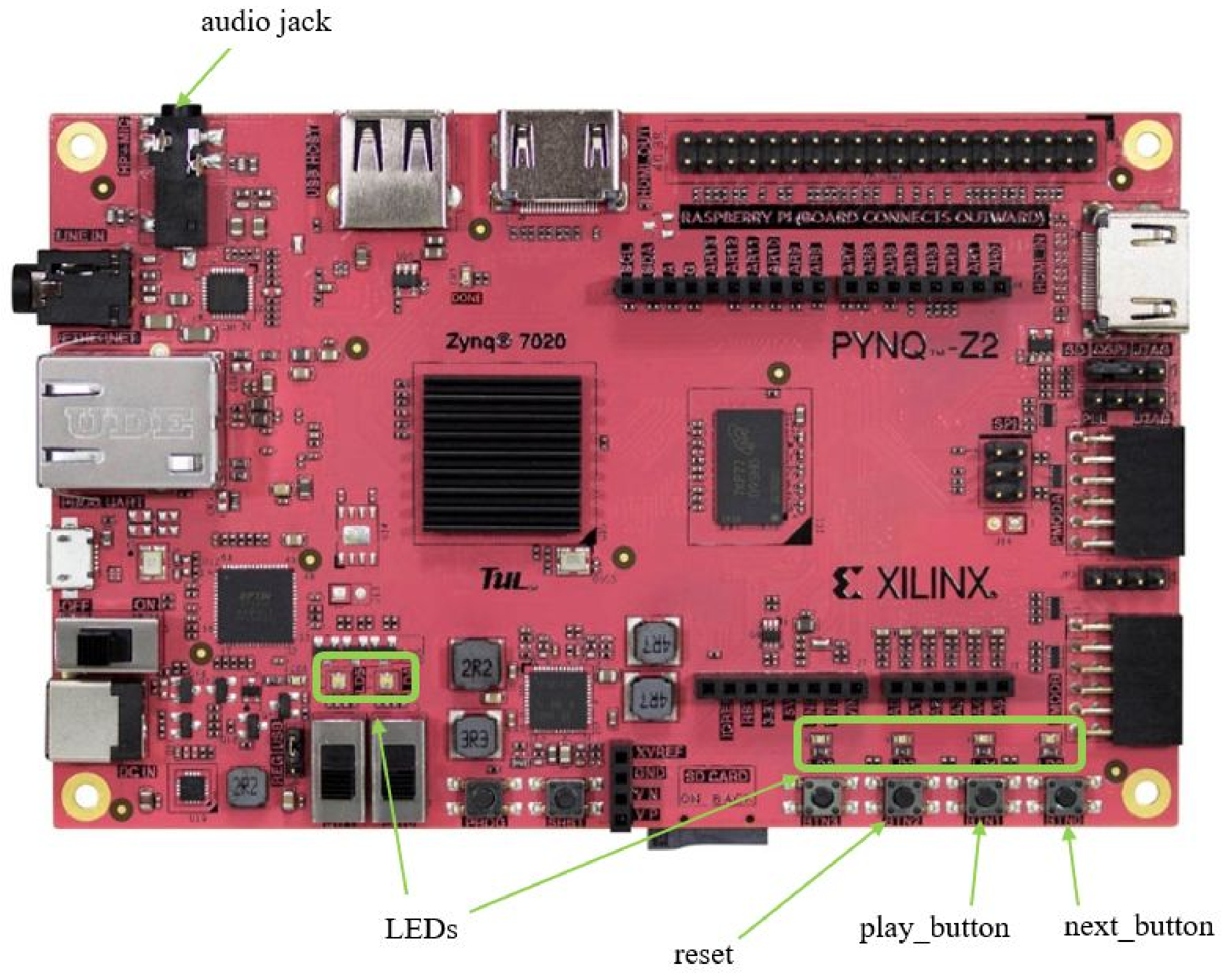
Signal	**Direction**	**Description**
Clk	Input	Clock signal
Reset	Input	Reset signal
new_sample_in[15:0]	Input	The sample to send to the codec on the next new_frame
latch_new_sample_in	Input	True when we should latch the data on the new_sample_in input.
valid_sample[15:0]	Output	The always-valid sample we present to the ac97 codec.
new_frame	Input	Input from the codec that tells us to output the next value.
generate_next_sample	Output	Tells the note_player that it's time to generate the next sample. This, combined with the latch_new_sample_in, constitutes a handshake between the codec_conditioner and the note_player.

Board Interface:

Note that while there are 2 audio jacks, the one we want is the top one (labeled). If you plug into the one on the left, you won't hear anything.

Be very careful with the audio level! Since we haven't programmed the FPGA to respond to volume control, it will always be at max volume. If you're using headphones, the volume may be extremely loud, so you should start with the headphones outside of your ears.

There are 6 LEDs (labeled) on the board that correspond to various bits of the sample_out. When everything is working properly, you should see the LEDs change with the music. If you don't hear anything and you don't see the LEDs flash, then your sample likely isn't being generated, which may indicate an issue with the communication between song_reader and note_player. If you don't hear anything but your LEDs are flashing, then samples are being generated, but they aren't the correct samples.



Setup:

As usual, the starter files are on Canvas. You should do the usual Vivado setup:

- Add design sources, simulation sources, and constraint sources.
- Set up the github repository.
- Adding/customizing IP. Just like in lab 3, you only need to use the clock wizard IP for this lab. Just set clk_out1 to 100MHz.

3. Deliverables

Read this whole document. At least twice. Understand all the signals for each module. In particular, how they flow and what causes them to change. Please don't read the list below until you know what's going on.

Zip up the following files and submit it on Gradescope

1. Block diagrams for note_player and sine_reader.
2. FSM state diagrams for song_reader and the MCU.
3. Timing diagrams for the song_reader, note_player, and sine_reader showing the delay in reading from the ROMs.
4. The final verilog files for your mcu, song_reader, note_player, and sine_reader modules.
5. Test benches for all of the above and their annotated output. Remember to write thorough test benches and deliver good annotations. You can refer to the "Debugging, Testbenches, and Annotation Example.pdf" file on canvas to have some guidance.
6. The timing and synthesis reports generated after creating the bitfile.
7. The **critical path** of your design. You can find it by going to "Design Runs" then double clicking on the Worst Negative Slack (WNS). Then, click on the Worst Negative Slack value under "Setup." This report lists the longest paths in your design, with the one with the least positive/most negative being the critical path. **Report the slack, source, and destination of the critical path**. Look at the list of locations the signal visits under 'Maximum Data Path'. Do you recognize from where in your logic this path comes from?
8. Write a song of your choice for song 3 in the song ROM. There is a provided spreadsheet (rom_generator.xls) which makes writing songs trivial: you can literally just type in A B C D E F G to get a scale, for example. Provide the final song rom module you use.
9. A video demo to demonstrate that your design works on the board. All four songs should be played in your demo.

Feel free to draw diagrams by hand and scan or photograph them to turn them in. Just make sure they are legible.

FAQ

Q: Should "note_done", "load_new_note", "generate_next_sample", and "new_sample_ready" be one pulses or set high?
A: One pulses

Q: I see a timing violation in the codec (with a source or destination of "adau1761_codec/i2c_interface/Inst_i2s_data_interface/..."); is this something I need to fix?
A: Nope, this is a bug in the starter code we've sourced for the class but shouldn't affect your final product.

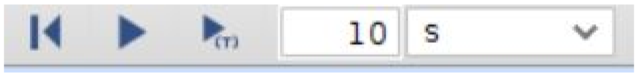
Q: I see a timing violation that's not in the codec, is this something I need to fix?
A: Yes, this is something you need to fix. In the past, some students have written code that passes all of their testbenches, but doesn't output any sound. In some cases, this was due to a

timing violation in the code they wrote. If the source or destination of your timing violation is not adau1761_codec/… as noted above, you have a timing error in your code.

Simulation Notes

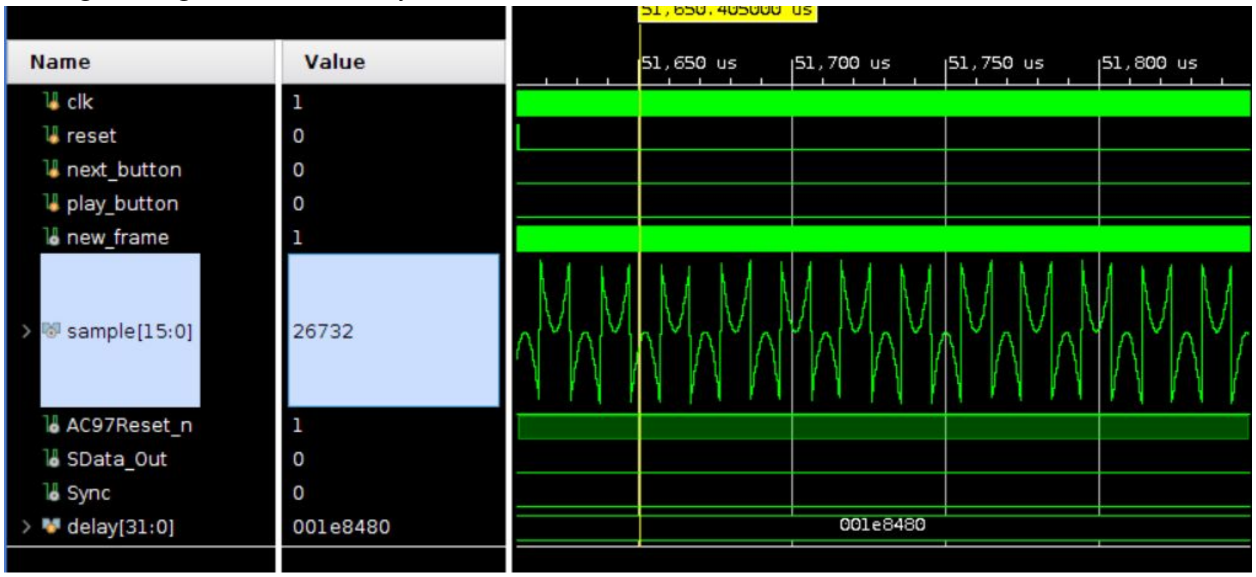
Below is an example of the output from the provided top-level testbench (music_player_tb.v). Note that you can see the different frequencies and lengths of the notes playing, as well as the states of the various modules. Your results may vary depending on what is in your song_rom, and you may want to change the song_rom to make it easier to debug.

By default, Vivado may only run the testbench for a short amount of time. You can run the testbench for additional time at the top by specifying a time and then clicking "Run for ____." If you add additional signals to your testbench, make sure to rewind/restart before clicking "Run for _____."



The third button will extend the testbench for 10 additional seconds.

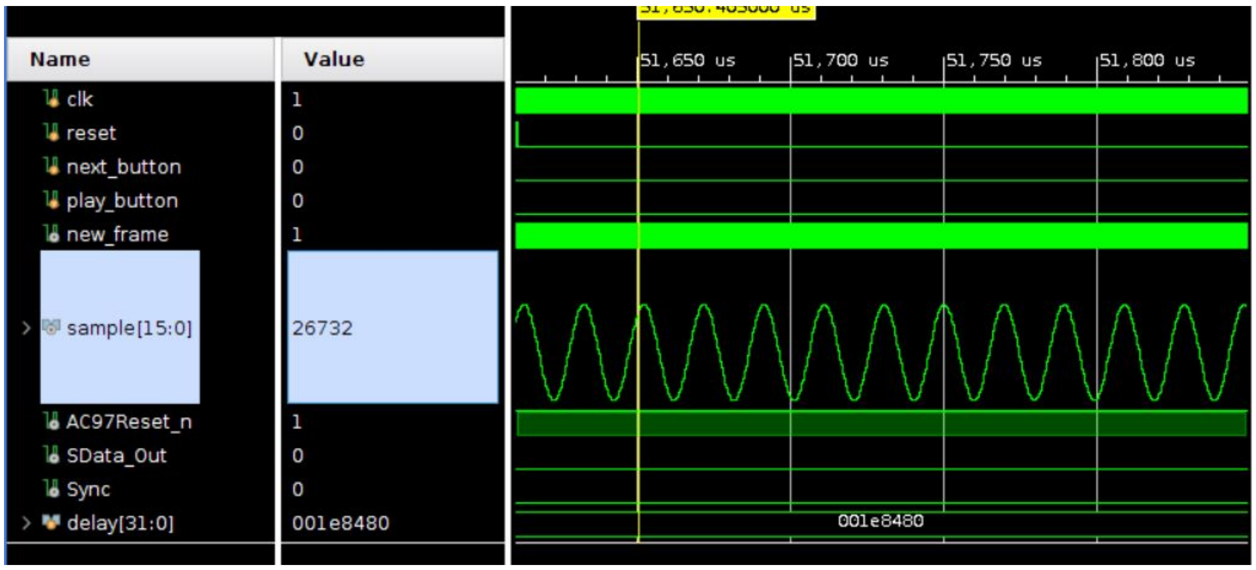
Displaying signals as analog waveforms will be extremely helpful when debugging your sine_reader. When you launch a testbench, right click the wave signal and choose Waveform Style -> Analog. You can also add "height" by right clicking and choosing "Waveform Style -> Analog Settings". This will let you view the waveform as below.



Name	Value
clk	1
reset	0
next_button	0
play_button	0
new_frame	1
sample[15:0]	26732
AC97Reset_n	1
SData_Out	0
Sync	0
delay[31:0]	001e8480

Not quite

You may notice that the waveform looks piecewise sinusoidal. This is because our waveform is, signed, but Vivado displays it as unsigned by default. To fix this, right click the waveform and select Radix -> Signed Decimal. This results in what we want!

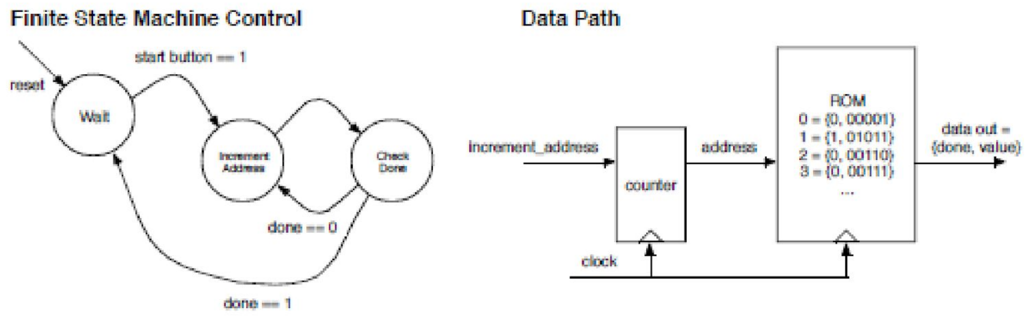


Much better!

4. Timing Diagrams

This lab has mentioned several times that you need to write timing diagrams for various modules. The point of the timing diagrams is to make sure that your FSM and your other modules are going to work in sync. Here's a simple example to get you started.

We're going to control the data path shown below with the FSM shown below. The idea is that we read sequential values out of the ROM by incrementing the address counter until the first bit from the ROM's output (done) is a 1. Then we keep that value and go to the Wait state. This is pretty close to the FSM in the song_reader, and is actually part of one of the FSMs in the final project.



So at first glance this looks like it will work pretty well. We are in Wait until the button is pressed, then we increment the address, then check if we're done. If we're done we go back to Wait, otherwise we increment the address again and repeat. No problem. Unfortunately this will not work. To see why, we make a timing diagram. We assume the system is reset at the beginning and that the only input is the start button being pressed in cycle zero. Here's the timing diagram for our system:

Cycle	**Start button**	**Increment address**	**Address**	**Data out**	**State**
0	1	0	0	{0, 00001}	Wait
1	0	1 **(A)**	0	{0, 00001}	Increment Address
2	0	0	1 **(B)**	{0, 00001}	Check Done
3	0	0	1	{1, 01011} **(C)**	Increment Address
4	0	1 **(D)**	1	{1, 01011}	Check Done
5	0	0	2 **(E)**	{1, 01011}	Wait
6	0	0	2	{0, 00110} **(F)**	Wait

Notice two things: it takes 1 cycle for the address to change when we assert increment address, and it takes 1 cycle for the ROM data to change after we change its address. What this means is that the done bit we're checking in the Check Done state is not the one from the new address! (In cycle 2 above, we're still seeing the data out from the previous address.) The way to see this is to notice in the timing diagram above that one cycle after we assert increment address **(A)** we see the address increment **(B)** and one cycle after that **(C)** we see the new data from the ROM. The same thing happens later with **DEF**. To fix this we need to change the FSM so that it waits to check the data out only after it has had time to let the address increment and the new data become available. You will run into exactly this issue anywhere you use a module (such as a ROM, timer, or counter) that updates on the next cycle!