Lab 4: Music Player

Version 1.0 – David Black-Schaffer
Version 1.2 – David Black-Schaffer, 12 Feb 2007

Introduction

In this lab you will implement a music player. The music player will take songs stored in a ROM consisting of notes and durations, look up the frequency for the note in a ROM, and then play them through a speaker using a sine ROM to synthesize a sine wave of the appropriate frequency. The sine wave output will give you pure tones. In the final project you'll 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 important that you stick to the interfaces we give you. (And if you think you need a different interface chances are you're doing something wrong so please talk to the TAs.)

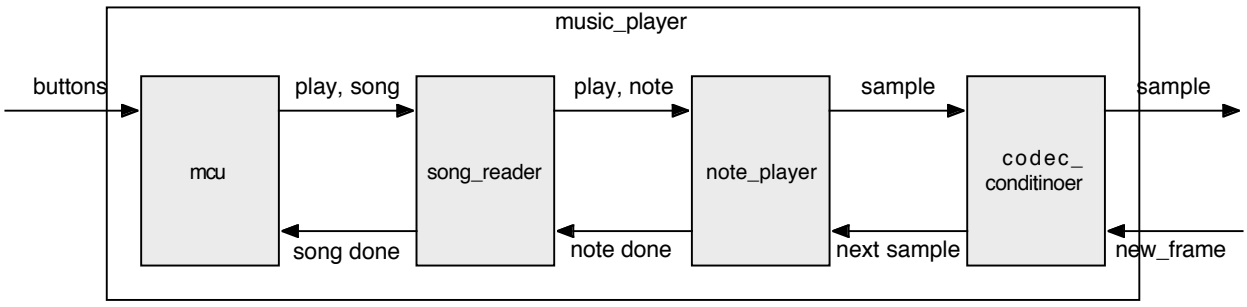
This is a much larger lab than anything you've done so far, so we're providing you with a bunch of helpful starter files, including the whole music_player module.[1] Make sure you look through the provided files before you get started.

Overview

The music_player module is the top-level[2] 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 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 inputs your music_player will output the right audio samples at the right time.

To understand how this works, look at the diagram below:



The play button tells the mcu to start playing. The mcu then tells the song_reader to play the next song. The song_reader then gets the first note of the song from the song ROM and tells the note_player to begin playing it. The note_player looks up the frequency for the note in the frequency ROM and then calculates first audio sample for a sine wave of that frequency, and waits for the codec_conditioner to request the next sample. At the other side, the new_frame input from the audio chip on the board tells the codec_conditioner to tell the note_player to generate the next sample. The note_player then returns the sample to the codec_conditioner and

[1] Before you get too excited I should mention that all the music_player module does is hook up the sub-modules, so while this will save you a bunch of typing it isn't really the whole assignment.
[2] When you synthesize your design for the FPGA you should use the lab4_top.v file as it contains all the inputs/outputs to the actual FGPA.

figures out the next sample. When the note_player finishes the current note (which happens when the codec_conditioner has asked it for enough samples) it tells the song_reader that it is done with the note. The song_reader then tells the note_player what the next note is. When the song_reader is done with a song (which happens when the note_player has asked for enough notes) it tells the mcu that the song is done. 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 system. Once we have defined the interface between each of these blocks you can write them independently and test them independently before you hook them up.

So let's dive into each one in turn.

The 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
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 output should be generated
Reset_player	Output	Goes high when the player is moving on to the next song to reset the other parts of the system so they aren't in the middle of something else.
Song_done	Input	From the song_reader indicating that the current song has finished.
State[1:0]	Output	This is a debug output. You should send the current state of your FSM to this.
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. (That is the sample you send to the audio codec should not change.) When the play button is pressed it will begin playing the current song. If the play button is pressed while playing the song it will pause, and resume at the same point when play is pressed again. (This is what the play output is for.) When the song finishes it should wait in the paused state at the beginning of the song. If the next button is pressed at any time it will go to the beginning of the next song and pause there. After song 3 the MCU should return to song 0, which is trivial as it just means you let your song counter wrap around.

The Song Reader

The song_reader is responsible for reading the song out of the song_rom. The song_rom has 128 entries of 12 bits each. Each address is a note, in the format of {6'note, 6'duration} where the note represents a note 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 is divided into 4 songs, each of 32 notes. A song that is shorter than 32 notes must be filled with 0-length notes at the end. Take a look at the song_rom file for the details.

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. (Remember that the song_rom takes one cycle to return the data!) 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. At this point it 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.

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 the section at the end about timing diagrams.) You should instantiate the song_rom within the song_reader since it is the only module that will be using the song_rom.

For example, here are some entries from the song_rom we're providing:

Address	**Value (12 bits)**	**Note Value**	**Duration Value**
90 = 10 11010 song 3, note 26	{6'd43, 6'd6}	Note 43 = D# or Eb 4	6/48ths
91 = 10 11011 song 3, note 27	{6'd44, 6'd14}	Note 44 = E 4	14/48ths
92 = 10 11100 song 3, note 28	{6'd0, 6'd28}	Note 0 = rest (silence)	28/48ths

When the third 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 next note will be the 4th E for 14/48ths of a second, followed by silence (rest) for 28/48ths of a second. The frequency_rom contains the definitions of each of these notes. For example, entry 44 in the frequency_rom (E 4) will contain the frequency for playing an E 4 note, which the note_player will then use to actually generate the note. I'm assuming that everyone has been exposed to basic music notation and understands what an A# or Bb is. If you don't have any idea about these you should be sure to look them up.

Both the song_rom and the frequency_rom are generated using the song_rom worksheet Excel document, 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.)

Song Reader Interface:

Signal	**Direction**	**Description**
clk	input	clock
reset	input	reset signal
Play	Input	True if the song reader should be playing.
Song[1:0]	Input	The song to play.
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.
Current_note_and_song[6:0]	Output	A concatenation of the current song (MSBs) and the *address* of the current note (LSBs) being played. This is displayed on the 7-segment LEDs on the board for your amusement. Note that

		this is the address of the note and note the note itself.
Note_done	Input	From the note_player to indicate that the note has finished and that it is ready for the next note.
State[2:0]	Output	The internal state of your song_player, used for debugging.

Hint: 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 along the lines of $display("Playing note %d of song %d, which is note %d duration %d", note_address, song, note, duration);. That way whenever you were debugging it would tell you what note it was playing.

Note Player

The note_player is the central part of this lab. Its responsibility is to take in a note, lookup the step size required to generate the correct frequency for that note in the frequency_rom, and then synthesize a sine wave at that frequency by using the step size to walk through the sine wave in the sine_rom. To make this simpler you will design a separate module (the sine_reader) which takes in the frequency and generates the individual samples. If we pull out the sine_reader, it simplifies the note_player so all it needs to do is store the current note and duration it receives from the song_reader, keep playing it for the duration of the note, and then tell the song_reader it's done. When the note_player gets the note it needs to look up the step size for the note in the frequency_rom and send that off to the sine_reader. The note_player also needs a counter to keep track of the duration of the note it is playing. This counter counts down every 48th of a second that the note is playing (remember that if we pause we don't keep counting down). The 48th beats are generated by a beat_generator module (which we provide). The beat_generator uses the same signal from the codec_conditioner to generate its beat.

Note Player Interface:

Signal	Direction	Description
clk	input	clock
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.
State[5:0]	Output	Debugging output of your state.

Sine Reader

The sine_reader is a sub-module of the note_player which takes in a step size and generates a repeating sine wave that steps through the sine wave, generating a frequency determined by the step size. This is really pretty simple. If we have a sine wave in a ROM, all we have to do is walk through it at different rates. If we want a higher-frequency tone we walk through the sine wave faster (bigger step size). For lower frequencies we walk through the sine wave more slowly (smaller step size). This allows us to generate sine waves at different frequencies by either skipping samples or repeating samples. (If you're not following this draw out a sine wave

and draw out the resulting sine wave if you walk through the original one with larger or smaller steps.) So whenever the sine_reader receives a generate_next signal it just adds the step size to the current address to get the address of the next sample of the sine wave. It then gets this data from the sine_rom and outputs it as the next sample. Since we're counting in binary the address will automatically wrap around at the end and we're all set! Sound easy? Well, it is.

…kind of. No, really, it's not that complicated. There are three gotchas with this part. The first is that the sine_rom takes one cycle to output its value, so you need to make sure you're thinking about this when you design your logic. (I.e., draw a timing diagram!) The second is that our sine_rom has 1024 samples, but we need more precision than that to get good tones. I.e., we want to be able to specify a step size of 10.5 not just 10 or 11. For example, a really low tone might have a step size of 0.01. Such a small step size would mean that the sine_reader only goes on to the next sample in the sine wave every 100 cycles (0.01*100 = 1.00). How are we going to do this? Well, it's actually very easy: we'll use a 20 bit step size, but we'll treat the lower 10 bits as fractional bits. That is, we'll keep our address counter as 20 bits, but we'll only use the upper 10 bits to actually access the ROM. For example, if we have a value of 10.5 for our step size (0000001010.1000000000), our first addition will give us address 10:

0000000000.0000000000 + 0000001010.1000000000 = 0000001010.1000000000
but we only use the top 10 bits as the address to the sine_rom, which are 0000001010 = 10

but our second addition will give us 21:

0000001010.1000000000 + 0000001010.1000000000 = 0000010101.0000000000
but we only use the top 10 bits as the address to the sine_rom, which are 0000010101 = 21

Remember that we can interpret numbers however we want, so there's nothing special about treating this as a 10.10 number, except that we need to make sure the frequencies are calculated that way. (Indeed, the frequency_rom we've provided is calculated exactly that way. You can even change it in the spreadsheet if you want.) You should make sure this bit is clear as it is the second most confusing part of the whole lab.

So that's not too bad. We're using a 20 bit counter and taking the top 10 bits as the address. Now let's go back to the third sentence in the first paragraph: the one where I said, "If we have a sine wave in a ROM…" well, it turns out you don't. All we've given you is a quarter sine wave. So that kind of sucks, until you remember that sine waves are symmetric. The second quarter is just the first flipped horizontally, and the last two quarters are the same but negated.

Okay, so how do we deal with this? Well, let's see if we can play the same trick we did with the numbers above. If we increase our address to 22 bits, we can use the top 2 bits to divide the sine wave into 4 sections. Then all we need to do is adjust our address and value depending on which quarter we're in. (You're going to want to draw a picture to make sure you've figured this out. This is the most confusing part of the lab, and it's really quite easy once you draw the picture.)

So the sine_reader is the most complicated part of this project, but it's also the most fun to debug. When you are working in ModelSim you can display the output as a sine wave by right-clicking on it and choosing Properties and then setting the formatting as follows: Height:80, Analog Step, Offset: 40000.0, Scale: 0.001. You'll then get a cool sine wave on screen. Once you've got your waveform all setup in ModelSim you can save it to a .do file so you can easily re-load the display configuration later. This will prevent you from having to re-add the signals every time you run ModelSim.

Sine Reader Interface:

Signal	Direction	Description
clk	input	clock
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.

Codec Conditioner

So far I haven't mentioned anything about how your design interfaces with the actual audio hardware on the FPGA board. This is done through the ac_97if module. This module gives you a new_frame signal which goes high every time you're supposed to provide a new sample. The tricky part is that you are supposed to provide a new sample as soon as it goes high and not change it until it goes high again. This is a pain.

Remember all those times I've mentioned that your ROM takes one cycle to output its value? Well, how are you supposed to provide the next value immediately when new_frame goes high if you need one cycle to get through the sine_rom, one cycle through the frequency_rom, and possibly one cycle through the song_rom? Well, the solution is that you'll have a codec_conditioner. This module simply has two registers. The current sample is output from one of them, and the other is where you write the next sample whenever you get around to it. That way when the next new_frame signal comes in, the codec_conditioner just copies the next value into the current one and tells the note_player to generate the next one. As long as you can generate the next value before the next new_frame comes along it will all work. And that shouldn't be a problem.

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. Note that I mentioned that the beat_generator runs off this signal as well. This is convenient because it means that instead of having to divide 100MHz down to 48Hz we only have to divide 48kHz down to 48Hz. It also means that if the audio chips clock drifts a little bit we'll still be in sync with it.

To make this lab a bit easier we're providing you with the codec_conditioner module. We're providing the ac_97if module as an .ngc file, which is a pre-compiled netlist. All you have to do to use this is copy the ac_97if.v and ac_97if.ngc files into your Xilinx directory. (However, if you use the "clean up project files" command you'll have to copy the .ngc file back again.) We also provide a codec_sim.v module which simulates the codec with a new_frame every 5 cycles for faster simulations.

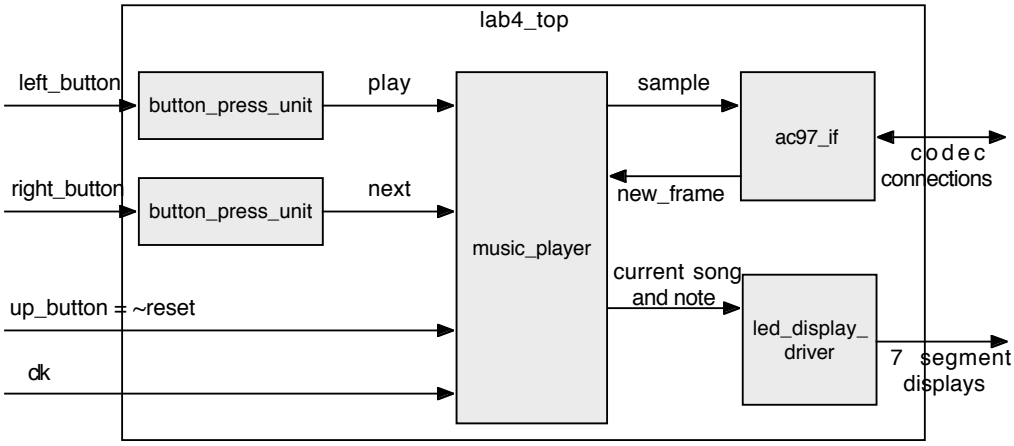
Codec Conditioner Interface:

Signal	**Direction**	**Description**
clk	input	Clock
reset	input	reset signal
New_sample_in[15:0]	Input	The new 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	The input from the ac97 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 constitute a handshake between the codec_conditioner and the note_player.

Note: the ac_97if.ngc may generate a few warnings about equivalent registers. You can safely ignore these.

Other Details…

There are a few other details you'll have to take care of for this lab. The first one is that you'll need to synchronize, debounce, and one-pulse your inputs from the buttons on the board before using them in your FSMs. Remember that the mechanical buttons on the FPGA board are really little springs, and that when you press one down it will bounce up and down randomly for about 20ms. (That's 2,000 clocks.) If you don't debounce them you will get somewhere between 1 and 2,000 button presses into your FSM every time you press a button. We've provided a button_press_unit module which chains together a brute_force_synchronizer, a debouncer, and a one_pulse module. This is instantiated for both buttons in the top level module as follows to provide clean inputs for you. (This is the same module you used in lab 3.)



We've also provided you with a music_player module which already hooks up all of the sub-modules. You should not need to modify this. This module has an output called new_sample which may generate a warning saying it is assigned but not used when you compile it in ISE. Don't worry about this. You will need this output for lab 5 to determine when you should start capturing the wave for the VGA display.

It would also be nice if you could see what your design is doing, although you should have done all your debugging in ModelSim. To help out I've provided an led_display_driver module which outputs four 4-bit numbers to the four 7-segment displays on the FPGA board. The current_note_and_song from the song_player goes into it so you see the song on the left and which note you're playing on the right. This is all set up for you in the top level file, but please take a look at it and feel free to modify it if you can think of anything cooler.

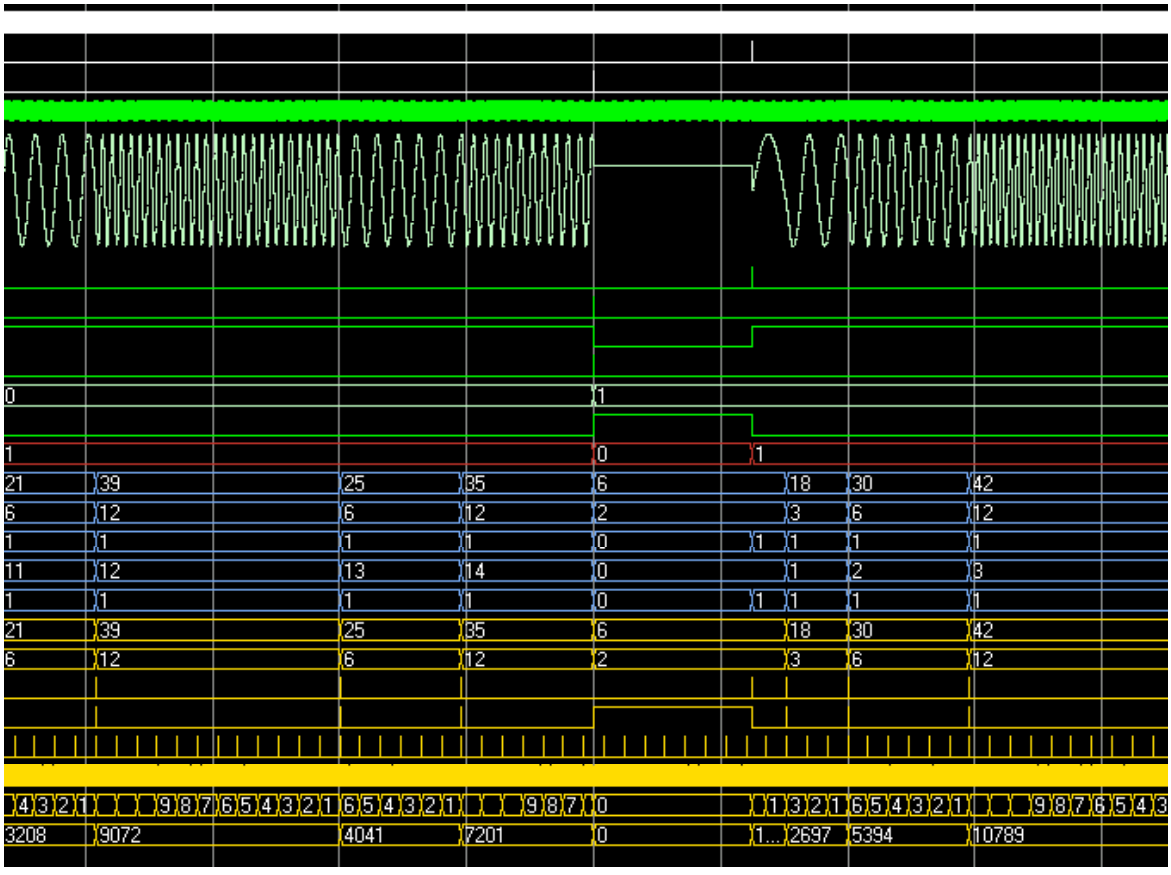
If you can't get your design to work we've provided a ChipScope interface in the music_player module and a ChipScope project file. This will allow you to see what's going on inside your design while it's running, but this is a very painful way to debug so don't plan on using it. The ChipScope modules (ila and icon) may generate warnings when you compile. You may ignore these.

What you need to do…

Read this whole document. At least twice.
If you haven't read all of the above don't even bother reading the list below as it will just be a waste of time.

1. Understand all the signals for each module. In particular how they flow and what causes them to change.
2. Draw FSM state diagrams for each module which requires an FSM.*
3. Draw timing diagrams for the song_reader, note_player, and sine_reader showing the delay in reading from the ROMs.*
4. Implement the mcu, song_reader, note_player, and sine_reader.*
5. Write test benches for all of the above, and include the output in your report as well as the testbenches themselves.*
6. After those test benches work, run the provided top-level test bench to make sure your whole system works.* (We've provided a .do file which you can open in the Wave view in ModelSim to load all the signals automatically. When you're simulating the top-level file you can load it into your wave window with the open command. This test bench takes a long time to run...)
7. Write a song of your choice for song 4. (There is a provided Excel spreadsheet which makes writing songs a lot easier.)

All of the above marked with an * need to be included electronically in your prelab.



Here's an example of the output from the provided top-level testbench. Note that you can see the different frequencies and lengths of the notes playing, as well as the states of the various modules. You should refrain from running this testbench until you've thoroughly tested each of your modules individually as this takes about

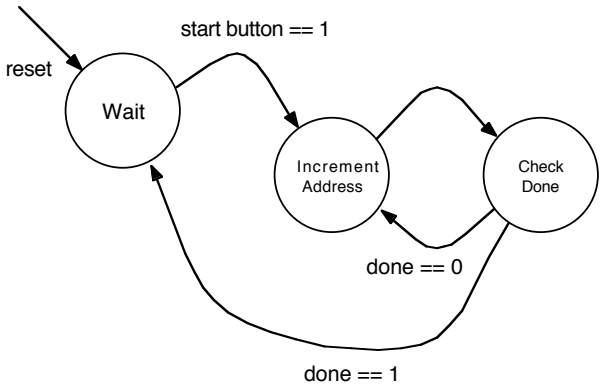
6 minutes to run. (Your results may vary depending on what is in your song_rom, and you may want to change the song_rom to make it easier to debug.)

Timing Diagrams

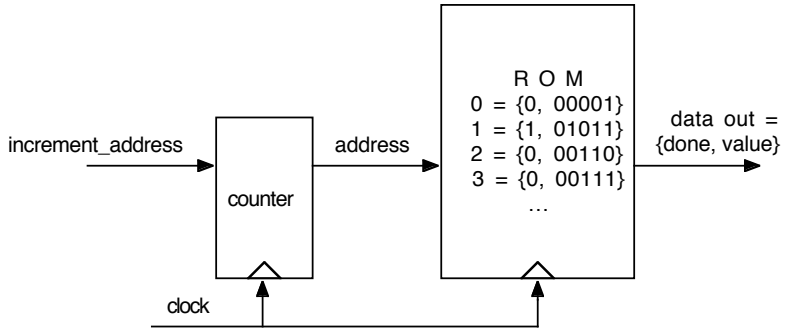
I've mentioned several times in this lab that you need to do timing diagrams for various modules. The point of the timing diagrams is to make sure that your FSM and your other modules are going to work in sync. Here's a simple example to get you started.

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.

Finite State Machine Control



Data Path



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!