EE108 Lab 5

Waveform Display

Lab Due: October 28th, 2020 at 11:59pm

Objective: The purpose of lab 5 is to render the waveform being outputted by lab 4 to a raster display.

1. Introduction

So, what are we doing here?

In lab 5, you will use the audio samples generated in lab 4 to display a real-time waveform on a computer monitor. The display will show the actual sine wave that you are sending to the speakers. The display should be roughly centered horizontally, and should show the full sine wave in the upper half of the monitor, vertically.

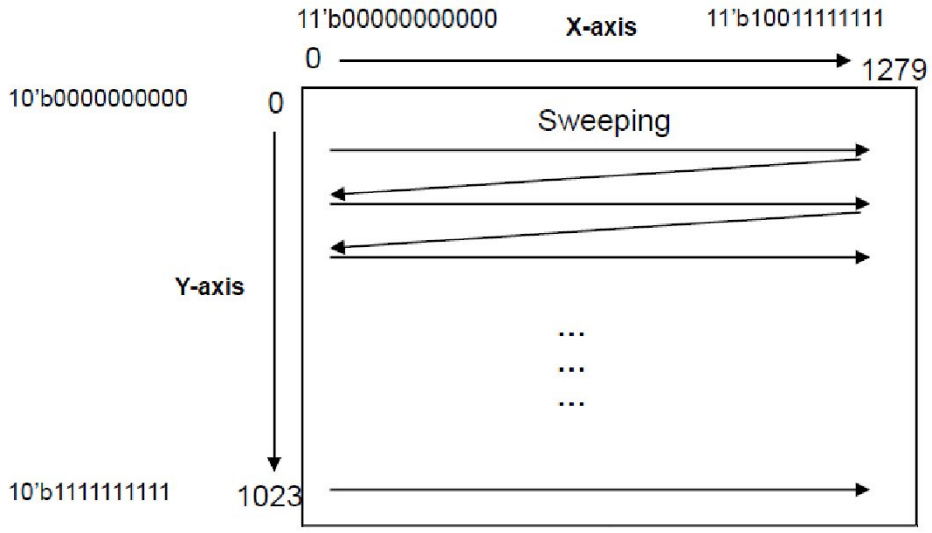
You will be reusing the modules you developed for lab 4. If you were unable to get lab 4 working, let us know and we can provide a working mcu, song_reader, and note_player. However, we highly recommend that you use your own modules (and get them working if they aren't already), as the final project will be built off of your lab 5 submission and it will be much easier to use code that you have written and understand.

We have provided empty files for the modules related to lab 4. You will need to copy in your implementations for the following files:

- music_player.v
- mcu.v
- note_player.v
- sine_reader.v
- song_reader.v
- song_rom.v

Raster displays

Before we discuss the inner workings of the wave_capture and wave_display modules we need to understand how the VGA display maps the X-Y coordinate plane onto the screen. The diagram below illustrates this concept:



VGA sweeps across a row of pixels from left to right and then moves down to the beginning of the next row and starts again, like a typewriter. Displays that draw images this way are called **raster displays**.

For this lab, that means on every clock cycle your wave_display module will be given the X and Y coordinates of a screen pixel and on the next cycle you have to output what color that pixel should be. The order these coordinates will be given to us is exactly the pattern outlined in the diagram above.

Color representation
Color is defined by 24 bits: 8 bits each for red, green, and blue. The larger each of those 8-bit numbers are, the brighter that color will be. So, 24'hFF0000 is bright red, 24'h00FF00 is bright green, 24'h0000FF is bright blue, 24'8F0000 is dark red, 24'h808080 is dark gray, and 24'hFFFFFF is white. In case you haven't seen it before, the "h" in 24'h refers to hexadecimal, or base-16 numbering. In computing, 24 bit colors are almost always written this way: as a 6 digit hexadecimal number with the two left-most digits referring to red, the middle two referring to green, and the right two referring to blue.

A green box
Just as a quick example, the following logic inside your wave_display module would output a green box roughly in the middle of the screen:

```
if (x > 11'd600 && x < 11'd800 && y > 10'd400 && y < 10'd600 && valid)
   {r, g, b} = 24'h00FF00;
else
   {r, g, b} = 24'h000000;
```

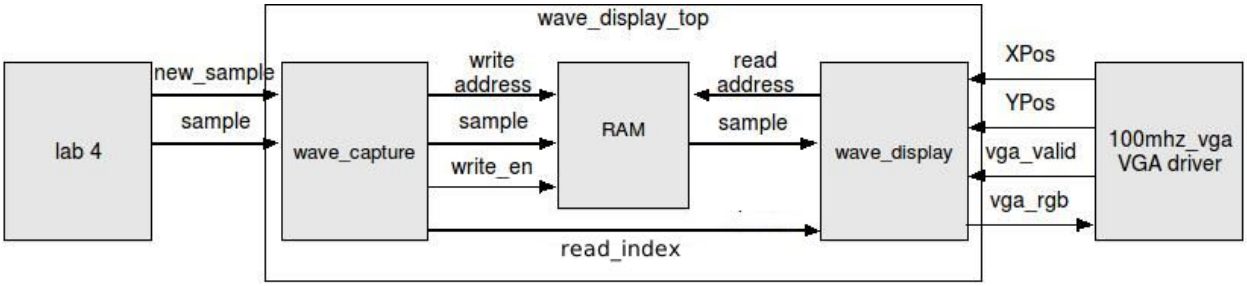
This would turn on the green color whenever the VGA was inside the box defined by x between 600 and 800 and y between 400 and 600.

Architectural Overview

The wave display is made up of two modules:

1. The **wave_capture** module, which collects the individual samples being outputted by your lab 4 design and reassembles them in a RAM
2. The **wave_display** module, which is given screen coordinates by the external VGA driver chip and figures out what color the pixel at that coordinate should be by reading the RAM

Below is a schematic of how these modules are connected and how they interface with the outside world:



RAM

You are provided the dual-port RAM module in the middle of the diagram. This RAM is similar to the ROMs we worked with in lab 4, except it allows us to write data into an address by asserting its 'wea' (write-enable port a) input high, which will cause it to store whatever value is on its 'dina' port (data-in port a) at the location specified on the 'addra' (address port a) input on the next clock cycle.

That the RAM is dual-ported means it has two address inputs and two data outputs, which allows multiple modules to simultaneously access the same memory. The RAM provided in ram_1w2r.v has 1 read-only port and 1 port that can perform both reads and writes. The two ports operate independently, allowing you to write in new data while reading out old data. The port connected to the wave capture module performs reads and writes, and the read-only port is connected to the wave_display. The RAM stores 512 separate 8-bit words, so the address is 9 bits wide.

Double Buffering

We have to deal with the contingency that wave_capture might be changing values in RAM at the same time wave_display is reading them. Though our dual-ported RAM supports this just fine, it will cause our waveform to look distorted.

Let's say wave_display renders the top half of the wave, recalling that it draws one full row of pixels and then moves down and draws the next. Then, halfway through, wave_capture starts

receiving new audio samples and rewrites all the values in RAM to something else. wave_display will render this new waveform instead of the wave it was previously rendering, and the result will look like two different images cut in half and then glued together.

The solution to this is **double buffering** – we split the addresses in our RAM into a top half and a bottom half, and we make sure our two modules never touch the same half of the RAM at the same time. We will store a single "read index" bit in wave_capture. When this bit is 1, wave_capture should write to addresses 0-255 and wave_display should read from addresses 256-511. When this bit is 0, wave_capture should write to 256-511 and wave_display should read from 0-255.

The logic for determining when to flip this read index is described in the wave capture section of this handout.

Note that we could also implement double buffering with two discrete RAM modules, but this would increase the overall complexity of our project since we would need twice as many wires for all of their ports and multiplexers to route signals to one RAM or the other. By unifying the buffers, all we have to do is add an offset to our read and write address.

Wave Display Top

You will put all your modules and logic in the wave_display_top module. This module is instantiated in the provided lab 5 top module and interfaces between your music player implementation from lab 4 and the wave_display and wave_capture modules you will implement in this lab.

Wave Display Top Interface:

Signal	Direction	Description
clk	Input	Clock signal
reset	Input	Reset signal
new_sample	Input	True if the incoming lab 4 audio sample is a new value
sample[15:0]	Input	The current sample from lab 4 audio logic
x[10:0]	Input	The current X position of the VGA display
y[9:0]	Input	The current Y position of the VGA display
valid	Input	Whether or not the VGA coordinates are valid or just garbage
vsync	Input	Stays at 1 most of the time and goes to 0 for a short time immediately following the last line of pixels on the screen. Indicates the VGA display has rendered a frame and is waiting a while before starting the next one.
r[7:0] g[7:0] b[7:0]	Output	24 bits of VGA color data: 8 bits each of RGB. This is automatically generated by taking the output colors of wave_display and ANDing it with its valid_pixel output.

Wave Capture

Wave Capture Interface:

Signal	Direction	Description
clk	Input	Clock signal
reset	Input	Reset signal
new_sample_ready	Input	A one-cycle pulse indicating the next new sample is ready
new_sample_in[15:0]	Input	The new audio sample
wave_display_idle	Input	High when wave_display is not rendering a frame
write_address[8:0]	Output	The address of the location to write in the RAM
write_enable	Output	High when the module needs to write to the RAM
write_sample[7:0]	Output	The top bits of the sample to write into the RAM
read_index	Output	Bit indicating which part of RAM wave_display should read from

The waveform capture implements a FSM that has three states: *armed, active,* and *wait*.

The FSM is initially in the armed state. In this state it waits to see a positive zero crossing on the audio output (i.e. a negative number followed by a positive number). Once a zero crossing is seen, the capture FSM switches to the active state. In the *active* state it stores the 8 most significant bits of the next 256 audio samples into the RAM. Once it writes the final sample, it moves into the *wait* state and sits there until *wave_display_idle* is 1. Once *wave_display_idle* is 1, it inverts the read index bit and moves back into the *armed* state.

The *wait* state ensures that we don't flip read_index while wave_display is reading from RAM or while wave_capture is writing to RAM. The next frame wave_display renders will be the latest data captured and new audio data will be written to the currently unused buffer.

You will need to keep track of the next address to write to using a counter. In the FSM's *armed* state, this counter is set to 0. In *active*, on each new sample from the music player, we store the value to address {~read_index, counter} and increment the counter by one. Make sure you only store audio samples when a new sample is provided (i.e. when the new_sample_ready output from the music player is high).

It is important to consider the difference in formats between the audio samples and the Y-coordinate values used with the display. Remember that the audio samples are signed 2's complement values centered around 0, but we need to store a positive number between 0 and 255. There is one short line of code required to make the adjustment.

Draw a timing diagram to ensure that your FSM captures the first positive sample to the correct address in the RAM. Consider why we're taking the 8 most significant bits, and what might happen if we took the least significant bits instead.

Wave Display

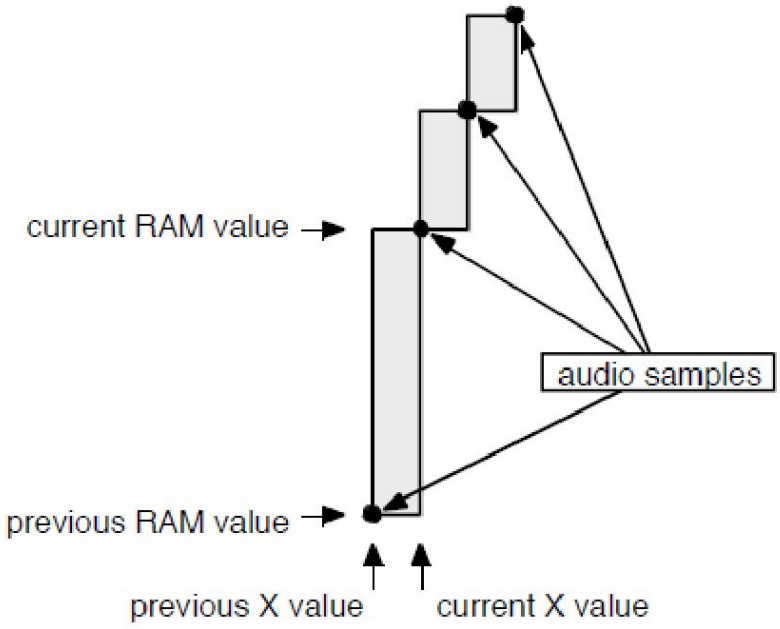
Wave Display Interface:

Signal	Direction	Description
clk	Input	Clock Signal
reset	Input	Reset Signal
x[10:0]	Input	The current X position of the VGA display
y[9:0]	Input	The current Y position of the VGA display
valid	Input	Whether or not the VGA coordinates are valid for displaying data.
read_index	Input	Bit indicating which part of RAM to read from
read_value[7:0]	Input	The data you read back from the RAM.
read_address[8:0]	Output	The address in the RAM to read. Remember: it takes one cycle to get the data back!
valid_pixel	Output	True if the pixel specified by x and y should be turned on.
r[7:0]	Output	The color you want the wave to be. This can be assigned a

g[7:0] b[7:0]		constant value.

The basic idea

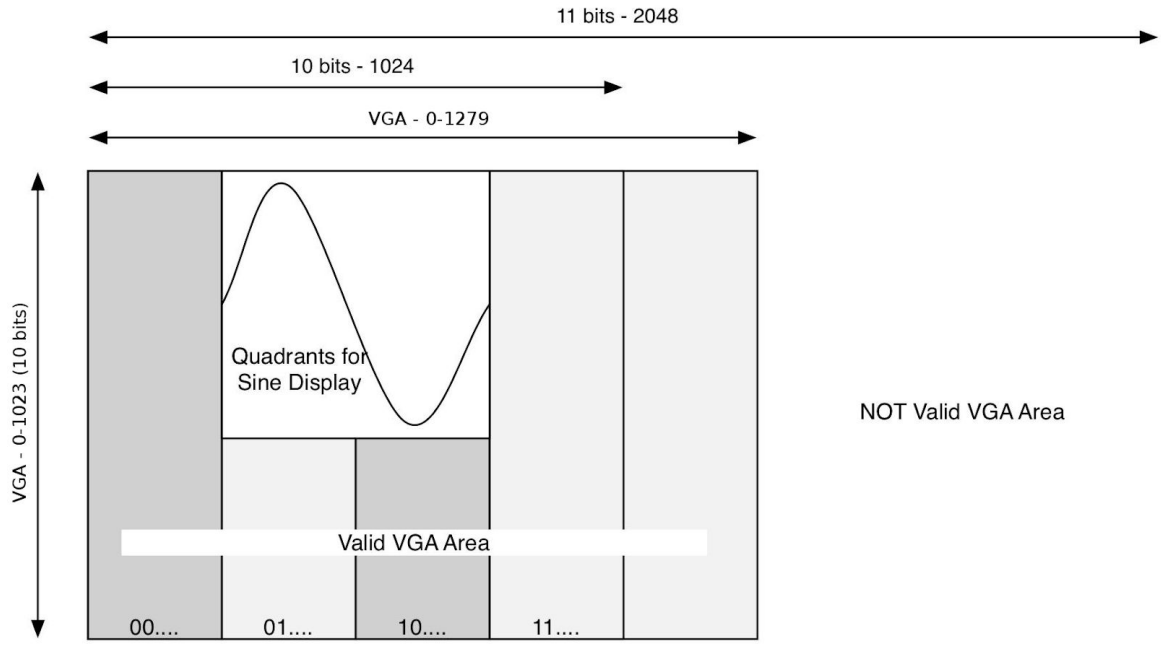
We can think of each X coordinate on the screen as mapping to a specific address in the wave capture RAM, and the Y coordinate as mapping to the value at that address. At any given coordinate (X,Y) the pixel should be colored white if Y value is between RAM[X-1] and RAM[X]. Because VGA scans across the screen in horizontal lines, we can say that the Y coordinate should be between the current value read from RAM and the previous value read from RAM. Represented visually:



Keep in mind that the waveform may also be moving downwards, in which case the previous RAM value will be greater than the current RAM value. Your logic has to deal with both of these possibilities.

Translating from (X,Y) to RAM addresses and values

You should design your display to place the full waveform in the quadrant shown below, by manipulating the bits of the X and Y coordinates:



First we have to translate the 11-bit X coordinate into a 9-bit RAM address. Drop the least significant bit of X to make the line we're drawing two pixels wide (this will make it more visible) and rearrange the bits of the signal as shown in the example below. Bits that are underlined should be dropped:

1st quarter of screen: x = 11'b000xxxxxxx → read_addr = //don't care//
2nd quarter of screen: x = 11'b001xxxxxxx → read_addr = {read_index, 8'b0xxxxxxx}
3rd quarter of screen: x = 11'b010xxxxxxx → read_addr = {read_index, 8'b1xxxxxxx}
4th quarter of screen: x = 11'b011xxxxxxx → read_addr = //don't care//

Concatenating the read index to this address ensures we only read from the half of the RAM that wave_capture is not currently writing to.

Translating the Y coordinate is a bit easier. We already designed wave_capture to transform the audio samples into positive Y values, so we just need to make sure we only draw pixels when we're in the top half of the screen:

Entire screen: y = 10'bxxxxxxxxxx → Top half: y = 10'b0xxxxxxxx

We drop the LSB of the 10-bit y value so that we compare signals of the correct size and each point we draw will be two pixels high. We drop the MSB because we're only displaying the wave in the top half of the screen, where it will always be 0. This is also why we only stored 8 bits of the sample in the RAM to begin with; storing any more would just have been a waste of memory.

If this is not clear you should draw it out on graph paper and work out which parts of the screen

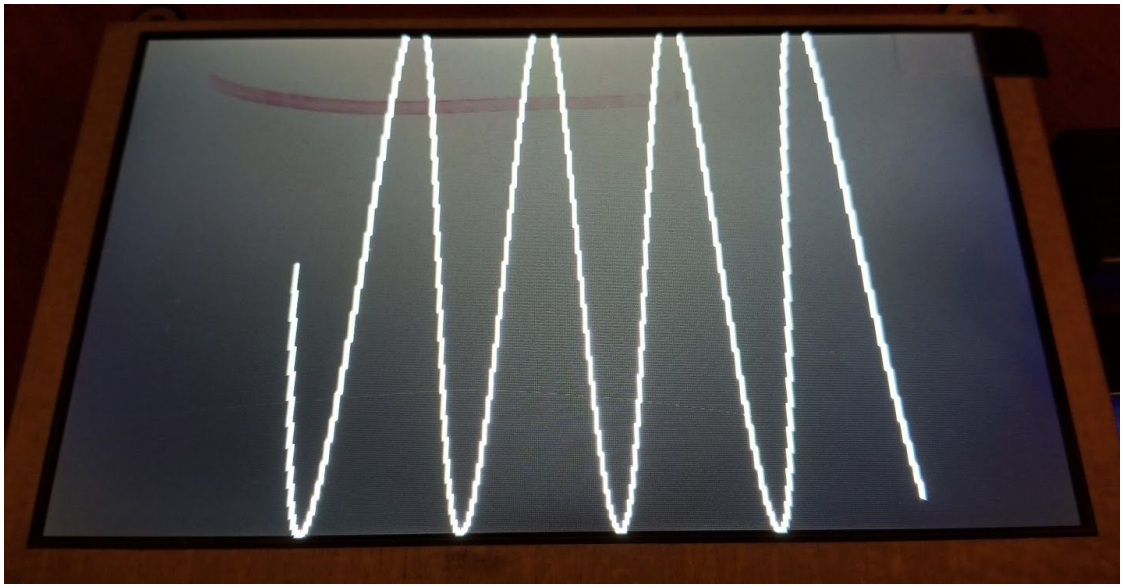
we are displaying in and how this maps to the RAM address.

Finally, we want to make certain that we only display waves while we are in the middle quadrants of screen, and while in the top half of the screen. To accomplish this, we want to make sure that we always output valid_pixel as 0 when we are not in quadrants 1 or 2, and when we are not in the top half of the screen. We can do this simply by looking at bits 10 and 9 of x and the MSB of y.

Remember that the RAM has a one cycle latency for outputting the data after an address change and that your Y bound is determined by the last two samples read from the RAM. The X value given to you will change on every clock cycle, but we also drop the LSB of the X-position to make our lines twice as thick. This means our RAM read address will only change every other clock cycle. You will need to account for this so that you're not interpreting the same data sample from the RAM as the last two data samples. The easiest way to accomplish this (in terms of getting the timing correct) is to detect when the read_addr signal changes and only then accept a new data sample from the RAM.

The end result of all of this should be a clean waveform, without vertical lines on the ends or other artifacts.

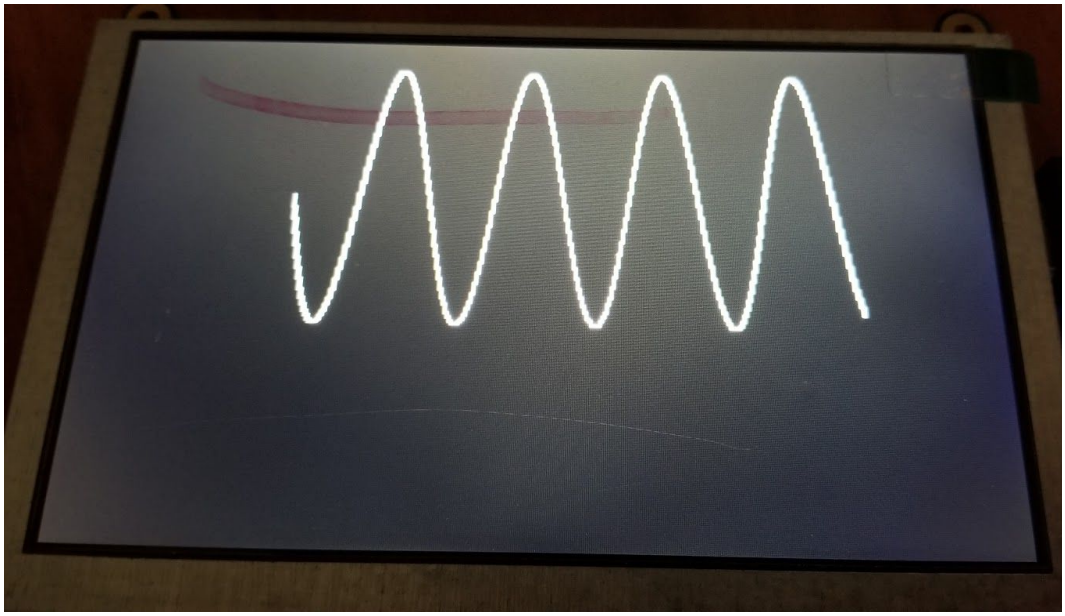
A quick note about the display used for remote coursework: this lab was originally developed for a full-size display of size 1280x1024, which is the size of the displays in the EE 108 lab. However, the display we've provided in your lab kit has size 800x480. If you've done everything correctly, you should see something like the following:



Note that the upper few pixels of the waveform pictured above are cut off. This is because of the difference between the two displays. You should fix this by creating an internal 8-bit signal, `read_value_adjusted`, according to the following formula:

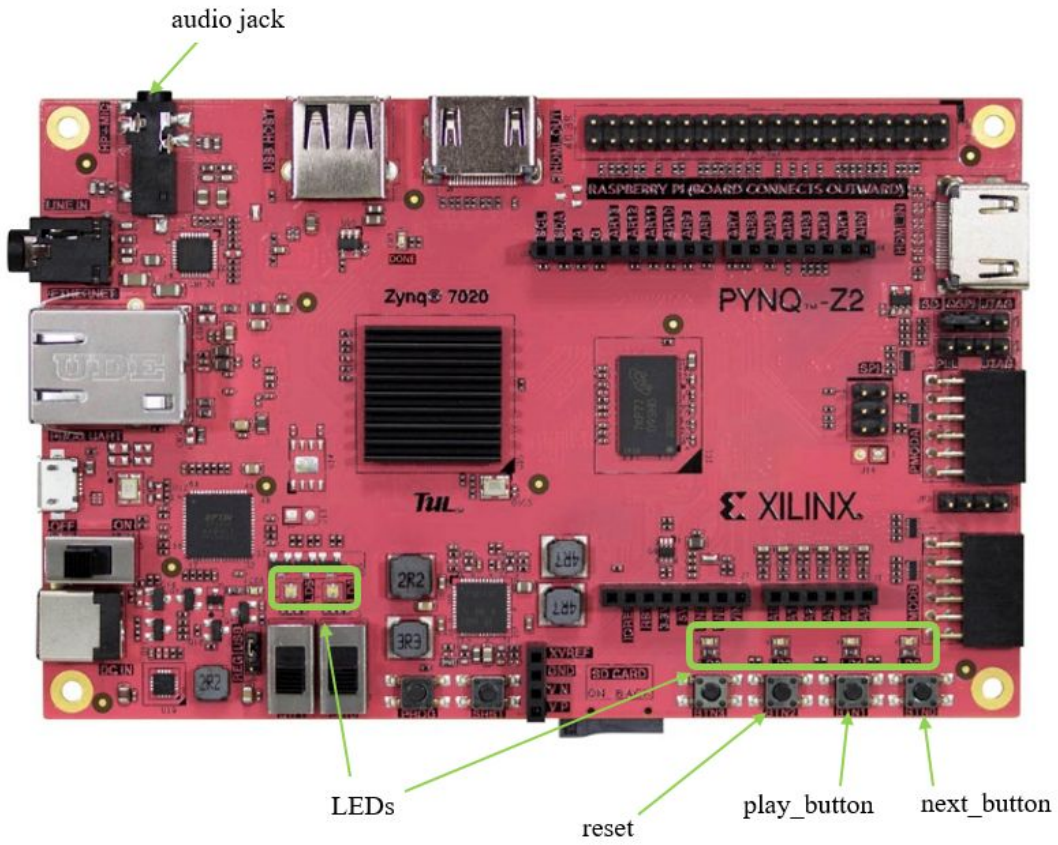
$$read\ value\ adjusted\ = \frac{read\ value}{2} + 32$$

Recall that you can easily divide by 2 by bit-shifting right by 1 bit. Then, within wave_display, use `read_value_adjusted` everywhere you normally would use `read_value`. This way, the waveform will not be truncated, and it will take up approximately the top half of the screen:



Board interface

The board interface is the same as lab 4. The diagram is copied below for your convenience. **As a reminder, be careful with the volume level! We haven't programmed volume control, so the FPGA is always at max volume.**



Setup:

There are a couple more steps to this lab, since it's the most complicated and you will be reusing code from lab 4. After you download starter files from on Canvas, follow these instructions:

- Copy music_player.v, mcu.v, note_player.v, sine_reader.v, song_reader.v, and song_rom.v from lab 4 into the starter files provided.
- Add design sources, simulation sources, and constraint sources.
- Set up the github repository.
- Adding/customizing IP. Just like in lab 2, you only need to use the HDMI TX and clock wizard IP for this lab. Make sure that the HDMI TX is named hdmi_tx_0, and that the clock wizard is named clk_wiz_0 and you set CLK_IN1 to sys_clk , clk_out1 to 100MHz, clk_out2 to 25MHz, and clk_out3 to 125MHz.

Other things

Make sure you have a good block diagram BEFORE you start writing your code. It is up to you to determine the infrastructure of your modules, but please use good style and write readable and efficient code, and maintain a good level of modularity. Labs turned in with poorly commented Verilog will be severely penalized.

Your final submission must be free of timing errors for the 100MHz clock. You can see if your design violates timing constraints by checking the timing report. If you do have to locate the critical paths and break them up with a flipflop somewhere in the logic chain. You can find these critical paths by checking the timing report file (.rpt file), or by checking the worst negative slack in Vivado. Note that the starter code has a few timing errors, which you don't need to worry about. These are:

- wd_top/sample_ram/… -> U3/inst/srldly_0/…
- vga_control/hcounter_reg/… -> wd_top/wd/…
- vga_control/hcounter_reg/… -> wd_top/sample_ram/…
- adau1761_codec/… -> wd_top/wc/…
- adau1761_codec/… -> music_player/…
- music_player/… -> adau1761_codec/…

It is strictly forbidden to divide clocks anywhere in this class (ask your TAs why, if you'd like to know). All of your logic should run off the same 100MHz clock and you should use enables in FSMs to deal with slower sample rates. The display itself is high-resolution and progresses very quickly; you may have to pay special attention to your design so that you don't fall behind!

You can force **initial blocks** in your testbench to pause until the DUT renders a full frame by using the command "**@(negedge chip_vsync);**" (do not use this in always blocks!). chip_vsync is a signal that triggers for a few cycles when the video driver has scanned all the way to the bottom of a frame. There is a chip_vsync shortly after reset, before anything gets rendered, so to halt until the first frame to complete you might have to put an initial pause before it, or put two in a row.

A fake_sample_ram module has been provided, allowing you to test your display without having to have a working wave_capture implementation. Definitely use (and modify) this module to test your wave_display—the timing interactions of audio, the song, and will result in a very strange first frame when simulating the entire system. Providing known (unchanging) values like the fake_sample_ram does is a great way to make sure things are working as they should.

For fun: you've got 24 bits of color, which means your VGA display can show 16777216 different colors. It's pretty easy to have the color change based on what you're displaying (just turn some groups of bits on or off) so you might consider doing some cool color display for your waveform. Remember to obey the VGA valid signal or the monitor will not display your output.

2. Lab submission

Put the following in an archive and submit it on Gradescope before the due date:

1. FSM state diagram for wave_capture and block diagram for wave_display
2. Timing diagrams for wave_display showing the delay in reading from the RAM. These

can either be in the form of tables or waveforms (drawn, not generated).

3. Your code:
    a. wave_display.v
    b. wave_capture.v
    c. wave_display_top.v
    d. wave_display_tb.v
    e. wave_capture_tb.v
4. Annotated output from all of your testbenches.
5. The **critical path** of your design. Select "Design Summary/Reports → Detailed Reports → Post-PAR Static Timing Report" and tell us the **slack, source, and destination** of the most critical path in the design.
6. Synthesize your design and fix any timing issues you encounter. Include the .twr and .syr files generated in your project's directory. **You should submit a bitfile that clears all timing requirements for the 100MHz clock.**
7. A picture, video, or something else to demonstrate that your design works on the actual board.