Lab 5: Waveform Display

Version 1.1 – David Black-Schaffer

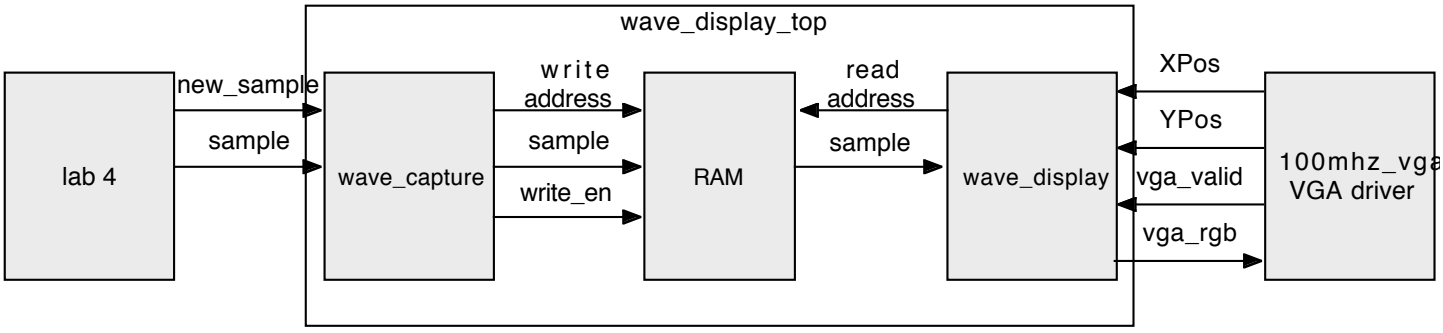
Introduction

In this lab you will take the audio samples generated in lab 4 and use them to display a real-time waveform on the VGA monitor. The display will show the actual sine wave waveform that you are sending to the speakers in real-time.

You will be reusing the modules you develop for lab 4. If you were unable to get lab 4 working the provided starter files for lab 5 include a copy of the mcu, song_reader, note_player, and codec_conditioner. Otherwise please use your own modules as you will be basing the final project on lab 5 and it's better to use your own code that you understand.

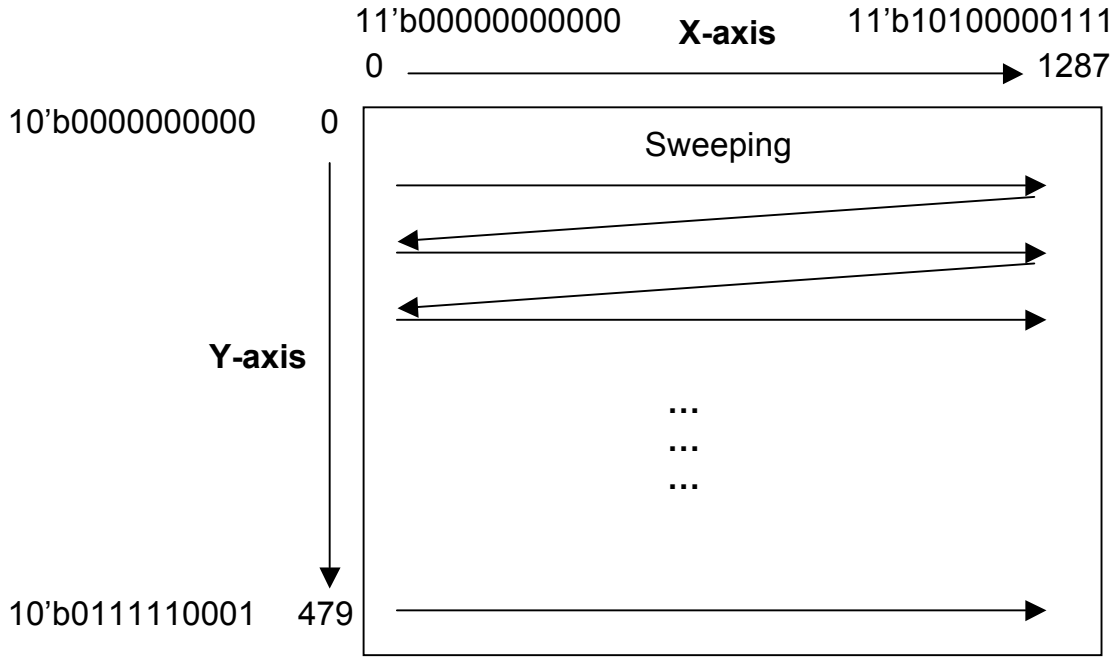
Overview

Below is a schematic of the design you will be implementing, which consists of three components. You are provided with the dual-port block RAM in the middle of the schematic. (A dual-port RAM is a memory with two independent ports. In our case we have 1 read/write port and 1 read port. The two operate independently, so you can write in new data while you are reading out old data. If you read and write to the same location at once it's not clear what will happen, so if you care about your data (and this is not necessarily the case) you need to think about this.. For this lab the RAM is defined in the ram_1w2r module.) The dual-port RAM allows you to access the same memory space from two different ports at the same time. As you can imagine this is a very useful feature. In this lab, each of the two ports has a fixed behavior. The port connected to the wave capture module is a write-only port, and the port connected to the wave_display is a read-only port. The RAM stores 8-bit words and is 256 deep (so the address is also 8-bits wide). The wave_capture module is responsible for storing appropriate chunks of the audio samples into the RAM, while the wave_display reads those samples out based on the X-position of the display and then determines whether a color should be displayed or not based on the Y-position (more on this in next section). Your job is to write the FSM and logic to capture the correct portions of the audio signal and store them into the RAM and build the logic to turn on the VGA colors when the screen scans over the area representing the signal you stored in the RAM earlier.



VGA Display

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 and Y coordinates on the screen. The diagram below illustrates this concept.



The sweeping is done by rows starting with (0, 0) which means that for each Y-position the entire X-axis is scanned before iterating to the next Y-position and repeating the process. The sync_gen100 module in the 100mhz_vga.v file is responsible for running these counters and telling you when the X- and Y-positions are "valid" on the screen. Your job is to watch the X- and Y-counter outputs and the valid signal and turn on the appropriate colors. The color is defined by 6 bits: 2 for each of red, green, and blue. When the bits are 1 the colors are turned on as specified, i.e., 6'b110000 is bright red, 6'b010000 is dark red, 6'b010101 is dark gray, and 6'b111111 is bright white.

For example, if you wanted to have a green box in the middle of the screen you would use code along the lines of:

```
if ((XPos > 11'd600) && (XPos < 11'd800) && (YPos > 10'd200) && (YPos < 10'd300) && VGAvalid)
        color = 6'b001100;
else
        color = 6'b000000;
```

This would turn on the green color whenever the VGA was inside the box defined by XPos between 600 and 800 and YPos between 200 and 300.

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 hooks the code from lab 4 and the VGA module to your lab 5 code.

Wave Display Top Interface:

Signal	Direction	Description
clk	input	clock
reset	input	reset signal
XPos[10:0]	Input	The current X position of the VGA display
YPos[9:0]	Input	The current Y position of the VGA display
Vga_valid	Input	Wether or not the VGA coordinates are valid for displaying data.
Vga_rgb[5:0]	Output	6 bits of VGA color data (red 0,1 green 0,1 blue 0,1)
Sample[15:0]	Input	The current sample.

New_sample	Input	True if the sample is a new value

Wave Capture

The waveform capture subsystem waits for the audio signal to cross zero, and then writes the next 256 samples from the audio waveform into the waveform buffer. Make sure you only store audio samples when a new sample is provided, which happens when the new_sample_ready output from lab 4 is high. The waveform capture FSM has two states, armed and active. 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 next 256 audio samples into the BRAM - starting with the first positive sample. Think about the timing of your FSM and how to ensure that the FIRST positive sample is stored to the first address in the BRAM. (Hint: draw a timing diagram.) The capture FSM keeps track of the address it is writing to in an address register. This register is set to zero in the armed state. On each new audio sample, the audio sample is stored into the BRAM at the address indicated by the address register and the address register is incremented. When the address register wraps from 255 to 0 the capture FSM returns to the armed state and awaits the next zero crossing.

It is important to consider the format of the audio samples relative to the format of the Y values used on the display. Remember that the audio sample has adjacent values that jump from positive to negative 2's complement values. (This is how you are synchronizing to the start of the sine wave.) However, the VGA display deals with unsigned values. Therefore from the display's perspective the jump between +1 and -1 is huge since it considers +1 (9'b000000001) a small number and -1 (9'b111111111) a huge number. It is up to you to figure out how to account for this. There is a relatively simple way (one short line of code) of adjusting the audio sample before sending the data to be written to the BRAM. Consider what would be displayed if you did not adjust the audio samples before writing them to the BRAM. Then think about what operations you can perform on the audio sample to bring it closer to the expected display (where you have a sine wave displaying in the middle of the screen).

Hint: You want to make sure that both halves of the sine wave are connected together (to make a continuous wave), and that they are displayed in the center of the screen. Two simple operations on the audio sample should be sufficient to accomplish this (one for each improvement).

Wave Capture Interface:

Signal	Direction	Description
clk	input	clock
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
Write_address[7:0]	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 sample to write into the RAM.

Wave Display

If you recall from the VGA display diagram the X-axis does not extend much further than 1023 (maximum 10-bit number) and the Y-axis does not reach 511 (maximum 9-bit number). Therefore, to simplify matters in the wave_display module lets drop the XPos and YPos MSB and assume they are 10-bit and 9-bit numbers respectively.

Lab 5: Waveform Display

The waveform display logic accepts the X and Y positions from the VGA counters (100mhz_vga.v module) and produces RGB color output. It uses the X input to generate addresses for the BRAM and then computes RGB out based on the current Y input and the last two values READ from the BRAM. To convert the 10-bit X input to an 8-bit BRAM address we restrict our attention to the middle half of the screen (discarding the left and right quarters) and then drop the least significant bit of X (to ensure that the lines are fat enough). In order to have the X values from the middle of the screen map to contiguous 8-bit addresses we must drop the 2nd MSB and concatenate the MSB to the 3rd MSB. Below is an example that illustrates this point. To get the read_addr value you simply drop the Xin bits highlighted in red and then concatenate the other bits together.

2nd quarter of screen:	Xin = 10'b01xxxxxxx	read_addr = 8'b0xxxxxx
3rd quarter of screen:	Xin = 10'b10xxxxxxx	read_addr = 8'b1xxxxxx
Entire screen:	Yin = 9'bxxxxxxxxx	Y_effective = 8'bxxxxxxxx

Make sure to also drop the lowest bit of the 9-bit Yin value to get an 8-bit number that can be compared to the 8-bit data samples stored in the BRAM.

If this is not clear you should draw it out on graph paper and work out which parts of the screen this is selecting and how it maps to the BRAM address.

At each X position received from the VGA module we set RGB to 6'b111111 (white) if the most significant 8 bits of Y is in the range bounded by the last two samples read from the BRAM. Otherwise RGB is set to 0,0,0 (black). The last two samples read from the BRAM combined with the last two X values define a rectangle on-screen bounded on the left and right by the X position value and the top and bottom by the last two sample values. (This is key, so make sure you draw out a picture to understand how this rectangle is defined! Remember that when we cut-off the LSBs of the X-position we are making the X range of the rectangle bigger because there are more X-positions output from the VGA module that correspond to the same BRAM address.) However, more criteria must be imposed on the coloring scheme. You want to make sure that you only display waves corresponding to one full iteration through the BRAM. Otherwise you will get discontinuities at the boundaries of the BRAM since the last sample in the BRAM is not guaranteed to connect back to the first sample. To accomplish this you can simply restrict the X values for which color can be displayed to be within the range of the middle two quarters of our screen. **It is important to use all 11-bits of the XPos signal when specifying these boundaries so you don't get a fraction of the sine wave at the end of the screen.** This should be the only place where you need all 11 bits.

Remember that the BRAM has a one cycle latency for outputting the data and your Y bound is determined by the last two samples read from the BRAM. Think about which X-position coming from the display actually corresponds to the cycle when the OLDEST sample from the BRAM you're looking at is the sample from the FIRST address. This is the X-position where you will want to start displaying color on the screen. You can find the last X-position for displaying color in the same manner by considering when the NEWEST sample from the BRAM corresponds to the LAST address.

Wave Display Interface:

Signal	Direction	Description
clk	input	clock
reset	input	reset signal
XPos[10:0]	Input	The current X position of the VGA display
YPos[9:0]	Input	The current Y position of the VGA display
Vga_valid	Input	Wether or not the VGA coordinates are valid for displaying data.

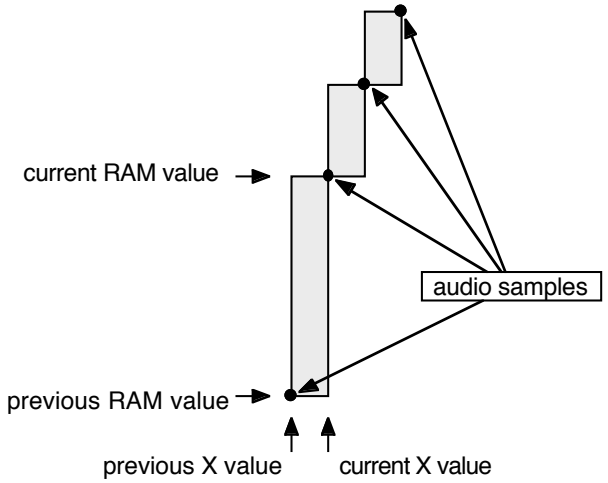
Read_address[7:0]	Output	The address in the RAM to read. Remember it takes one cycle to get the data back!
Read_value[7:0]	Input	The data you read back from the RAM.
Valid_pixel	Output	True if the pixel specified by XPos and YPos should be turned on.

Other things

It is up to you to determine the infrastructure of your modules so please use good style by writing readable and reasonably efficient code, and maintaining a good level of modularity. Also, don't forget to comment for your partner's and the graders' sakes (not to mention debugging). You will need to have a bunch of registers for storing things like the last two samples and some counters for determining RAM addresses along with your FSMs and some math logic. Make sure you have a good block diagram before you start writing your code.

For this lab you will need to be careful with your timing. The provided lab 5 top level module latches the output from your wave_display_top module to try and reduce the risk of timing errors, but if your syntheses report tells you that you are not meeting timing (i.e., your maximum clock speed is > 10ns or <100MHz) you **must** go in and add flip flops to your design to break up the critical timing paths. You can find your critical paths by running the Generate Post Place & Route Timing Analysis process and looking at the list of paths provided. You'll have to figure out what the exact route is by looking at the signal names and figuring out where they are in your design because the tools put in a lot of randomly named intermediate signals. (I.e., wave_fsm<0> to nsl_3_2 to wave_adder<3> indicates you have a path from the 0th bit of wave_fsm to the 3rd bit of wave_adder, and that the intermediate signal nsl_3_2 is between them.)

Remember that the image you are displaying is made up of rectangles. These are defined by the last X value from the VGA and the Y value read from the RAM for that X value, and the current X value and the current Y value read from the RAM for that X value. When you are displaying your image onscreen, keep in mind that sometimes the first Y point is large than the second, and sometimes it's the other way around. If you don't handle this you will only see a waveform when the wave is going up or down, but not both. Take a look at the diagram below and make sure you understand the two cases (only one is shown here).



Please be aware of how fast each module should update its signals. The wave_capture module deals with audio samples that update at 48KHz, which is why you need to use the new_sample_ready signal from the music_player to know when to update values in your module. On the other hand the VGA display changes its X-position at 50MHz, and the FPGA's main clock runs at 100MHz. Additionally, we drop the LSB of the X-position when generating the read address signal for the BRAM. Therefore the read address signal only changes once every four clock cycles (at 100MHz). You will need to account for this so that you're not interpreting the

same data sample from the BRAM 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 BRAM. Nowhere in your design should you be dividing clocks. All your logic should run off the same 100MHz clock and you should use enables in FSMs to deal with slower clocks.

Debugging this lab is a bit harder since there is no good way to simulate the VGA. Therefore you will have access to the lab to work on this assignment. However, the more careful you are with your testbenches the less time you will spend staring at displays that don't work.

For fun: you've got 6 bits of color, which means your VGA display can show 64 different colors. It's pretty easy to have the color change based on what you're displaying (just turn some 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 may not lock onto your output.

What you need to do…

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 wave_display showing the delay in reading from the RAM.*
4. Implement the wave_display, wave_capture, and wave_display_top modules.*
5. Write test benches for all of the above, and include the output in your report.* (You can't test everything here because the VGA is complicated, but you should try to do as much testing in ModelSim as you can before you start running through Xilinx because it is so much faster.)
6. Get it working on the VGA monitor in lab.

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