Lab 3: VGA

Introduction

By the end of this lab, you'll have applied your knowledge of sequential logic to create your very own VGA protocol! Similar to lab 2, you'll need to dig up those clock dividers and FSMs to follow the VGA timing specifications, then write some combinational logic to display some pixels on your screen. If you run into any issues, consult the resources linked at the bottom of this spec, and please don't hesitate to find your leads in the lab, or reach out to them on Discord!

I. VGA

Very Good, Adrian

Take some time to look through (or download) the code skeleton we've provided here.

The VGA driver is composed mostly of a few counters and some combinational logic to drive HSYNC and VSYNC. You'll also write some extra combinational logic to determine what is displayed for each pixel.

In this first section, your task is to fill out the code skeleton according to the VGA spec. These steps should be relatively simple, and we'll outline them for you in the following subsections.

1. VGA Constants

For the purposes of DAV, you'll be interfacing a 640x480 VGA display. There are eight constants that are defined, specific to this display size. **Fill out the correct constants under TODO(1).** You can find these numbers on the internet by searching for "VGA timing spec", looking through our lecture slides, or understanding the figure above.

2. Counting Problem

The pixel counters (hc and vc) dictate the logic in the VGA driver and are updated in the sequential block triggered on the edge of the VGA clock. If the reset button is pressed, you need to reset these counters to zero. Otherwise, you should increment the appropriate counter. **Write this logic under TODO(2).**

The logic for counter increments is as follows: increment hc until it reaches the end of a line of pixels, then increment vc and set hc back to zero. When vc reaches the end of a frame, wait until hc reaches the end of its line, then set hc and vc to zero. You'll need to use the constants you defined earlier to figure out at what value of hc you need to move to the next line, and at what value of vc you need to move to the start of the frame. Keep in mind that hc and vc are zero-indexed when writing your conditions for incrementing them! If hc equals 800 at any point, you did something wrong :-)

3. Let that sync in

Next, you'll need to drive the HSYNC and VSYNC signals that the display uses to synchronize its timings. This should be done in a combinational block using the hc and vc values; **write this logic under TODO(3).**

Look back at the constants defined in subsection 1 to determine the conditions for driving HSYNC and VSYNC high. Note that, for a majority of the time, HSYNC and VSYNC should be driven high, and are only pulled low during the blanking periods. Check this subsection specifically for off-by-one errors!

4. Business major homework (coloring within the lines)

Finally, the VGA expects some pixel outputs in the active display area. **Assign the red, green, and blue color outputs under TODO(4).** During the blanking period, assign these outputs to zero; otherwise, assign them to input colors: input_red, input_green, and input_blue. Remember that the input colors are 8-bit and the output colors are 12-bit; you'll need to perform some left shifts for your color outputs!

II. From the top

Make it drop

After filling out your VGA skeleton, **create a top level** to drive it. This will be a separate module.

Your top level should do at least the following:

1. Create and instantiate a 25 MHz clock using a PLL (more on this below)
2. Instantiate the VGA module with correct inputs and outputs (remember to use a shift register for the reset signal)
3. Pin planning!

1. DLC: PLLs

PLLs, or Phase-Locked Loops, are another way of generating clock signals aside from the clock dividers we introduced last quarter. While you don't need to know how they work (go ask the WRAP HW lead), do know that PLLs are a way to generate a more precise clock signal than what your typical clock divider can accomplish, and as such, are preferred when working with faster clocks and tighter timings.

To generate a PLL in Quartus, search for "ALTPLL" in the IP Catalog (found under Tools > IP Catalog) and launch the wizard. After entering a filename (something along the lines of "vgaclk"), enter the input clock frequency (50 MHz) on the first page. Hit "next" until it asks you to specify the output clock c0. Select "Enter output clock frequency", enter 25 MHz as the frequency, and hit "Finish". This will generate a module that you can instantiate in your top level to create your clock signal.

2. Please work

Once you have a top module, feel free to **test your VGA driver on an FPGA**! You don't really *need* a testbench here, as it'll be very clear whether or not your driver is working.

III. Graphics & Blocking

Pixel perfect

Here, you will create a graphics driver to determine how things show up on your screen. In essence, the graphics driver is a module that outputs the color of a certain pixel address it receives as input. Eventually, for your capstone, the graphics driver will also need to process some logic, possibly relating to your game state or FFT outputs.

Your graphics module should **take in the horizontal and vertical position** from your VGA driver (hc_out and vc_out), and **output the pixel address** calculated from those counters **alongside the RGB color** corresponding to that pixel address.

To calculate the pixel address, we can combinationally calculate [(vertical position) * (pixels per line) + (horizontal position)] anytime the VGA is within the active display area, and set the address to some don't-care value when the VGA is in a blanking period.

Due to space limitations and for your own sanity, you'll want to implement **blocking** in this module as well! Define some signals to be scaled-down versions (use a factor of 20 for this lab) of the horizontal and vertical position, and use these for your address calculation instead. We are effectively reducing the resolution of our display, and you'll need to change the "pixels per line" in your address calculation accordingly.

```
// for 20 by 20 pixels
assign xpos = hc / 20;
assign ypos = vc / 20;
```

Once you've finished your graphics module, remember to instantiate it in your top level and connect the ports. If you've done everything correctly, you should be able to paste the code from this file into your graphics module and see a familiar character on screen!

IV. Ping, Pong!

Li Shang introducing Mulan to Atari's 1972 tennis-like video game

In lecture, we described the difficulties of preventing screen tearing. To fix this, a double buffer is used to store pixels such that no buffer is being read from and written to at the same time. An effective ping-pong RAM requires asynchronous RAMs, which goes beyond the scope of what we'll cover in DAV (feel free to research on your own!). For practice, you will implement a double buffer in this section as a placeholder for ping-pong RAM.

To implement this, it will be easier to first create the RAM as **two arrays of registers**, with **one byte for each blocked pixel** on the screen. Once you have this working, you can replace the arrays with RAM IPs (found in the same IP catalog as the PLLs).

This module should accept the following inputs:

- A clock signal, synchronizing the read and write of your buffers to the VGA
- The hc and vc from the VGA driver, to calculate the RAM address to read from
- The RAM address to write at, which is output by the graphics controller
- The pixel data (color) you want to write to the buffer at the specified address

Your module should output the pixel data from the read address. Note that the width of your RAM address will depend on how large your pixel buffer is, which in turn depends on your blocking factor.

Instantiate a 1-bit register to determine which buffer is selected to be read from to control the RAM switching. Then, when hc and vc reach zero, flip the register to read from the other buffer. Make sure that this RAM is flipped sequentially, to keep everything synchronized. In a combinational block, set the addresses to read and write from the RAM based on which buffer is selected; this address will also be used to read data to the VGA driver. Finally, instantiate this module in the top level, and modify the port connections of your graphics driver to output to the ping-pong RAM.

Resources and Reference Material:

Lecture 1

Lecture 2

Lecture 3

Pin Sheet

Verilog Docs, FAQ

Bryan and Adrian