ES 4 Lab 7: driving a VGA monitor

1 Introduction

In this lab, you'll build a VGA display driver, which can display a 640x480-pixel image on a computer monitor or projector. Although VGA is finally being replaced by all-digital protocols such as DVI, HDMI, and DisplayPort, it is still a great starting place for several reasons:

- VGA runs relatively slowly (tens of megahertz), so it's easier to implement and easier to debug and analyze with the tools we have in the lab.
- At the same time, VGA finally pushes into the "sweet spot" for FPGAs. Even fast microcontrollers generally cannot drive VGA without special hardware to support it, but our little iCE40 FPGA doesn't even break a sweat: generating the output signals and displaying a test pattern requires fewer than 1% of the available logic blocks.
- DVI, HDMI, and other display protocols use a similar data layout and timing scheme, so understanding VGA is a good bridge to the newer protocols.

After successfully completing this lab, you should be able to:

- Describe the basic format of the VGA protocol
- Use the PLL hardware in the FPGA to generate (mostly) arbitrary frequencies

2 Prelab

The VGA protocol consists of 3 wires with analog voltages, one for each of red, green, and blue. Since the FPGA can only produce digital signals, each color channel uses two output pins connected with carefully-chosen resistors to create an analog voltage.

Pixels are transmitted a row at a time, starting with the top-left corner and scanning down to the bottom.

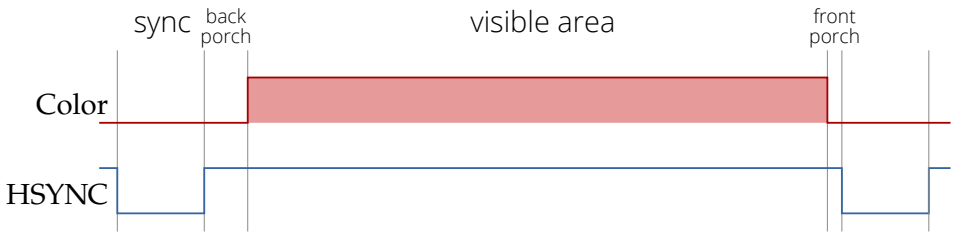
There is no shared clock, so timing information is transmitted with two additional signals:

- HSYNC (horizontal sync) pulses once per line, telling the screen to start a new row
- VSYNC (horizontal sync) pulses once per frame, telling the screen to start redrawing at the top

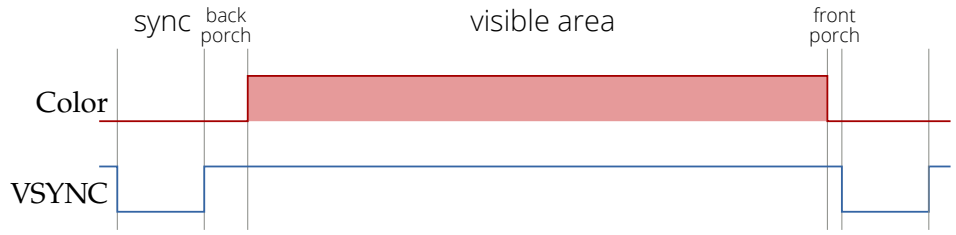
There is also some "downtime" between rows and between each frame, known as the "horizontal blanking" and "vertical blanking" periods. This blanking time is divided into the "front porch", "back porch", and sync pulse periods[1].

> **P1**: Look up the timing characteristics for VGA at 640×480 @ 60 Hz online. This page is a good source: http://www.tinyvga.com/vga-timing/640x480@60Hz
>
> For each segment of the VGA HSYNC signal shown below, write the duration in microseconds and in terms of the pixel count. That is, if you have a clock that runs at 1 pixel/clock cycle, how many clock cycles is each segment?

[1] To see where these names came from, take a look at this Wikipedia article: https://en.wikipedia.org/wiki/Analog_television#Structure_of_a_video_signal

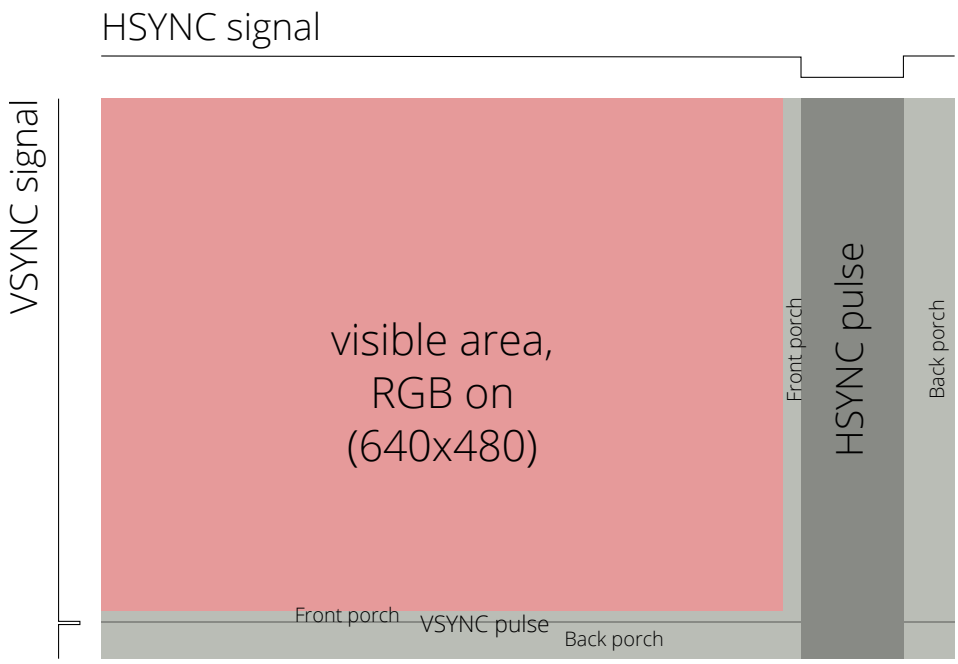


P2: For each segment of the VGA VSYNC signal shown below, write the duration in milliseconds or microseconds and in terms of the **line** count. That is, if you increment the line count each time one line is complete, how many counts is each segment?



The figure below may help make sense of how the HSYNC and VSYNC signals fit together. The display will scan across each row of pixels, with a blanking period between each line and a blanking period of several lines at the end of each frame. The HSYNC pulse will occur once per line, while the VSYNC pulse occurs once per frame.

Color data is driven on the R/G/B wires in the pink shaded region, which corresponds to actual pixels on the display. The color signals should be off during the gray-shaded blanking periods.



P3: Decide what kind of pattern you want to display. A simple option is to do something with a few bits of the horizontal and/or vertical coordinates.

See this article for an example of some really cool patterns you can draw with basic bit math: https://hackaday.com/2021/04/13/alien-art-drawn-with-surprisingly-simple-math/. Add a counter to the pattern generation module that counts up once per frame, and you can easily animate the patterns.

Another option is to create a scrolling starfield using an LFSR. You'll recall that an N-bit LFSR creates a pseudo-random sequence of N bit patterns, with a period of $2^N - 1$. For each pixel in an area with 2^N pixels, shift the LFSR and use the bits to decide whether the pixel should be on. When you repeat this for the next frame, all of the pixels will shift by one position, because the LFSR has a period one less than the number of pixels.

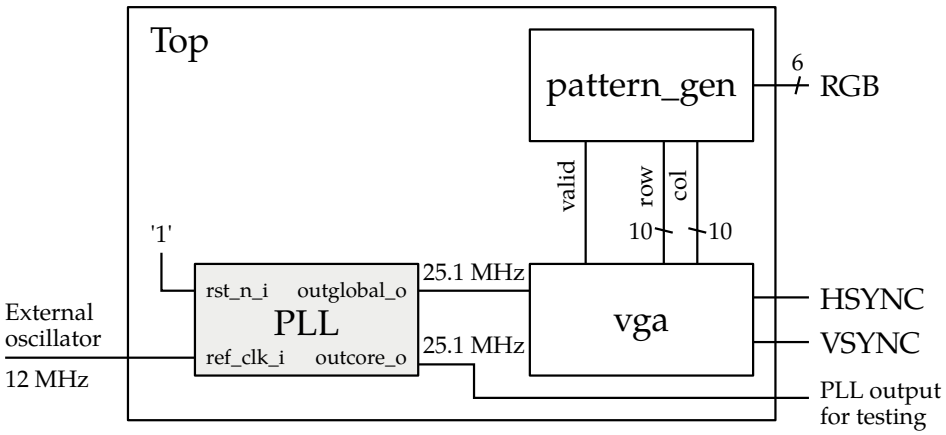
You can see an example of this at http://8bitworkshop.com/v3.3.0/?=&file=starfield.v&platform=verilog

For a 640x480 display, you could use a 512x512 pixel window and a 18-bit LFSR. Some of the display will be blank, and some of the starfield will overlap the blanking period, but that's ok.

While you don't need to write any code beforehand, writing at least a first iteration of your code will reduce your time in lab. You can simulate it in Modelsim or GHDL and check that your HSYNC and VSYNC signals look right.

3 In lab

The block diagram of the complete system is shown below. The PLL module already exists; your job will be to create the `vga` and `pattern_gen` modules, and wire everything together with the `top` module.



L1: First we need to generate the 25.175 MHz pixel clock. We could divide the 48 MHz HSOSC clock by 2 to get 24 MHz, but that's not close enough. Moreover, the internal HSOSC is quite jittery, which will produce fuzz and flicker on your display.

Instead, the UPduino actually has a much more stable oscillator onboard which produces a clean 12 MHz signal and a "PLL" which can convert one clock frequency to another,[a] and we can link these together to produce our pixel clock.

To add a PLL module:

1. Click on "IP Catalog" in the bottom-left corner of the Radiant window, below the list of files.
2. Double-click on "PLL". A wizard will pop up.
3. Select a name for the PLL module; something like "mypll" is fine. Click "Next".
4. Change the input frequency to 12 MHz (the frequency of the UPduino's oscillator).
5. Scroll down and set the output frequency to 25.27 MHz. The grayed-out boxes below labeled "Actual PLLOUT frequency" should change to 25.125 MHz.
6. Click "Generate" and then "Finish".
7. If you switch back to the File List, you can right-click on your newly-created PLL file and select "Copy VHDL component" to copy the component declaration to the clipboard. Just paste it into your `top` file. If this doesn't work, you can copy and paste the component declaration below.

[a] It does this by multiplying the clock frequency by an integer constant, and then dividing using a counter.

```
component mypll is
    port(
        ref_clk_i: in std_logic; -- Input clock
        rst_n_i: in std_logic; -- Reset (active low)
        outcore_o: out std_logic; -- Output to pins
        outglobal_o: out std_logic -- Output for clock network
    );
end component;
```

L2: Define a top-level input to your module for the clock, and and connect this to the PLL. On your breadboard, you'll need to connect a jumper wire between the "12M" pin and pin 20 (3 pins over). **If you have an UPduino 3.0 (without the sticker), note that the "12M" and "GND"**

labels are swapped - check the pin with an oscilloscope to confirm you got the right one! Alternatively, you can use a drop of solder to make the connection on the board, on the little square labeled "OSC".

L3: Connect the output of the PLL to an output pin so you can check that it really is 25 MHz. Note that you'll need to use the `outcore_o` output for routing to a pin; `outglobal_o` can only drive flip-flops. Build the design, and then use the Digital Discovery or an oscilloscope to check that the PLL output. *Be sure to capture appropriate results for your lab report!*

L4: Create your `vga` module, and set up counters representing the column count (incremented once on every pixel clock cycle) and the row count (incremented once every row).

L5: Use combinational logic to drive the HSYNC and VSYNC signals as a function of the row and column count, and the numbers you recorded in the prelab. We strongly recommend that you make '0' be the first pixel of the visible area and count from there.

You can use the Digital Discovery to verify that HSYNC and VSYNC are working correctly.

L6: Build your `pattern_gen` module which takes the screen coordinates and generates a 6-bit color to display. For example, you could make the left half of the screen white with the assignment:

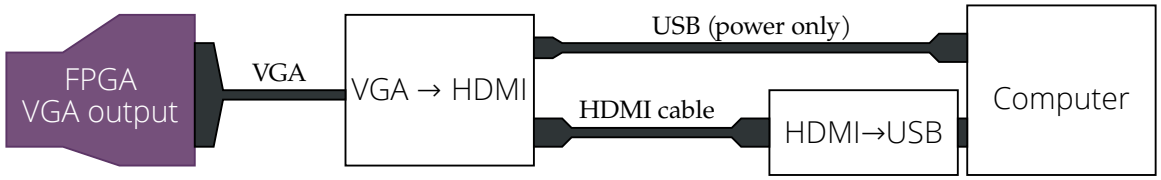
```
rgb <= "111111" when column < 320 else "000000";
```

If you take some bits of the row or column to use as the RGB value, you can easily generate bars of all the possible colors.

Make sure that the RGB signals are off (low) during the horizontal and vertical blanking periods. Some monitors assume the RGB signals are 0 and use this time to calibrate the signal levels, so displaying signals during this time can cause them to miscalibrate or completely ignore your colors signals.

L7: Wire up one of the VGA breakout boards to your FPGA. Depending on how you map your pins, you can plug the breakout directly next to the FPGA, without the need for any intermediate wires.

If you're in the lab, we have VGA → HDMI converters, HDMI cables, and HDMI → USB capture dongles[a] which you can connect as shown below. Don't forget the USB power for the VGA → HDMI converter.



The video feed should appear as a webcam or video input device on your computer, and you can view it with VLC, OBS, or any other webcam tool.

If you have access to a VGA monitor, you can simply plug the monitor into the VGA breakout and set it to VGA input (instead of HDMI or DVI).

If everything is right, you should see your pattern on the screen!

[a] This whole setup was approximately 1/10th of the cost of a VGA → USB capture device. Go figure.

4 What to turn in

For this lab, you do not need to write up and turn in a full report; we want you to focus on the final project. Instead, please turn in an abbreviated report with the following:

- Standard "front matter" (see the lab reports handout).
- Lab journal pages (you *are* still taking notes, right?)
- Your VHDL code.