Recitation 8

I/O Interfacing: VGA

Introduction

Students will be introduced to the VGA interfaces and controllers in this recitation. Understanding the interaction between the interfaces and controllers is key to understanding how to utilize different functional modules. Students will study the VGA controller and practice loading .mif files into their project directory. Students will alter the code so that the slide switches on the FPGA will be used to control the memory location holding the information for an image to be displayed.

Collaboration Policy

You will be working in groups of 2-3. Groups are allowed to collaborate.

Equipment

- Computer with Quartus Prime software
- DE2 FPGA board
- VGA Monitor
- DE2 board manual found under Sakai → Resources
- FPGA Blast Tutorial found under Sakai → Resources
- Quatus project file for FPGA found under Sakai → Resources->reci8.zip

Tasks

To receive credit for this recitation, you must complete:

- ❏ Task 1: Understand the VGA architecture and display different images on a VGA screen through the FPGA board

You must complete all parts of this recitation to receive credit. A TA must sign-off on the completion of each task. Ensure that the TA marks the completion of the tasks in Sakai.

Grading

- Completing Recitation Tasks: 1 point (pass/fail)

Understanding the VGA

In this part of the recitation, you will re-use the QAR file from the last recitation to implement the VGA controller. This recitation will help you to begin to get an understanding of how the VGA controller interfaces with the screen.

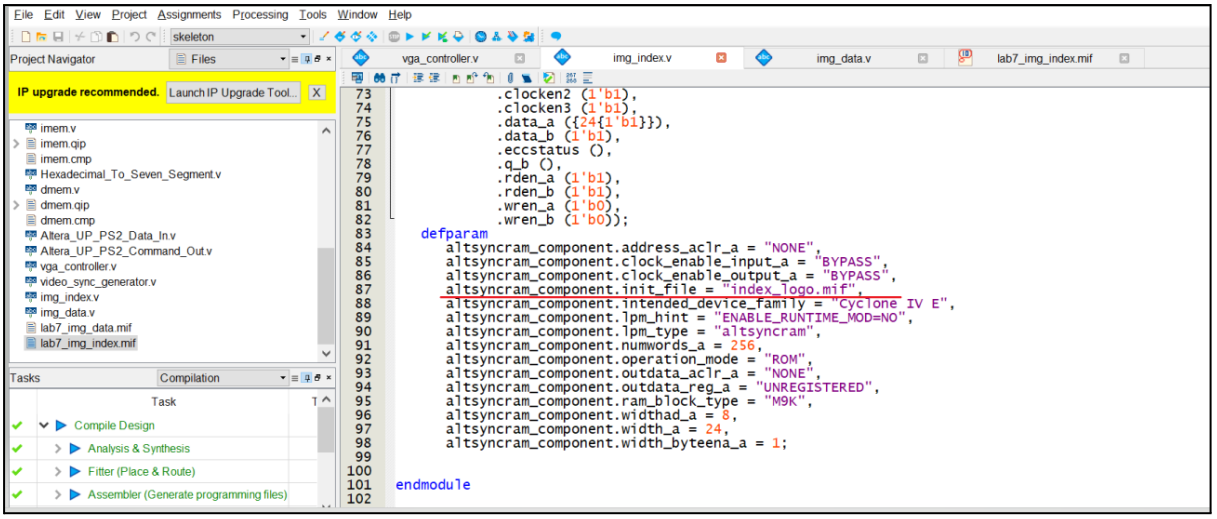
To start, open the “vga_controller.v” module. In this module, there is a section where Page 2 of 3 instances of img_data.v and img_index.v are called and connected to the signals in the VGA controller. It’s important to understand what they are doing.

The main idea behind how VGA works is that for each pixel on the screen, there is a 3- byte code that designates the values for red, blue, and green that the pixel will have. One byte for red, one for blue, one for green. The controller controls the image on the screen by updating one pixel at a time, doing all of the pixels in a given row on the screen, moving down to the next row, and similarly updating the color at each pixel. Once the controller gets to the last pixel on the screen, it moves back to the pixel at the beginning of the screen and starts the process over.

The color that a pixel takes on is the additive result of the red, blue, and green color values (e.g. white is 0xFFFFFF, black is 0x000000). In the controller, we have two modules, one that controls what pixel location we are at (img_index.v) and one that controls what the pixel value will be at that location (img_data.v). These modules interact with specified .mif files which hold the information used to display an image; This is useful because you can actually control the contents in a .mif file either by entering in the information manually or by using the load/store instructions of the processor ISA you’ve implemented previously.

Do a full compilation if you haven’t already and connect the VGA to the switch at your station. The image on the screen should be of the Altera DE2-115 logo.

Open up the img_data.v and img_index.v modules and find the lines that designate the .mif files that contain the information.

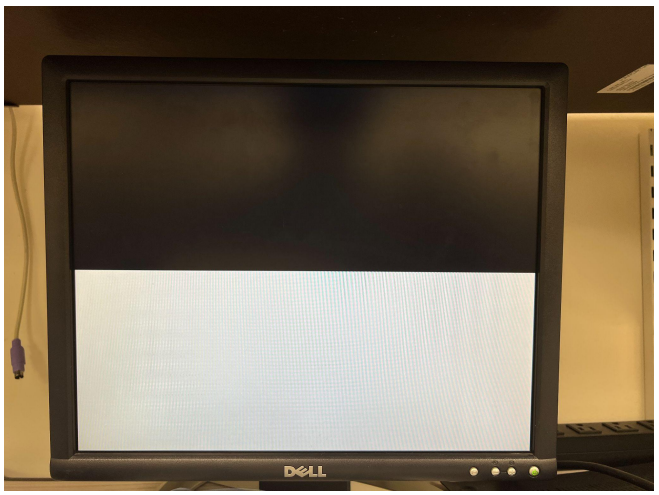


In the .mif file the img_data module calls, the mapping of a pixel address occurs (that is, for every byte in the .mif file, there is a 1-byte piece of information that will be used to locate the pixel color information in another .mif file). Notice how there are 0x4AFFF total memory locations; in decimal, this is 307,200. For a 640x480 sized screen, this means that each pixel is mapped on a 1:1 basis (do the math if this doesn't make sense at first). The img_index module maps to the .mif file that contains the pixel color info. Notice that in this file, there are only 0x0FF locations, each holding 3-bytes worth of information. These are the colors that the pixels will have.

After looking through these files, the manner in which the image on the screen is generated should start to make sense. Talk to your TA in case you need a little more help to understand these concepts!

Next, you will control the image being displayed on the screen by modifying the files provided for you. Using the **reci8_img_data.mif** and **reci8_img_index.mif** files, make the VGA display 2 different solid colors (one on the top half of the screen and one on the bottom half). To make the process of editing the contents of the .mif cells, go to Edit-->Custom Fill Cells to edit cell contents. You can choose your own color but the result should look like the following picture.

Hint: Change .mif file in img_data.v to reci8_img_data.mif and .mif file in img_index.v to reci8_img_index.mif.



Creating a moving square

For the last VGA task, you will be rendering a square image over the two-colored background you have created, and allowing it to move in all 4 directions on the screen through button inputs. You can use as much behavioral verilog as needed to complete this. The following is an example of how you could do this, not necessarily what you will do for your own work.

First, you must choose a reference point on the square in order to keep track of its location (e.g., top left corner, middle corner). Knowing this, create registers in behavioral Verilog to store the x and y coordinates of this position on the square.

The next step to complete is movement. Create 4 input signals for each direction of movement. As skeleton is the top level module, you will have to add these inputs to skeleton.v as well (don't forget to add them to the instantiation of vga_controller). Using an always block, update the x and y coordinate registers based on the input signals. Consider how frequently this `always` block should latch new values, and how this will affect the speed at which your square moves across the screen. A counter will likely be needed (look at the address generator for reference). Remember, you can use behavioral Verilog such as "==" to make sure the always block only updates when the counter is at a specific value.

Finally, you will need to create a multiplexer that chooses between displaying the background image or the square for any given value of "ADDR". In order to easily do so, create a module that converts the linear pixel address into x and y coordinates. Hint: addressing begins in the top left corner of the screen and goes across then down. For example, address 0 is the top left pixel, and address 640 is directly below. Using different comparison symbols such as "<", as well as addition and subtraction,

find a way to choose whether the background is displayed at a specific address given its x and y coordinates and the x and y coordinates of the square. If you get stuck anywhere in this process, make sure to ask your TA for help. Understanding this process is very critical. **Show your work to a TA after you finish this section.**