ECEN3002 – VGA Controller, Part 4B

Spring, 2021

Complete by Monday March 29

Now that you are expert in use of FPGA memory and the concept of frame buffers, this lab will ask you to enhance your video controller pixel generator, by adding a larger video memory to your system. The goal of this lab is to display images on your monitor.

The project will be developed using a crude 4x4 pixel size, in order to keep Quartus compile times reasonable. As you complete the design, keep in mind what will be required to convert your final project to 2x2 pixels using 4 bits/color.

You will submit a .qar file of your finalized 2x2 pixel design (4 bits each RGB), along with a photo of your display with a displayed image of your choice. If you went wild and created a script to display multiple images, feel free to submit a short video.

Part 1: Create your new frame buffer

In a 1280 x 720 display (720p), there are 921,600 unique pixel locations. The DE10_Standard board supports 24 bits (or 3 bytes) of color data per pixel, requiring 2.77 Mbytes of memory for a full pixel and color resolution display. This is more memory that is available in the CycloneV FPGA, so we cannot create a full pixel resolution, full color resolution frame buffer using the embedded memory blocks.

However, we can create a frame buffer with enough resolution to display identifiable images on our monitors. To do this, we will create a frame buffer that displays each memory location 16 times, in a 4x4 square. We will maintain the color resolution at 24 bits per pixel (8 bits each RGB). The resulting frame buffer size will be 320 x 180 x 3 bytes/pixel, or 57,600 words of 24 bit wide memory.

Note: The CycloneV device on the DE10-Standard board contains 553 M10K memory blocks, or approximately 550Kbytes of memory. The 4x4 pixel design will use 172,800 Kbytes (about 31 percent of the memory). The 2x2 pixel design (with 4 bits per color) will use twice as much memory, or about 62%.

By using the 4x4 pixel approach, the Quartus compile time shouldn't be too bad.

1) Create another version of your pixel generator module for this assignment. You will use 720p resolution.

2) You will need a memory initialization file (.mif) for this project. We will construct our .mif files as part of this lab, but when you generate your memory block, you can use an empty file.

3) With your project open, add a ROM from the IP catalog with the following characteristics:

  a. 1 port ROM
  b. 57600 deep x 24 bits wide
  c. single clock
  d. do not register the output port
  e. include the memory content file
  f. Allow In-System Memory Content Editor

Note that when you browse for the .mif file in the ROM dialog, by default the selector is looking for a .hex file. Just change the dropdown for the extension to .mif to locate your file. Why the Quartus GUI people didn't default to look for both is one of life's great mysteries.

4) Create a new version of the pixel generator file, and instantiate the ROM.

5) You will need to generate an address to the memory based on your horizontal and vertical counters. To create the 4x4 pixel, you will not use the lowest order two bits of either address counter.

6) You can test your design during development by creating a simple .mif file, or by using the In-System Memory Content Editor as before.

Part 2: Updating just the .mif file contents

If all you want to do is use different .mif file contents in your design, but not change the hardware, there are several ways to do this. The most time consuming is to rerun an entire place and route, and this is not necessary. The name of the .mif file you are using is linked to the ROM memory block you created, so it is easier to keep the name of the .mif file the same.

The .mif file contents do not have any effect on the synthesized logic or the placement of the logic. After the Quartus fitter has run, the Quartus assembler step reads the .mif file contents and merges it with the .sof file. Each embedded memory bit has a corresponding value in the configuration file, and the .mif file contents are simply added to the .sof file.

Methods to use modified .mif file contents

1. In Quartus, run Processing > Update Memory Initialization File, then rerun just the programming file generation step, Processing > Start > Run Assembler.

2. If you want to create a script instead of using the GUI, run these two commands in the Quartus Tcl console:

    a. quartus_cdb –update_mif <project name>
    b. quartus_asm <project name>

3. Use the In-System Memory Content Editor
    a. Edit > Import Data From File
    b. Note that this approach is only temporary, and does not update the configuration file, but is a useful approach for testing. You could also Tcl script this and update your display with different images at a desired interval.

If you do want to change the .mif file that is associated with the ROM block, you can do this by (carefully) modifying the .v file that is generated with the ROM IP. In the .v file, look for a line containing **altsyncram_component.init_file** and modify the filename. I do not know how this might affect simulation of the ROM component, so do this at your own risk.

Part 3: Creating a .mif file

I am certain you could do some very impressive and extensive image manipulation in Matlab, but for this class we will use Matlab to break out the RGB values of an image, and construct the resulting .mif file. The resizing (downsizing) of any image results in a loss of clarity. Use of more sophisticated image processing tools would help.

1. Start with an image you want to display. Using your photo tool of choice (Paint is a simple tool in Windows), resize the image to be 320 pixels wide.

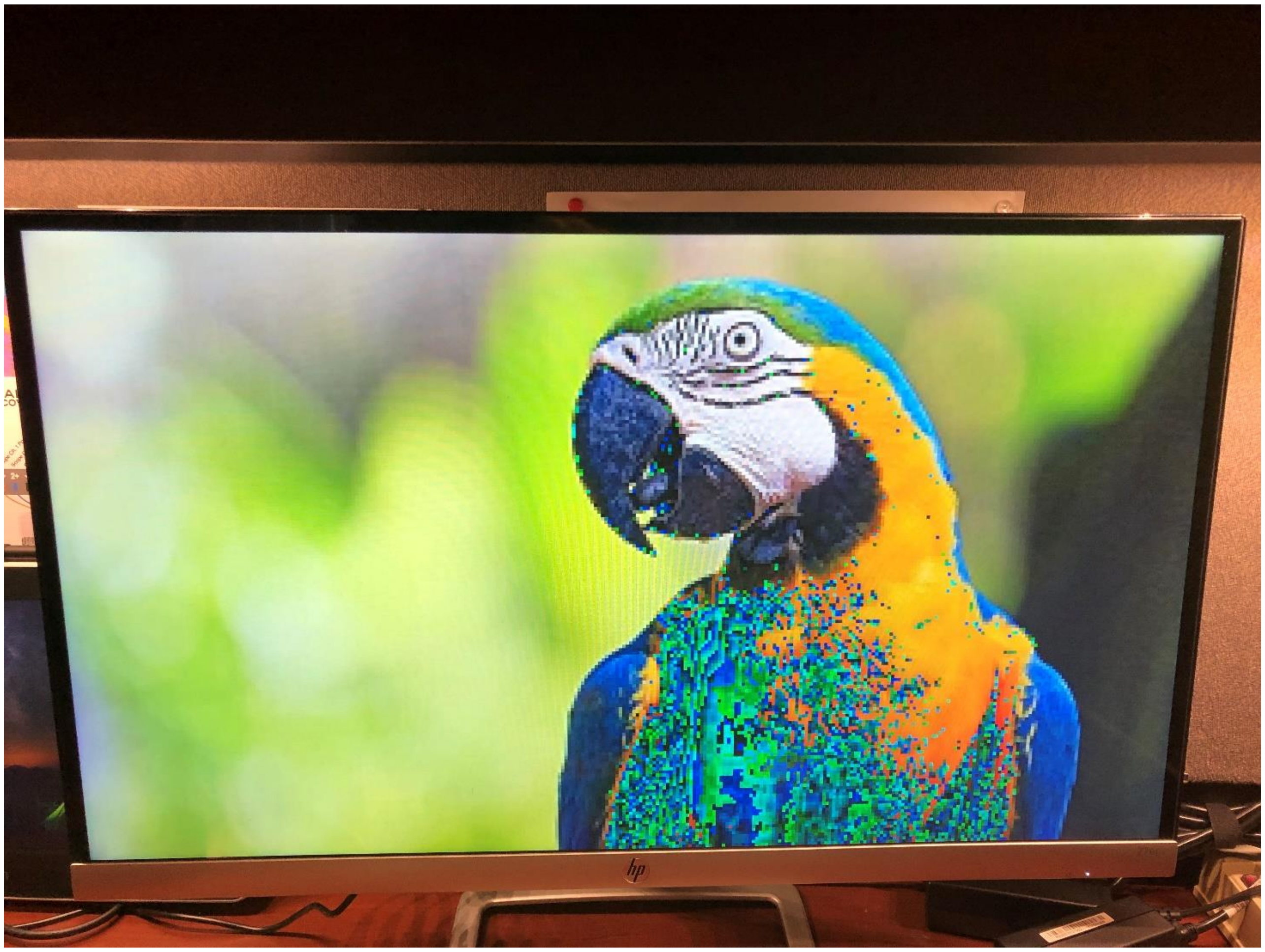
2. Process the image in Matlab to generate your .mif file. The Matlab script is available in Appendix A, and also as a file in the lab area.

Note: The Matlab script as written will work for the 4x4 pixel approach. You will need to make minor modifications to the script to work for the 2x2 version of your project.

I will provide a .mif file of a parrot to assist with your project testing, but for your submission create an image of your choosing.

Results

I found a .jpf image of a parrot, and decided to use that. First the image was read into Microsoft Paint and the resolution was reduced to 320 pixels wide. The vertical dimension was left at whatever value resulted by maintaining the aspect ratio. This image was then processed in Matlab to create the .mif file, which was then displayed in 720p using the 4x4 pixels from this project. You can see the obvious pixelation, but the image is easily recognizable.



Appendix A:

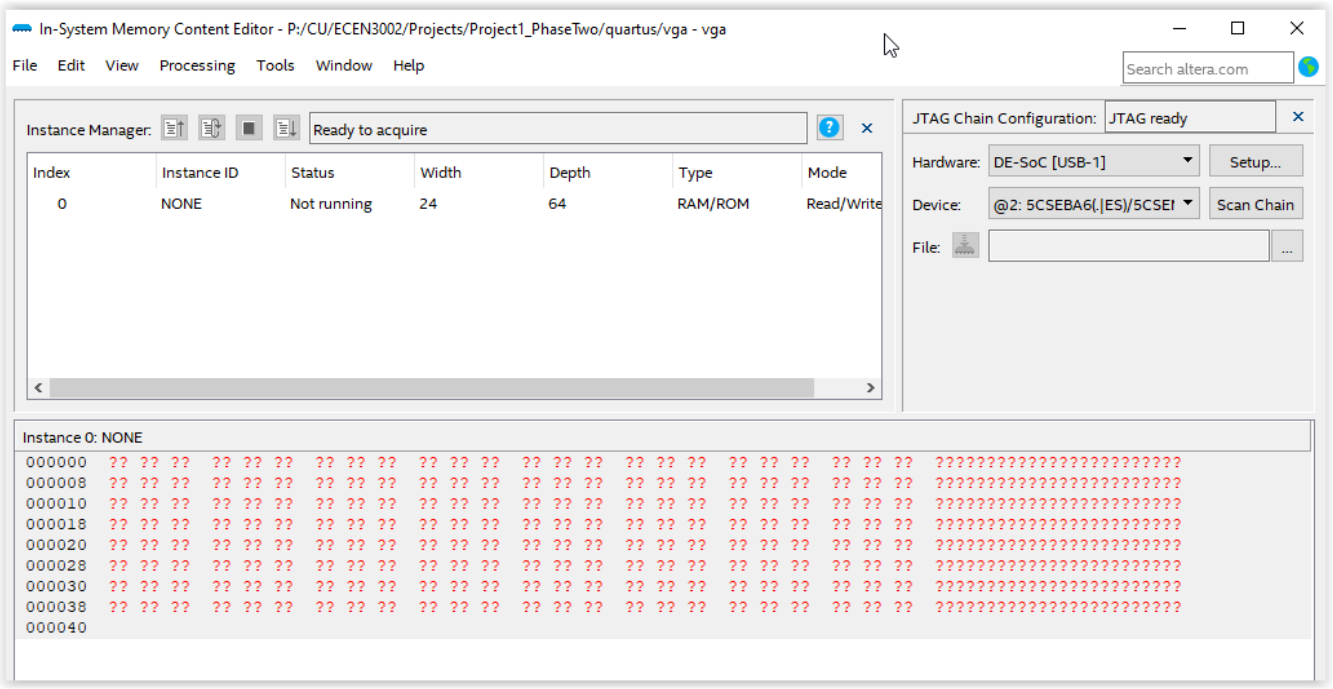
This Matlab script is not very sophisticated, no doubt your could write a better version (if you do, please pass it along to me).

The script reads in an image based on the `depth` and `wordlen` values, and separates out the red, green, and blue color values. The script then constructs a .mif file with the appropriate headers, and writes the RGB data into the .mif file.

```
% Assumes picture is 320 pixels wide, 8 bits each RGB
% Framebuffer is 57600 24 bit words
pic = imread('<your_file_here.jpg_or_bmp');
depth = 57600;
wordlen = 24;
red   = pic(:,:,1);
red   = red';
green = pic(:,:,2);
green = green';
blue  = pic(:,:,3);
blue = blue';

% Write header info
fid = fopen('<your_name_here.mif', 'w');
fprintf(fid, 'DEPTH=%d;\n', depth);
fprintf(fid, 'WIDTH=%d;\n', wordlen);
fprintf(fid, 'ADDRESS_RADIX = DEC;\n');
fprintf(fid, 'DATA_RADIX = HEX;\n');
fprintf(fid, 'CONTENT\n');
fprintf(fid, 'BEGIN\n\n');

% Write RGB data
for i = 0 : depth - 1
    fprintf(fid, '%d\t:\t%x%x%x;\n', i, red(i+1), green(i+1),
blue(i+1));
end

fprintf(fid, 'END;\n');
fclose(fid);
```

Appendix B: In-System Memory Content Editor

1. Open the memory content editor (Tools > In-System Memory Content Editor. In the JTAG Chain Configuration area, select the cable and Device @2. Your display should look like this:



Ready to acquire

Index	Instance ID	Status	Width	Depth	Type	Mode
0	NONE	Not running	24	64	RAM/ROM	Read/Write

JTAG Chain Configuration: JTAG ready

Hardware: DE-SoC [USB-1]

Device: @2: 5CSEBA6(.|ES)/5CSEI

Instance 0: NONE

```
000000  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ????????????????????????
000008  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ????????????????????????
000010  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ????????????????????????
000018  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ????????????????????????
000020  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ????????????????????????
000028  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ????????????????????????
000030  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ????????????????????????
000038  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ?? ?? ??  ????????????????????????
000040
```

2. Highlight the line in the white display area, then read the memory contents. This can be done by click on the leftmost icon with the up arrow, Processing > Read Data From In-System Memory, or by pressing F5. Your memory contents should now be shown.

3. In the hex display region, enter new values for addresses 0 - 2, then update the memory by either pressing the icon with the down arrow or by pressing F7. Experiment with changing other values that result in the color bars changing color or intensity.

The In-System Memory Content Editor can be used to read or write memory in a design. One limitation of this approach is that any memory you use must be created as a single port memory, as the editor uses the second port to access the memory.

Appendix C: Scripting In-System Memory Content Editor

You knew this was coming, right? The good news is that I am not asking for you to write a script, but you could do some fun things with the display if you did.

1. Open a command prompt, then launch a Tcl console - quartus_stp.exe -s

2. Change directory to your project directory containing all your Quartus files.

3. You will open the project in the same manner as when you used the ISSP. These commands open the project, get the name of the download cable, and get the name of the FPGA device.

```
     project_open <your_project_name>
     set usb [lindex [get_hardware_names] 0 ]
     set device_name [lindex [get_device_names -hardware_name $usb] 1]
```

4. You enable memory access like this. Note that $usb was set above, as was $device_name. ;

```
begin_memory_edit -hardware_name $usb -device_name $device_name
```

Get an error? This occurs if the Memory Content Editor GUI is still open. Close the GUI and try again.

Get another error? Type this command:

```
end_memory_edit.
```

5. Try again.

```
begin_memory_edit -hardware_name $usb -device_name $device_name
read_content_from_memory -instance_index 0 -start_address 0 -word_count 10
```

You should should see the memory contents displayed in binary. You can change the values of start_address and word_count if you want. Note that the instance_index value is set to 0, as this is the only accessible memory block in the design. If you had multiple memory blocks, you would change the index value as needed.

To get a hex display, add this to your read_content_from_memory command:

```
-content_in_hex
```

6. Write to memory

```
write_content_to_memory -instance_index 0 -start_address 0 -word_count 1 -
content “FF0000” -content_in_hex
```

should turn the first vertical stripe red. Since the original ROM block was created to be 24 bits wide, each value of word_count assumes 24 bits.

Complete with

```
end_memory_edit
project_close
```