ECEN3002 – VGA Controller, Part 4A

Spring, 2021

Complete by Thursday March 11

Nothing to turn in (yet)

This lab will ask you to create a new video controller pixel generator, by adding a video memory to your system. You will create a checkboard pattern on your display by pulling pixel values from the video memory.

Next, you will use the Quartus In System Memory Content Editor (QISMCE) to interact with the display values, and of course, Tcl.

Part 1: Create your video memory

In a 1280 x 720 display, there are still 921,600 unique pixel locations, requiring 24 bits (or 3 bytes) of color data, for a total of 2.77 Mbytes of memory. This is more memory that is available in the CycloneV FPGA, so a smaller memory will be used. If we are going to create geometric patterns (checkboard), then the amount of memory required will be much less.

Also, very large memory blocks require a lot longer to compile in Quartus.

For this design, we will create a 64 location x 24 bit wide ROM. The ROM will be a single port ROM, but with the second port enabled for the QISMCE. If you had some external processor that could load the video memory, you would create the video memory as a RAM, and not a ROM.

1) Create another version of your pixel generator module for this assignment.

2) You will need a memory initialization file (.mif) for this project. Here is one I created you can use as a starting point. Note that the % sign is used for comments. You will need to have a .mif file when you create the ROM IP in part 3. The file can be empty at that point, but it must exist.

```
% DEPTH and WIDTH are decimal values %
% RADIX values can be BIN, DEC, or HEX %
% Address : Data %
% [A0..Ax] : D , address range 0 to x contain value D (each address gets this value %
% [A0..Ax] : D0 D1 , A0 gets D0, A1 gets D1, A2 gets D0, etc %
% A : D0 D1 D2 , address A gets D0, address+1 gets D1, etc %

DEPTH = 64;
WIDTH = 24;
ADDRESS_RADIX = DEC;
DATA_RADIX = HEX;

CONTENT
BEGIN
% Red, Green, Blue %
[0..7]   : ffffff;
[8..15]  : ff0000;
[16..23] : 00ff00;
[24..31] : 0000ff;
[32..39] : 204060;
[40..47] : 406020;
[48..55] : 602040;
[56..63] : 000000;
END;
```

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

  a. 1 port ROM
  b. 64 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

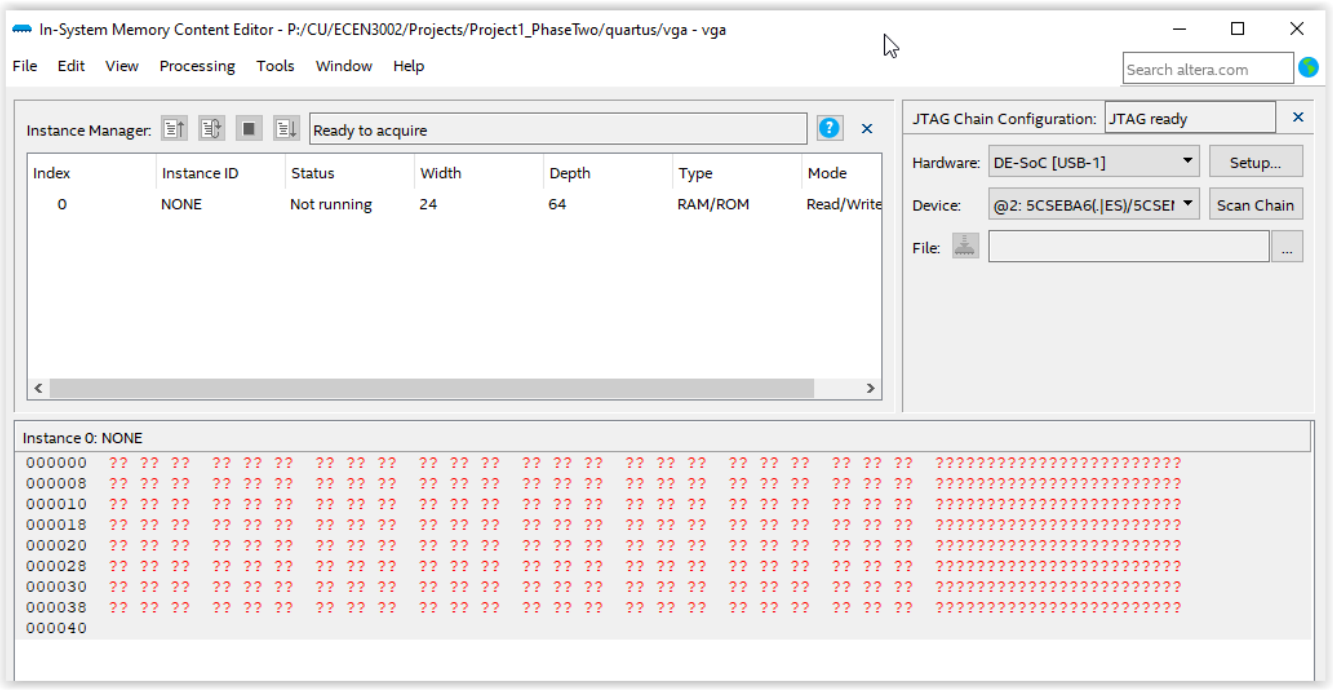
Note that when you browse for the .mif file in the ROM dialog, by default the selector is looking for a .hex file. Just change the dropdown for the extension to .mif to locate your file. Why the Quartus GUI people didn't default to look for both is one of life's great mysteries.

4) Create a new version of the pixel generator file, and instantiate the ROM. The ROM outputs will drive RGB, clock the ROM using your pixel clock. Set the address of the ROM to 0 for the first 160 pixels, address = 8 for the next 160 pixels, until you reach all 1280 pixels. Note: Make sure you are driving all 0s the the RGB values when outside visible display area.

5. Compile your design and verify that your display shows 8 vertical color bars.

Part 2: 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:



In-System Memory Content Editor - P:/CU/ECEN3002/Projects/Project1_PhaseTwo/quartus/vga - vga

Instance Manager: 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

File:

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.

Part 3: 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
```

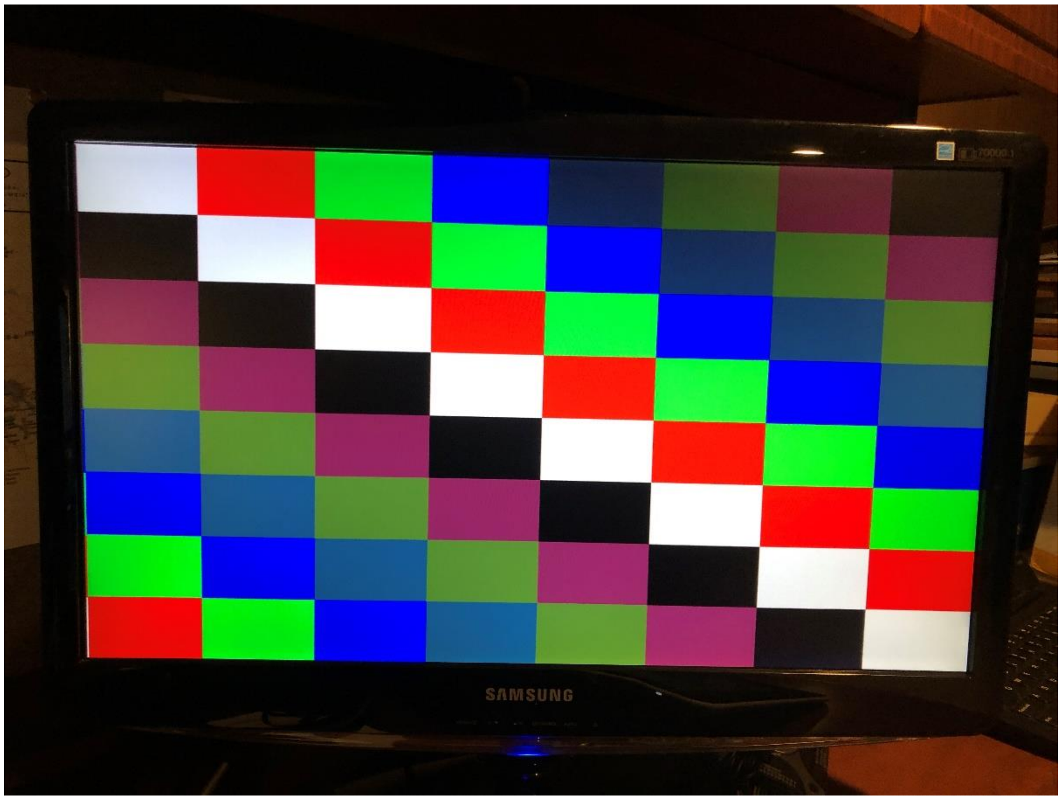
should turn the first vertical stripe red. Since the ROM block was created to be 24 bits wide, each value of word_count assumes 24 bits.

Complete Part 3 with

```
end_memory_edit
project_close
```

Part 4: Make the checkboard pattern

This is the dreaded “exercise left to the reader” section. Start with your pixel generator using the ROM, and modify the code to create a checkboard pattern of colors.



Make sure your output colors are correct (don’t drive video outside the visible area).

You may notice one pixel wide strips at the top and left side of your display. I think this is due to the input address register on the video ROM that cannot be disabled, resulting in the memory output being delayed one clock cycle.

5 extra points if you figure out how to get rid of this unwanted behavior (and tell me what you did).

We created a video memory that contains 64 locations, but we only used 8 locations in this lab. You can experiment with filling each location with a unique color and modifying your pixel generator to use all 64 addresses (if you didn't already). You can write a Tcl script that rewrites the memory contents in an infinite loop in order to some interesting effects. You can increase the memory size to get finer resolution control.

What you can't do with this approach is map each pixel to a unique memory location, which would be a fun thing to do. Later we will replace the memory block with the SDRAM on the DE10-Standard board and see if we can display a photo or some other image.

The number of things to try with a video controller design are almost infinite, and are always fun to try.