ECE 550 Fall 2015
Homework 2

Due: 11:59PM ~~Sept 25,~~ 2015 October 2nd
Demo by: 11:59PM ~~October 2nd~~, 2015 October 9th

For this homework, you will be answering two pencil and paper questions, as well as writing VHDL. You should submit your answers to the pencil and paper questions, along with your VHDL code in a .tar, .tar.gz, or .zip format (no other formats will be accepted) to Sakai before the deadline. Your answers to the pencil and paper questions **must** be in .pdf format—no Word documents will be accepted. For drawings, you are encouraged to draw in computerized tool, but may draw by hand and scan the drawings into a pdf format, as long as they are clear and easily readable.

Within one week of the submission deadline, you must demo your VHDL code to a TA, who will ask each group member to explain various aspects of how it works. **All** group members are responsible for understanding the entire submissions.

Question 1:

You want to make a vending machine, which accepts nickels (5 cents), and dimes (10 cents)—for simplicity of this problem, it does not accept quarters nor pennies. Some of the snacks which this vending machine dispense cost 25 cents, while others cost 40 cents. For this question, you will draw the finite state machine for your vending machine's control logic. The control logic takes the following inputs:

inDime The user put in a dime.

inNickel The user put in a nickel.

vend25 The user requested a 20 cent snack.

vend40 The user requested a 35 cent snack.

Your control logic should output the following signals:

outVend Dispense the requested snack.

outDime Give back a dime to the user.

outNickel Give back a nickel to the user.

Your vending machine controller must meet the following requirements:

- It only asserts outVend in response to vend25 or vend40 if sufficient money has already been paid to satisfy the request.
- After asserting outVend, it must return any money entered beyond the price to the user via outDime and/or outNickel (so if the user puts in 40 cents, and requests a 25 cent snack, 15 cents must be returned in change).
- Only one of outDime or outNickel may be asserted at a time.

You may rely on the following to simplify the problem slightly:

- At most one input will be asserted at a time.
- If the user enters more money than the maximum snack price, you may return some or all of the money over that price immediately.
- To keep your diagram simple, you can omit any self-loops on no input (that is, if you do not draw a self loop on a state, we will assume that on no inputs, you stay in that state).

For this question, you should submit a drawing of the state diagram, and a clear description of your output function.

Question 2:

Perform the binary division of 110111 divided by 101 (both are unsigned numbers). For each step (states are numbered by which bit of the divisor is examined) , show the contents of the remainder register and the answer register (both of which are 6 bit unsigned numbers):

State	Remainder	Answer
Start	000000	000000
5		
4		
3		
2		
1		
0		

Question 3:

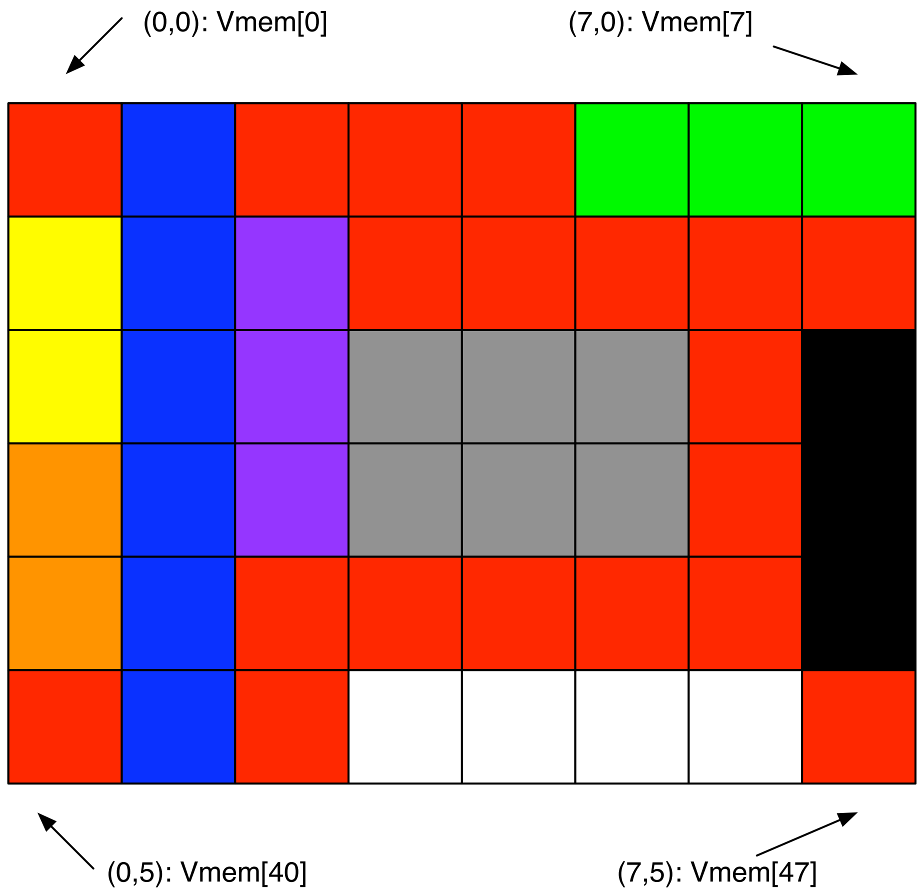
For this question, you will be writing VHDL for a "baby video card". The DE2 boards has pins which let you send digital color signals to a VGA connection (there is some digital to analog conversion involved, but you do not need to worry about that).

We have provided you with a quartus archive file (`vgacontroller_dist.qar`) which gives you the pin assignments, a clock at the proper frequency, and an active high reset signal (meaning you get a '1' for reset—by contrast the signals coming from the buttons on the board are active low, meaning they send a '0' when its pressed).

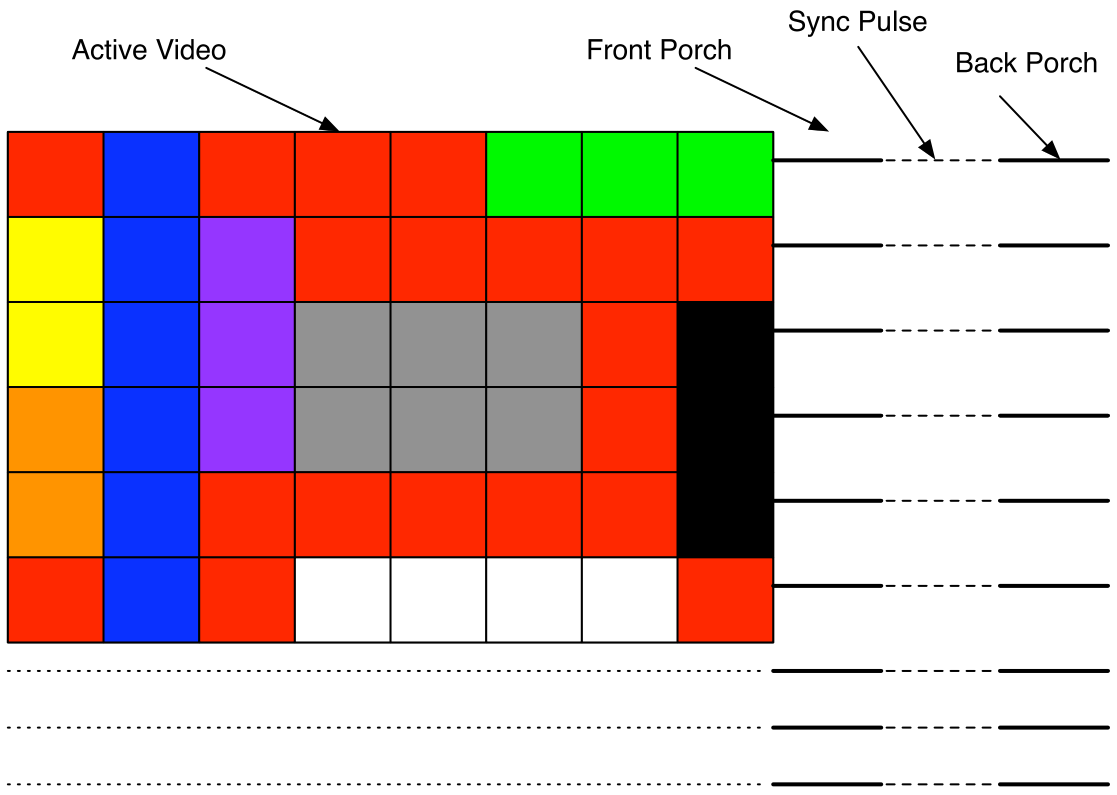
You should not need to edit any file other than `myvga.vhdl`, where you will fill in the `myvga` entity (currently it just has placeholders which do nothing useful).

Before we go into the details of the provided VHDL, it is useful to give you a description of how video cards work, and what needs to be done to properly control the VGA signals.

Video cards store pixel data as numeric values in memory (located on the video card). The pixels make up a 2D array of dots, with (0,0) in the upper-left corner of the screen. How many pixels there are depends on the screen resolution. You will be working with a resolution of 640x480 (so 640 pixels in the x direction (columns), and 480 in the y direction (rows)). Video memory is indexed linearly, the rows laid out one after the other—the index into the video memory for a pixel at (x, y) is $y * w + x$, where w is the width of the screen (in pixels). The drawing below illustrates for a resolution of 8x6:



To communicate the video data to the monitor, the video card sends out one pixel at a time (which then gets converted to an analog signal, and sent over the monitor cable). The pixels are sent out row by row from top to bottom. For each row, the pixels are sent out left to right. In between each row, the video card sends black pixel data (all 0s), and changes the hsync signal (this is how the monitor figures out the horizontal resolution). These black pixels come in three phases, illustrated below:



This makes four phases of each line, each of which have the following requirements:

Active Video Pixel data is sent out one pixel per clock cycle. The hsync signal is high. For 640x480 resolution, this lasts 640 clock cycles per row.

Front Porch Pixel data is black (all 0s). The hsync signal is high. For 640x480 resolution, this lasts for 16 clock cycles per row.

Sync Pulse Pixel data is black (all 0s). The hsync signal is low. For 640x480 resolution, this lasts for 96 clock cycles per row.

Back Porch Pixel data is black (all 0s). The hsync signal is high. For 640x480 resolution, this lasts for 48 clock cycles per row.

After scanning all 480 rows in this manner, the video card goes through a similar procedure to establish the vertical resolution. It scans out blank rows, and has a vertical front porch, vertical sync pulse, and vertical back porch. During each of the blank rows that it scans out, it goes through the same horizontal procedure as for a "real" row, except all of the pixel data during the "Active Video" phase is black (it still does all four phases). This makes for the following four vertical phases:

Vertical Active Video : The vsync signal is high. For 640x480 resolution, this lasts for 480 rows (as above).

Vertical Front Porch : The vsync signal is high. For 640x480 resolution, this lasts for 11 blank rows .

Vertical Sync Pulse : The vsync signal is low. For 640x480 resolution, this lasts for 2 rows.

Vertical Back Porch : The vsync signal is high. For 640x480 resolution this lasts for 31 rows.

After completing all 524 of these rows, the process begins again at the upper left.

Specifics for your video card implementation

A few specifics for your implementation of the video card.

I have provided you with a ROM (read only memory) with a test image in it (a real video card has a RAM, but we don't want to write something to dynamically draw new images here). This ROM is available to you in myvga via this component declaration:

```
component vmem is
  port (
   address: in std_logic_vector  (15 downto 0);
   clock  : in std_logic;
   q      : out std_logic_vector (8 downto 0));
end component;
```

Unfortunately, the DE2's FPA does not have enough room to hold a color image at 640x480, so the test image is stored at 80x480 in 9-bit color (by comparison, most monitors use 16-bit, or 24-bit color). This means that for each adjacent pixels, you will want to read one single ROM entry (e.g., 0–7 reads entry 0, 8–15 reads entry 1, and so forth). Effectively, you will want to read entry $\frac{N}{8}$ whenever you normally would read entry N. Fortunately, dividing by 8 does **not** require a divider, you can simply ignore the low three bits of your pixel index, e.g.,

```
...
address => pixel_index (18 downto 3),
 ...
```

The 9-bit value that you read out with have red in the high 3 bits, green in the next three bits, and then blue in the last 3 bits. The red, green, and blue signals that you output from the FPGA will have 10 bits. For each color channel, you will want to put them in the high 3 bits, and have the low 7 bits be 0s.

Here are specific descriptions of the input and output signals for the myvga entity:

clk This is your clock. It will be the right frequency for 640x480 from the logic outside what you write. You should use this to clock all your DFFs.

rst When this signal is high, it indicates that the reset button (Key) on the DE2 board. Before reset is asserted, the state of your controller can be undefined, and you can use the reset signal to initialize to known good values (e.g., all 0s, correct starting state, etc).

red This 10 bit signal should specify the red color channel. Since you only read 3 bits of red from the ROM, put them in the high three bits of the output signal, and 0 the other 7 bits.

green Similar to red, but for the green channel.

blue Similar to red, but for the blue channel.

blank This signal is active low. If it is '1', then the red,green,blue data will be used. If blank is '0', then the rgb data will be ignored and treated as 0. You can use this during the porches and sync pulses (but don't have to if all the red, green,and blue data is 0).

hsync The horizontal sync signal, which should be 0 during horizontal sync pulses, and 1 at all other times.

vsync The vertical sync signal, which should be 0 during vertical sync pulses, and 1 at all other times.