CSC 258 - Lab 7

Fall 2016

Memory and VGA Display

Learning Objectives

The purpose of this lab is to learn how to create and use on-chip Block Random Access Memories (BRAMs) as well as use the video graphics adapter (VGA).

Marking Scheme

Each lab is worth 4% of your final grade, but you will be graded out of 8 marks for this lab, as follows:

- Prelab - Simulations, Schematics and State Diagram: 3 marks
- Part I (in-lab): 2 marks
- Part II (in-lab): 3 marks
- Part III : Bonus 2 marks (1% overall)

Preparation Before the Lab

You are required to complete Parts I and II of the lab by writing and testing Verilog code and simulating it with ModelSim using reasonable test vectors. Show your state diagrams, schematics, Verilog, and simulations for Parts I and II to the teaching assistants.

In-lab Work

You are required to implement and test all of Parts I and II of the lab and demonstrate them to the TAs.

Part I

In addition to logic blocks and flip-flops, contemporary FPGA devices provide flexible embedded memory blocks that have configurable memory bit widths and depths along with many other parameters. To access these blocks you will use another feature of Quartus, which creates modules that serve as interfaces to embedded blocks in FPGA devices. The module built using this feature provides all inputs and outputs required to work with the specific embedded block, and can be instantiated in your design. In this part of the lab exercise you will create a small RAM block and interact with it to understand how it works. Using the Quartus *IP Catalog* you will first create a module for the desired memory and then test the memory module using the switches and hex displays for inputs and outputs.

The memory module we would like to create is shown in Figure 1. It consists of a memory block, an address register, a data register and a control register. You can see that the address, input data, and the write enable control signal are all stored in registers before being presented to the memory array. Using the registers means that the *DataOut* value will be stable for one clock cycle and allows the inputs to be changed after the rising clock edge in preparation for the next clock cycle. It is a small memory so that we can easily interact with it using the available switches and displays on the DE1-SoC board.

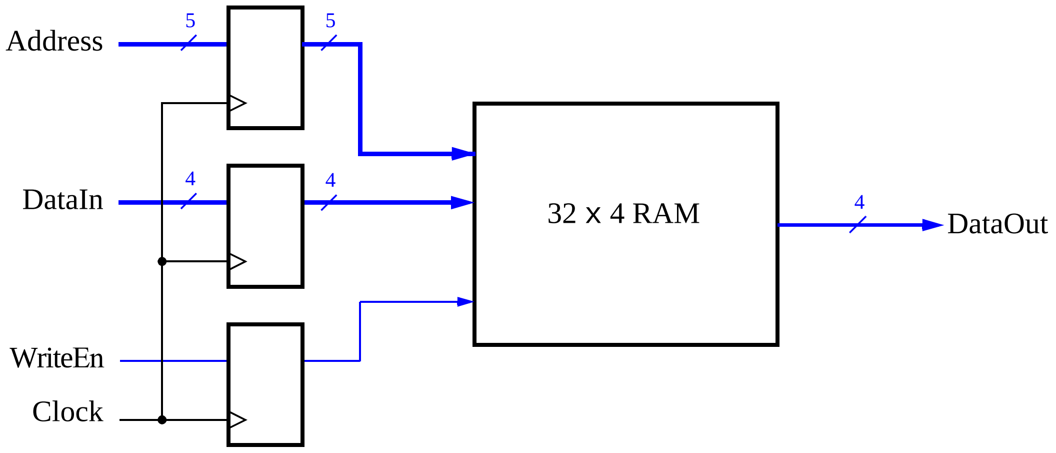

Figure 1: Schematic of the 32 $\times$ 4 embedded memory module.

A timing diagram showing reading of the memory is shown in Figure 2. Four locations at addresses *A0*, *A1*, *A2* and *A3* are accessed and the corresponding data *D0*, *D1*, *D2* and *D3* are read from those addresses, respectively. Figure 3 shows the timing for writing data to the memory. Observe that *WriteEn* is only high for addresses *A1* and *A2*. This means that only data words *D1* and *D2* are written, respectively.

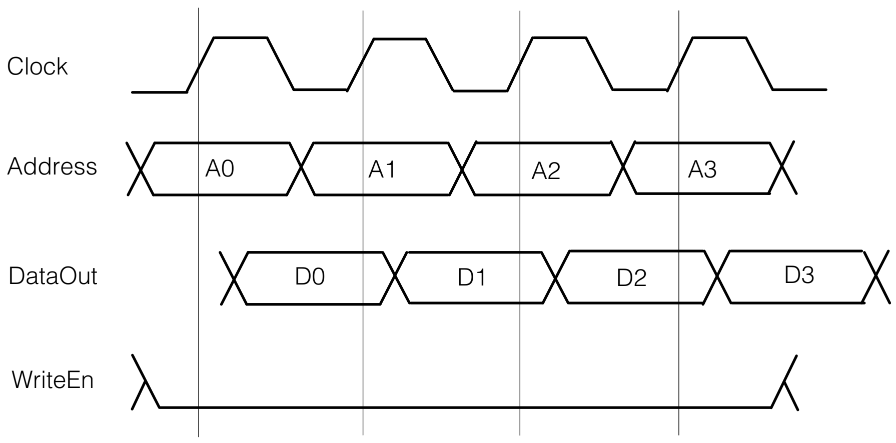

Figure 2: Timing diagram for read operations.

Perform the following steps:

1. Open Quartus Prime.
2. You will now create a memory module that you can include into your design. First, select Tools–>IP Catalog. Note that the IP Catalog usually shows up as a pane on the right side of the Quartus Window, and may have already been there by default. **(PRELAB)**
3. In the IP Catalog pane, expand Installed IP–>Library–>Basic Functions–>On Chip Memory, then double-click on *RAM:1-PORT*. **(PRELAB)**
4. Browse to the folder or directory where you want to build your project. This is where the file for the memory module will be created. Call the file *ram32x4.v*. Choose the IP variation to be Verilog and click OK. **(PRELAB)**

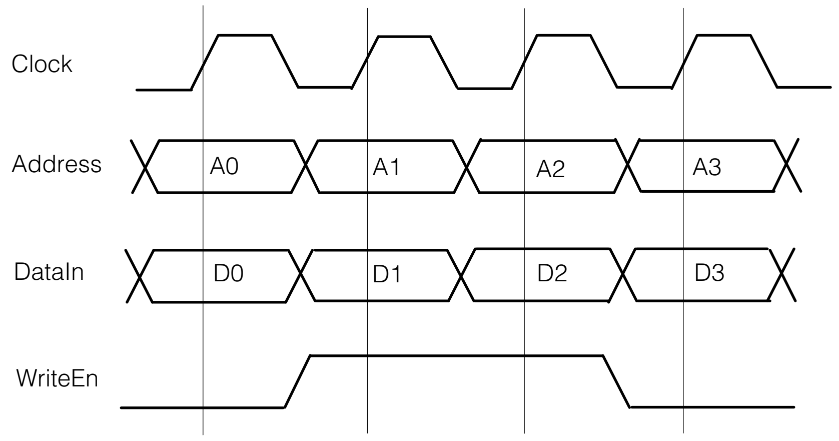

Figure 3: Timing diagram for write operations. Note that only addresses *A1* and *A2* are written.

5. Select a 4-bit wide (width of 'q' output bus) memory with 32 words. Leave the memory block type as *Auto* and use a *Single Clock*. Observe how the memory block diagram on the left side of the window changes automatically as you change different parameters. Pay particular attention to the fact that registers that hold the address, data and write enable signals are included in the block diagram. This means they will be part of the memory module, and you do **not** need to add extra registers outside the memory module. Click Next. **(PRELAB)**
6. **Unselect** q as a registered port. **(PRELAB)**
7. Click Finish and Finish again to generate the new Verilog file, *ram32x4.v*. **(PRELAB)**
8. Examine the newly created Verilog file. Observe that it declares a module with the required ports as shown in Figure 1. You can now instantiate the module into any design. **(PRELAB)**
   Note that the created Verilog file contains some Verilog constructs you may not have encountered before. You do not need to understand all of these constructs to use the memory module, as long as you correctly connect all required input and output ports.
9. Simulate the created memory module using ModelSim for a variety of input settings, ensuring the output waveforms are correct. Show that you can read and write to the memory at several locations. You must show this to the TA as part of your preparation. **(PRELAB)**
   Note that since your design includes embedded FPGA blocks, you must add a special switch to your *vsim* command to be able to simulate the circuit in ModelSim:
   ```
   vsim -L altera_mf_ver ram32x4
   ```
   The *-L* switch specifies that ModelSim should include Verilog models for FPGA embedded blocks found in the *altera_mf_ver* library, which are provided with Quartus Prime to allow simulation of FPGA embedded blocks, such as memory blocks.
10. Instantiate the *ram32x4* module into a top-level Verilog module that connects to the inputs and outputs in the following way: Connect SW[3:0] to the data inputs, SW[8:4] to the address inputs, SW[9] to the Write Enable input and use KEY[0] as the clock input. Show the address on HEX5 and HEX4, the input data on HEX2 and the data output of the memory on HEX0. **(PRELAB)**
11. Draw a schematic describing the circuit and explain it to the TA as part of your preparation. **(PRELAB)**
12. Create a new Quartus project and add your top-level module file and *ram32x4.v*. Make sure the project is stored in your `W:\` drive. Make sure to select the correct FPGA device (5CSEMA5F31C6) and import the pin assignments.
13. Compile the project. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit.

Part II

For this part you will learn how to display simple images on the VGA display. Your task is to design a circuit to draw a *filled* square on the screen at any location in any colour. You are provided with a VGA adapter module that provides the functionality of accepting a set of (x, y) coordinates, known as a pixel, on the screen and a colour to draw at that pixel.

Background

The VGA adapter that you will use is shown in Figure 4. The inputs to the adapter module are similar to the memory interface in Part I. The (x, y) inputs specify a pixel location on the screen while *colour* specifies the pixel colour. The *plot* input is a write enable signal that tells the controller to update the pixel specified by (x, y) coordinates with the value *colour* at the next clock edge. The outputs of the adapter drive the off-chip video digital-to-analog converters (DACs) that subsequently drive the monitor. Please note the VGA adapter code is provided to you and you do **not** need to modify it.

To make things a bit easier, you will work with a screen that is **160 pixels wide by 120 pixels high**. Compare this with standard VGA, which is 640 × 480 pixels and your HDTV at 1920 × 1080 pixels. You can think of your 160x120 screen as a 2D array where each pixel's location can be uniquely identified by a set of coordinates (x, y). The pixel at the top-left corner of your screen (the way you look at it) has coordinates (0, 0), while the pixel at the top-right corner of your screen has coordinates (159, 0). The pixel at the bottom-right corner of your screen has coordinates (159, 119). Note that pixels in the same row share the same y coordinate value, while pixels in the same vertical column share the same x coordinate value.

We will be using only three bits to specify a pixel's colour. The three bits correspond to red, green, and blue, which is called an *RGB* colour coding. There is one bit for each colour so if you want to draw red, then input (1,0,0) as the colour bits. You can combine colours as well, such as (0,1,1), which will produce a colour you would get if you shone two beams of light, one green, one blue, into the same spot. Note that mixing light beams of different colours produces an *additive* colour, which is different than the colour you would get by mixing two paint colours (subtractive colour). While you will use only one bit per colour, a very fancy display may have 24 bits per colour.

An important consideration is the amount of memory required to store the pixels. The reason why we are using a very small screen with only three bits per pixel is so that we only need a small memory. Figure 4 includes a memory block, which is called the *frame buffer*. The *VGA Signal Generator* in Figure 4 continuously reads the frame buffer and drives the signals that are sent to the monitor. Note that the frame buffer is already implemented within the VGA adapter module provided to you.

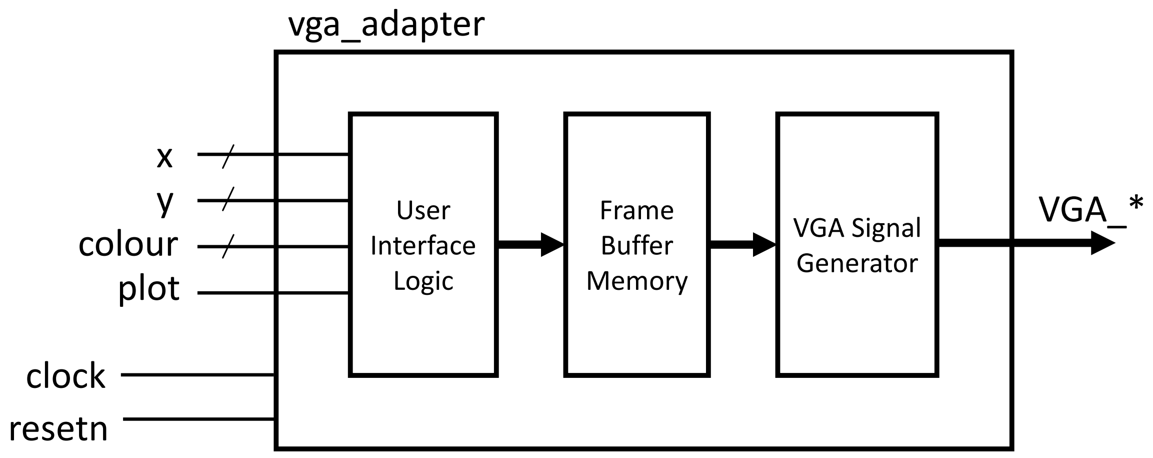

Figure 4: VGA adapter module schematic.

Drawing Squares on the VGA - Expected Behaviour

Your circuit will accept an X and Y location as input, as well as a colour. All these inputs will be provided via the switches as detailed below. The circuit should then draw a square whose size is 4×4 pixels, and whose *upper-left* corner is at the (X,Y) location specified by the input. The square should be filled with the colour specified by the input.

Here are the details about your inputs:

- KEY[0] should be the system active low *Resetn*.
- SW[9:7] should be used to specify the colour.
- SW[6:0] should be used to input (X,Y).

  Notice that although the VGA adapter has 160×120 pixels, we don't have enough switches to be able to specify the coordinates separately and to the full range of X. For X we need eight bits and for Y we need seven bits. We are short one switch for eight bits, so we will just use seven switches and only be able to access the first 128 columns of the display. However, we will still use an 8-bit register to hold the X value. To set a value for X, first set SW[6:0] and press KEY[3] to load a register with the X value. This load should also set the most significant bit of X to 0.
- The filled square should be drawn when KEY[1] is pressed.

After a square has been drawn, your circuit should allow additional squares to continue to be drawn (say, at different locations in possibly different colours). The high-level design of the circuit for the system is given in Figure 5. It contains 3 major blocks:

1. The VGA adapter is responsible for the drawing of pixels on the screen. This code is provided to you and also includes the frame buffer.
2. The datapath that contains arithmetic circuitry and registers, controlled by the FSM, that produce the (X,Y) values that are fed into the VGA Adapter to draw the 4×4 filled square.
3. A finite state machine that serves as a controller for the datapath. Some output control signals are shown in Figure 5 as examples. You need to decide how many control signals are needed based on your datapath. The three signals (ControlA, ControlB, ControlC) shown in the figure are just examples.

Perform the following steps:

1. Design (draw the schematic and write Verilog by adding to the provided skeleton code in `lab7-part2.zip`) and simulate a datapath that implements the required functionality. Note that the provided skeleton code incorporates the VGA adapter. Your simulation for this part should include the datapath only, not including the FSM or the VGA adapter. **(PRELAB)**
2. Design (draw a state diagram and write Verilog by adding to the provided skeleton code) and simulate an FSM that controls the implemented datapath. Your simulation for this part should include the FSM only, not including the datapath or the VGA adapter. **(PRELAB)**
3. Since the VGA adapter connects to an external circuit whose details you may not fully understand, it is not straightforward to simulate your entire top-level design. You may need to simulate the combination of the control and datapath circuits (but not including the VGA adapter) if you find that your design does not work. You should then verify that the outputs from the datapath (inputs to the VGA adapter) are correct. That is, your prelab should include a simulation that demonstrates that all of the pixel writes to the vga_adapter module work as anticipated. **(PRELAB)**

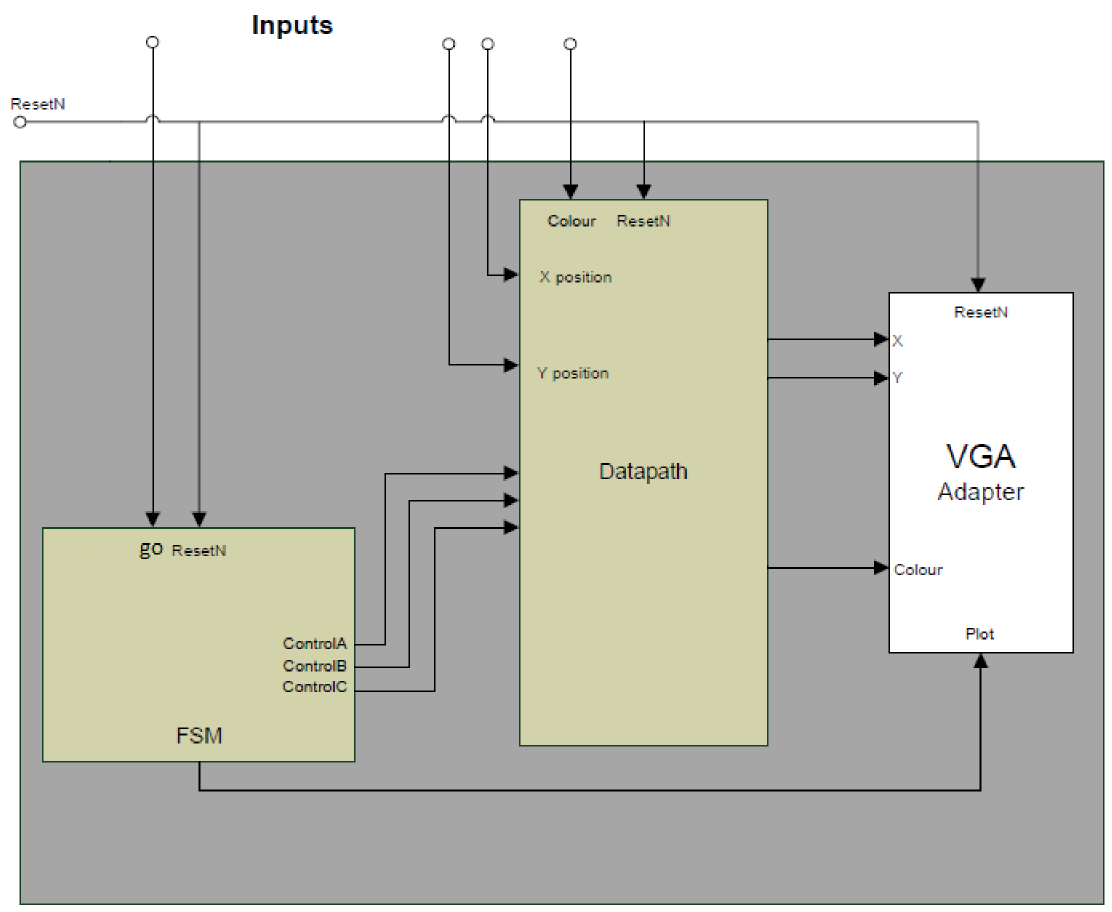

Figure 5: Design Overview - State Machine, Datapath and VGA Adapter. Although not shown, the ResetN signal should be connected to all the registers in the circuit (including the FSM state registers). The enable signal ($KEY[3]$) for the register that holds coordinate X is not shown.

4. Build your top-level design. If you examine the provided skeleton code carefully, you will note that it refers to a file called `black.mif`. This initializes the frame buffer when the FPGA is first configured. The `black.mif` file provided to you creates an "all-black" image, meaning that the screen will be blank when the circuit is programmed into the FPGA.
5. Compile and implement the design on the FPGA using Quartus. Make sure your project is stored in your `W:\` drive. Make sure to select the correct FPGA device (5CSEMA5F31C6) and import the pin assignments.

Advice

You may want to first draw a single pixel on the screen to ensure the pin assignments are all OK. And keep in mind that colour $3'b000$ is black, so if your switches are all set to zeros, you won't be able to see anything on the black background.

You will need to use a counter to help you plot the 16 pixels of the 4×4 colour-filled square. If you were to draw a 2×2 square, one way to implement this would be to use a 2-bit counter to represent the four pixels that need to be drawn. If the most significant bit of the counter was added to the *y* coordinate and the least significant bit of the counter added to the *x* coordinate, this would end up writing to all four pixels of a 2×2 square. You will need to adapt this idea for the larger 4×4 square.

Note that your circuit will be sending the coordinates to be filled to the VGA adapter one pixel at a time. However, since you will be using the 50 MHz clock *CLOCK_50* to drive your circuit, this will occur so fast that you will not be able to tell the difference (i.e., all pixels will appear to be drawn on the screen at once).

Part III - Bonus

You are strongly encouraged to at least attempt this part, especially if you are planning to use animations for your course project.

In this next part we will create a simple animation of the box from Part II by having it bounce around the screen. The colour of the box (4×4 pixels) will be selected by the switches but now the (X,Y) location of the box will be controlled by your circuit and will change over time. To accomplish the animation, your circuit will have to make it seem as though the box is seamlessly moving around the screen. It will do this by erasing and redrawing the box each time it is to be moved. We would like the box to always move in a diagonal fashion at four pixels per second. The VGA adapter updates the monitor at 60Hz or 60 frames-per-second, meaning that the entire contents of the frame buffer are output to the monitor every 1/60th of a second. Your circuit should not erase and redraw the box any faster than this rate.

You should use a counter to track how much time has passed. The counter should count for 1/60th of a second. You should also use a second counter to track how many frames have elapsed. If we want the box to move at four pixels per second, the box should only move one pixel every 15 frames.

You will implement the circuit in two steps. First, you will design the datapath for a module that is able to draw (or erase) the image at a given location. The datapath of this circuit will basically be the circuit used for Part II. In addition, you will need two counters that will contain the current (X,Y) location of the box as well as two single-bit direction registers (horizontal and vertical) that will track the direction the box is moving. The (X,Y) counters will be able to count up or count down since the box can be moving in any direction on the screen. The two single-bit direction registers will track the current diagonal direction of the box (up-left, up-right, down-left, down-right). To implement the *bounce* off the edges of the screen, the current location of the box and direction of travel should be used to update the direction registers. For example, if the box is moving in the down-right direction and the next position of the box would move it off the bottom of the screen, the vertical direction bit would be flipped indicating the box should start moving in an up-right direction. Likewise, if the box was moving in the down-right direction and the next position of the box was further than the right edge of the screen, the horizontal direction bit would be flipped indicating the box should start moving in a down-left direction.

A block diagram of your circuit is shown in Figure 6. It is not complete and lacks some details and signals. Consider it only as a starting point.

Use the same switches as you used in Part II, as needed. Remember that X and Y are no longer input from the switches.

A rough outline of the algorithm is as follows:

1. Reset the 1/60th second Delay Counter and the Frame Counter. Reset Counter_X to 0 and Counter_Y to 60, which will define the starting position of the square. Reset the direction registers to indicate up-right. You can choose how you encode the directions in terms of what a 1 means in the direction register.
2. Use the Part II datapath to draw the box in the current location.
3. Reset the 1/60th second Delay Counter and the Frame Counter. Then count 15 frames.
4. Use your Part II datapath to erase the current box.
5. Update Counter_X, Counter_Y based on the direction registers. Update the direction registers themselves (if necessary).
6. Go to Step 2.

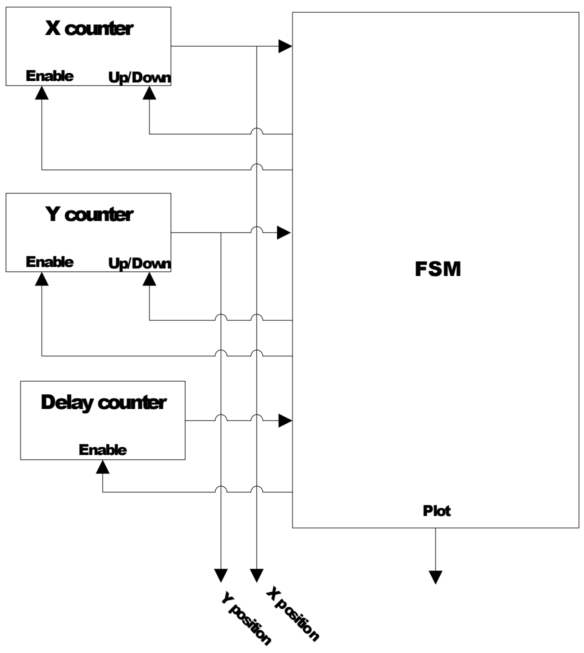

Figure 6: Rough schematic for your animated image circuit. There may be signals and pieces missing.

Implement the circuit by completing the following steps:

1. Design (draw the schematic and write Verilog) and simulate a datapath that implements the required functionality.
2. Design (draw state diagram and write Verilog) and simulate an FSM that controls the implemented datapath
3. Since the VGA adapter connects to an external circuit whose details you may not fully understand, it is not straightforward to simulate your entire top-level design. If you wish, you can skip detailed simulation of the combined controller/datapath circuit. However, you may need to perform this simulation for debugging purposes if your circuit does not work on the board. You should at least verify that the outputs from the datapath (inputs to the VGA adapter) are correct.
4. Build your top-level design. Compile and implement the design on the FPGA using Quartus. Make sure your project is stored in your `W:\` drive. Make sure to select the correct FPGA device (5CSEMA5F31C6) and import the pin assignments.
5. Demonstrate the working circuit to your TA.