CSC B58 - Lab 6

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

Prelab	/2
Part I (in-lab)	/3
Part II (in-lab)	/2
Clean work-space with all materials returned to their original state	/1
TOTAL	/8

Write your name, UTorID, and student ID:

Name: ______________________________

Student ID: ______________________________

UTorID: ______________________________

Write your partner's name, UTorID, and student ID:

Partner name: ______________________________

Partner student ID: ______________________________

Partner UTorID: ______________________________

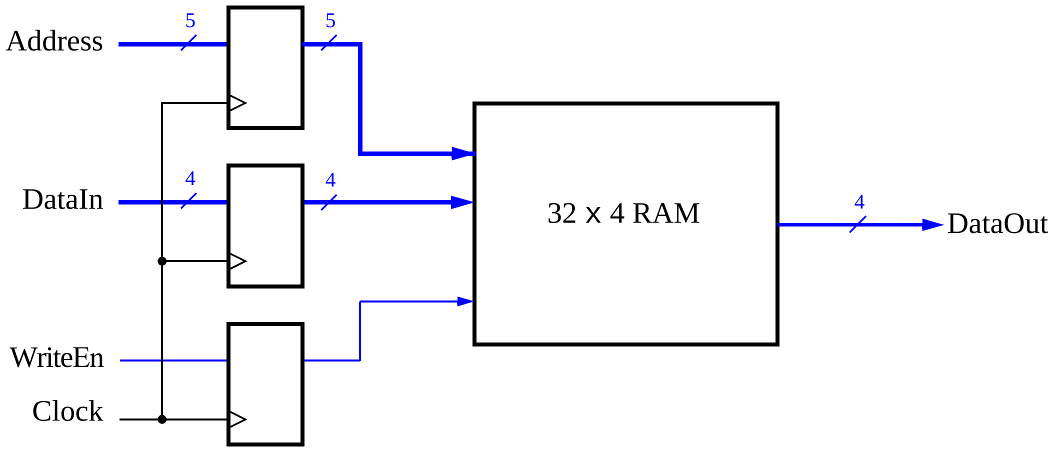

Figure 1: Schematic of the 32 $\times$ 4 embededed memory module.

Part I

In addition to lookup tables (LUTs) and flip flops, the FPGA provides flexible embedded memory blocks that can be configured into various bit widths and depths along with many other parameters. To access these blocks you will use another feature of Quartus that can build modules of various functions. 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, address register, data register and a control register. You can see that the address and input data are stored in a register as well as the Write Enable control signal. 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 (the rising clock edges are marked with gray vertical lines in the timing diagrams below). It is a small memory block so that we can easily interact with it using the available switches and displays on the the DE2 board.

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 (each address is 5 bits wide, while data is 4 bit wide). 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.

Perform the following steps:

1. Open Quartus.
2. You will now use Quertus to create a memory module that you can include into your design.
3. First, select Tools–>IP Catalog.
4. Open Installed IP–>Library–>Basic Functions–>On Chip Memory–>RAM:1-PORT.
5. 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.
6. 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*. Click Next.
7. Unselect q as a registered port.
8. Click Finish and Finish again to generate the new Verilog file, *ram32x4.v*.

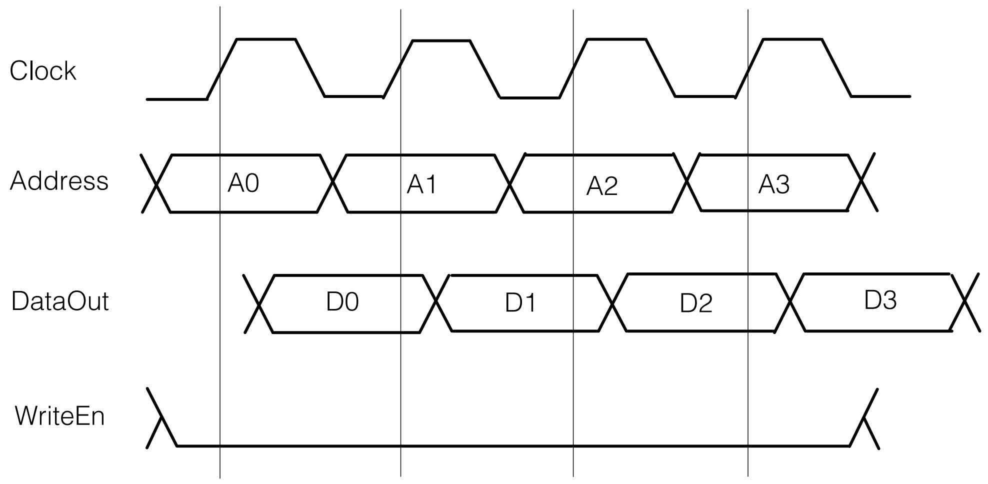

Figure 2: Timing diagram for read operations.

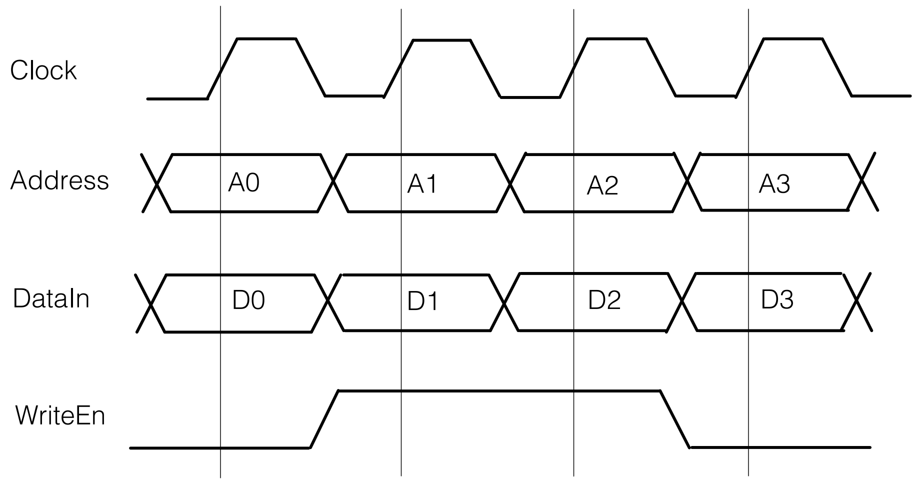

Figure 3: Timing diagram for write operations. Note that only addresses *A1* and *A2* are written.

9. Examine the newly created Verilog file. Observe that it declares a module with the required ports as shown in Figure 1, although the names of inputs and outputs may be slightly different. You can now instantiate the module into any design.
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.
11. Create a new Quartus project and add your top-level module file and *ram32x4.v*.
12. Draw a schematic describing the circuit and explain it to the TA as part of your preparation.
13. Compile the project.
14. 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. A screen is a 2D array of *pixels* (picture elements), each of them can take one of several colours. For example, a 640x480 display with 24 bits per pixel means the display is 640 pixels wide and 480 pixels tall, and each pixel takes on one of $2^{24} = 16,777,216$ possible colours. You are provided with a VGA adapter module that provides the functionality of accepting a set of (x, y) coordinates of 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.

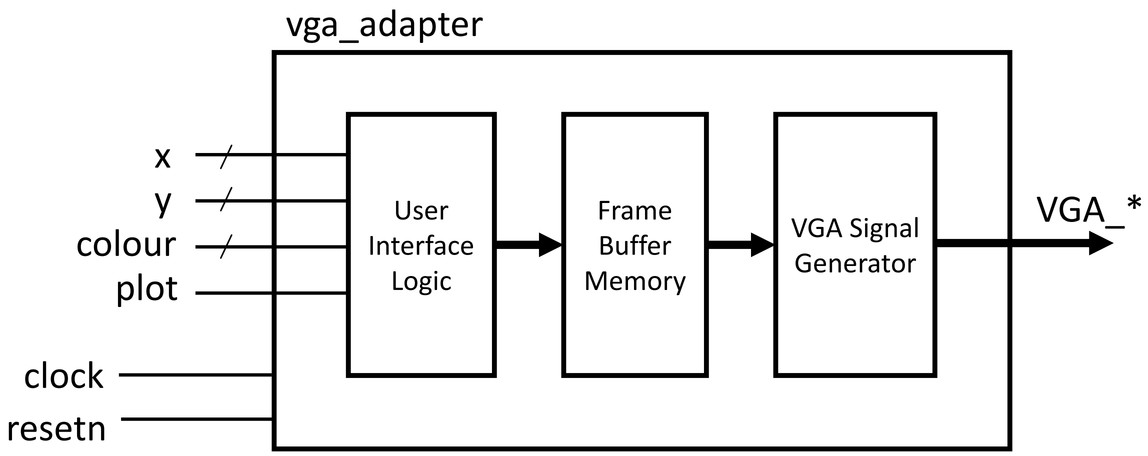

Figure 4: VGA adapter module schematic.

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 (X is horizontal location, Y is vertical location) 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) with the value *Colour* at the next clock edge. The outputs of the adapter drive the off-chip video digital-to-analogue converters (DACs) that subsequently drive the monitor.

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 $\times$ 480 pixels and an HD TV at 1920 $\times$ 1080 pixels. We are also using only three bits for the colour of a pixel. 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 blue, then input (0,0,1) as the colour bits, while (1,1,0) will make it yellow (red + green). Modern displays tend to have 24 bits per pixel (8 for red, 8 for green, and 8 for blue), while some fancy display may even have up to 16 bits for each colour channel!

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. In Figure 4 you will see the memory block, which is called the *frame buffer*. The frame buffer holds the color values for all pixels on the screen. The *VGA Signal Generator* in Figure 4 continuously reads the frame buffer and drives the signals that are actually sent to the monitor. Note that the frame buffer is already implemented within the VGA adapter module provided to you.

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 4×4 pixel square 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*. **(as always, use a switch if the key is problematic)**
- SW[9:7] should be used to specify the colour.
- SW[6:0] should be used to input X and Y: SW[6:0] normally controls Y; to set a value for X, first set SW[6:0] and press KEY[3] to load a register with the value for X.

  Note 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. Therefore when loading X, you should also set the most significant bit of X (i.e., the eighth bit) 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 lines are shown in Figure 5 as examples. You need to decide the number and bitwidths of these outputs. The 3 listed there (ControlA, ControlB, ControlC) are just for example.

Perform the following steps:

1. Design (draw the schematic and write verilog by adding to the provided skeleton code in `lab6-part2.zip`). Note that the provided skeleton code incorporates the vga controller. **(prelab)**
2. Design (draw a state diagram and write verilog by adding to the provided skeleton code) an FSM that controls the implemented datapath. **(prelab)**
3. 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. In this case, it creates an all-black image. **(prelab)**
4. Compile and implement the design on the FPGA using Quartus.

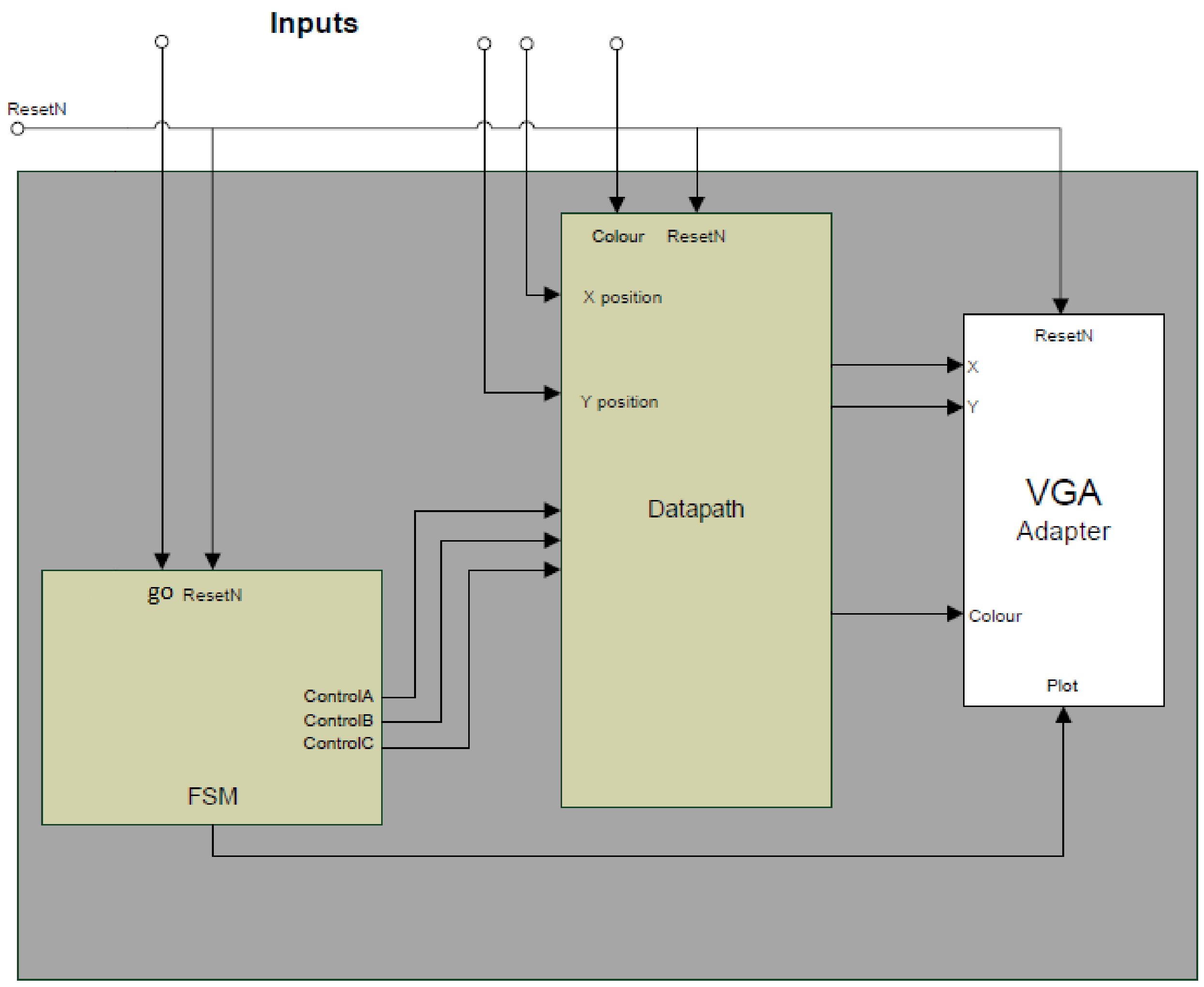

Figure 5: Design Overview - State Machine, Datapath and VGA Adapter. Although not shown, the ResetN signal should be connected to all the registers in the circuit (including the FSM state registers). The enable signal ($KEY[3]$) for the register that holds coordinate X is not shown.

Advice

You will need to use a counter to help you plot the 16 pixels of the 4×4 colour-filled square. Let's say you were to draw a 2×2 square. In this case you could use a 2-bit counter: you would add the most significant bit of the counter to the y coordinate, and the least significant bit of the counter would be added to the x coordinate. Think about how this create a 2×2 square where the top-left corner is (x,y). You will need to adapt this idea for the larger, 4×4 square.

Note that you will be sending the coordinates of one pixel at a time to the VGA adapter. However, since you will be using $CLOCK\_50$, you will not be able to tell the difference (i.e., all pixels will appear to drawn at once on the screen).

Part III - Optional

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 (i.e., drawing with black color) and redrawing the box each time it is to be moved. We would like the box to always move diagonally at the speed of 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 only erase and redraw the box no 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, or 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 rough schematic of your circuit is shown in Figure 6. It is not complete and most likely lacks some pieces and some 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 and Counter_Y to 0. Reset the direction registers to indicate down-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.

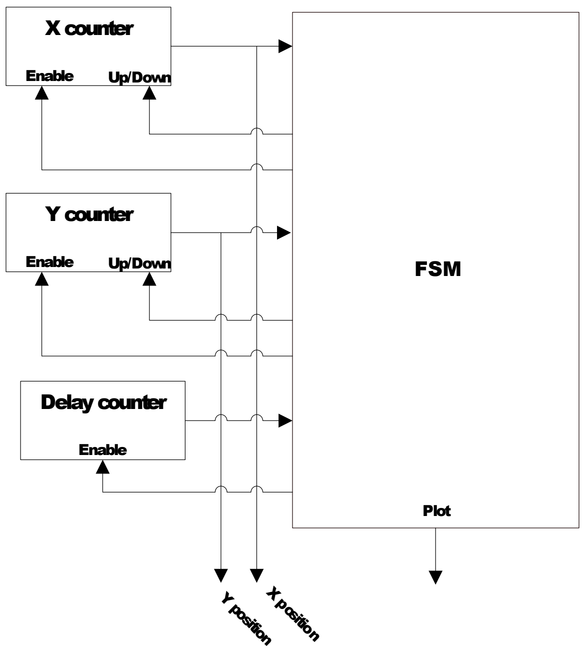

Figure 6: Rough schematic for your animated image circuit. There may be signals and pieces missing.

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. Again, since the VGA adapter connects to an external circuit, it is not so easy to simulate your entire top-level design. We will try and skip the step of combining the controller and datapath for simulation, which you really should do and you may still need to do this if things do not work. You should at least verify the outputs from the datapath (inputs to the VGA adapter) are correct.
4. Build your top-level design.
5. Compile and implement the design on the FPGA using Quartus.