Laboratory Exercise 7

Memory and the VGA Display

Revision of November 16, 2021

The purpose of this exercise is to learn how to create and use on-chip block random access memories (BRAMS) as well as use the video graphics adapter (VGA).

1 Workflow

For each part of the lab, you should begin by writing and testing Verilog code, using ModelSim. Once your design works in simulation, you should compile it with Quartus. You must simulate your circuit with ModelSim using reasonable test vectors written in the format used in Lab 2 for the simulation files. You should be prepared to show schematics, Verilog, and simulations to your TA, if requested. You will not be helped with debugging your code if you do not have schematic and ModelSim simulations. Issues identified with the tester, without doing your own ModelSim simulations will not get help.

Warning: This lab document is quite detailed and covers a lot of new information. Please read through it carefully. Do not rush to just get the labs done as you will spend much more time debugging your code than making progress. Also, it is highly recommended that you watch the tutorial videos on Quercus, in the Lab 7 page.

2 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. This will create a module that you can then instantiate in your designs when you need such a memory block. You can test the memory module using ModelSim and, if you have a board, 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

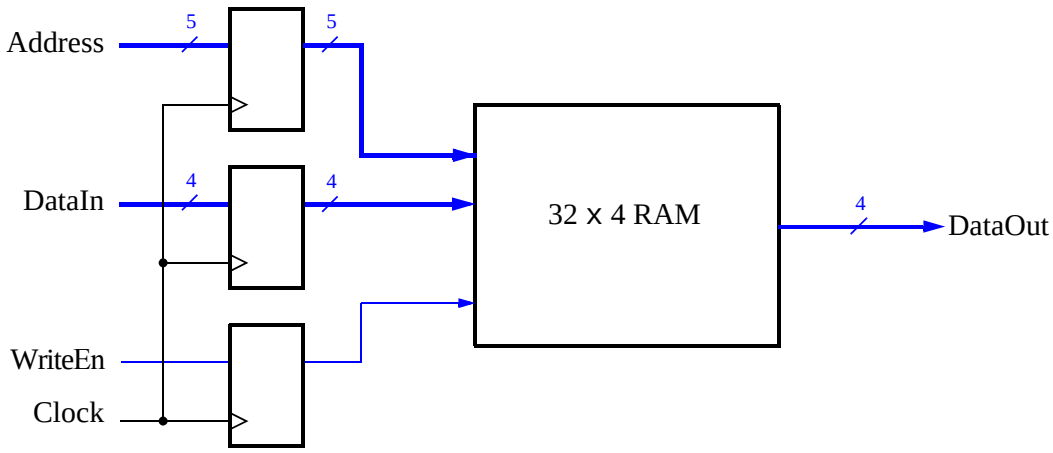

Figure 1: Schematic of the 32 × 4 embedded memory module.

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 the DE1-SoC board.

Figure 2 shows a timing diagram for reading from the memory. 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.

To generate the memory module for the small memory shown in Figure 1, perform the following steps.

1. Open Quartus.
2. You will now create a memory module that you can include into your design. First, select Tools–>IP Catalog
3. Open Installed IP–>Library–>Basic Functions–>On Chip Memory–>RAM:1-PORT
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 `part1.v`. Choose the IP variation to be Verilog and click OK.
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*. Click Next.
6. Unselect q as a registered port.

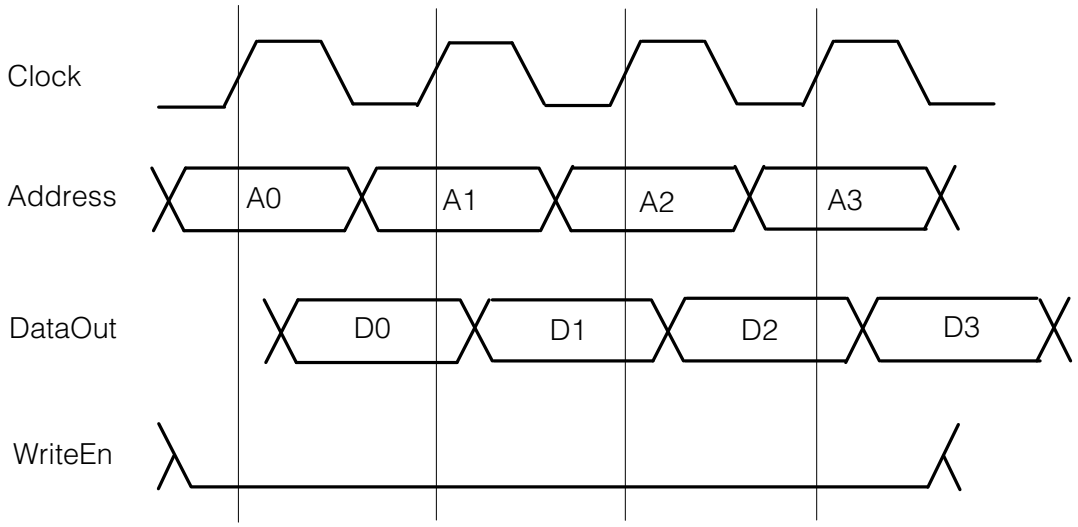

Figure 2: Timing diagram for read operations.

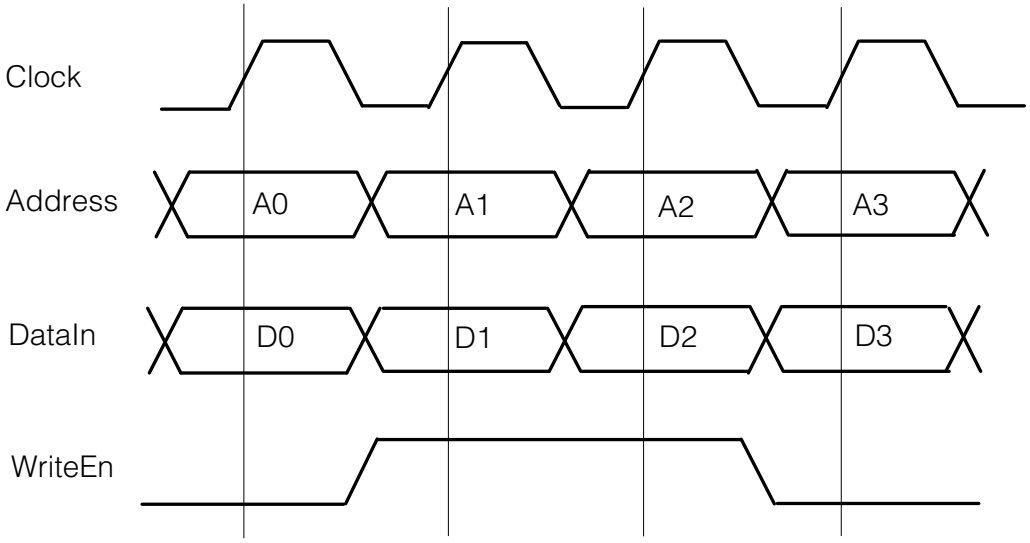

Figure 3: Timing diagram for write operations. Note that only addresses *A1* and *A2* are written.

7. Click Finish and Finish again to generate the new Verilog file, `part1.v`.

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.

9. Simulate your `part1` module with ModelSim to satisfy yourself that your circuit is working. Be prepared to justify that your test cases are enough to give confidence that your circuit is working.

   Since you are adding the BRAM IP to your project, you will need to add an additional parameter to simulate this. The following shows how to do this in your Modelsim scripts:

   ```
   vsim -L altera_mf_ver <module-name>
   ```

   For some more information on what this does, read Section 5 below. When you are satisfied with your simulations, you can submit to the Automarker.

10. Create a new Quartus project for your circuit. You will need a top-level module to make connections from the instantiation of your `part1` module to the switches, pushbutton and HEX displays of the DE1-SoC board. 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. Simulate your new circuit with the DE1-SoC connections with ModelSim to ensure that you have done the connections correctly.

11. Compile the project to generate a bitstream to make sure your code can at least be synthesized.

12. If you have a board, download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the various inputs and observing the outputs.

13. If you do not have a board, you can use *fake_fpga* to test that your circuit behaves as expected.

3 Part II

For this part you will learn how to display simple images on a VGA display. Your task is to design a circuit to draw a *filled* square on the screen at any location in any colour.

There are two ways that you can do this lab:

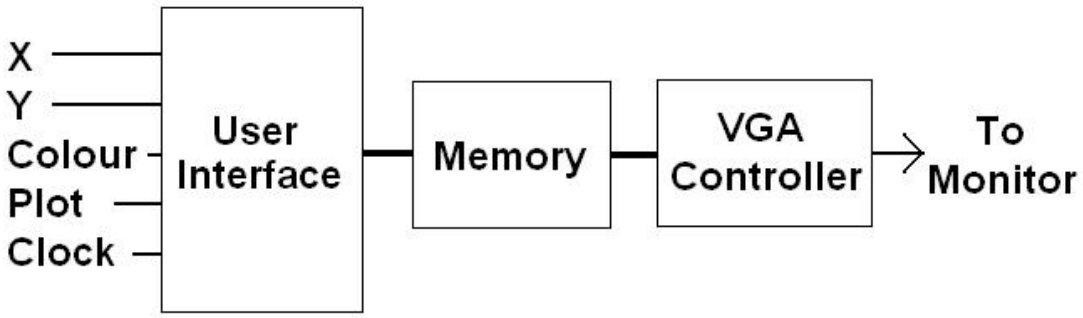

Figure 4: VGA adapter module schematic. Resetn is not shown.

DE1_SoC Board + VGA Monitor The DE1_SoC board has a VGA interface that enables you to connect the board to a VGA monitor. You are provided with a VGA adapter module in Verilog. It has an input interface of a pixel coordinate and a colour for the pixel. A pixel is one of the *dots* drawn on your screen. A VGA display is made up of 640×480 pixels. The VGA adapter logic will draw the colour on the screen at the specified pixel coordinate. The VGA adapter that you will use is shown in Figure 4.

fake_fpga *fake_fpga* has a simple VGA-like interface that can display images at the bottom of the GUI. When using *fake_fpga*, you can assume that the VGA adapter is a part of *fake_fpga*, rather than being a module that you have to instantiate. The interface to the *fake_fpga* VGA is the same as the physical VGA adapter module interface shown in Figure 4. This is a good alternative if you do not have the hardware available.

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. Note that `(X,Y) = (0,0)` is the top left corner of the screen. 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 VGA adapter drive the off-chip video digital-to-analogue converters (DACs) that subsequently drive the monitor.

3.1 (OPTIONAL) If you want to learn more about VGA and the VGA adapter

You are given enough information in this lab sheet to use the VGA adapter module, but if you are curious to learn more about the details you can get more information at `http://www.eecg.toronto.edu/~jayar/ece241_08F/vga/`. If you do not want to, you can skip this part and move onto Section 3.2.

In terms of other changes you can make to the VGA, you may want to change the resolution

and colour quality, in which case you should look at "Changing Adapter Parameters". If you stick to using the names of the ports used at that site, you will get the correct pin mapping through the `DE1_SoC.qsf` file that you have been provided, except for two port names that must be changed, as described in the next paragraph.

A word of caution if you want to try the example code at that web site. You will see mention of the DE1 and DE2 boards. You are using the DE1-SoC, so the examples may not work directly. In particular, watch for the names of two signals in the old examples. The signals were changed from `VGA_BLANK` and `VGA_SYNC` to `VGA_BLANK_N` and `VGA_SYNC_N`, respectively, for the DE1-SoC so you will have to change any of the example code accordingly. The files you are provided for this lab are for the DE1-SoC and will not need the above modification.

3.2 Constraints and Colour Coding

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 HD TV at 1920 × 1080 pixels or 4K UHD at 3,840 × 2,160 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 red, then input (1,0,0) as the colour bits. You can combine colours as well, such as (0,1,1), which will draw green and blue resulting in cyan. A very fancy display may have 8 bits per colour, for a total of 24 bits of colour, to define the colour of a pixel.

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. How many bits of memory are require to store an image size of 160×120 pixels where each pixel is three bits? How many bits of memory are require to store a UHD image size of 3,840×2160 pixels where each pixel is 24 bits?

In Figure 4, the memory block shown is called the *frame buffer*. Through the user interface, a user can update the contents of the frame buffer. The *VGA Controller* in Figure 4 continuously reads the frame buffer and displays the image in the frame buffer by driving the signals that are sent to the monitor.

3.3 Your Circuit

Your circuit will have the following inputs:

`iClock` The clock input to your circuit.

`iResetn` Active low synchronous reset.

`iXY_Coord`, `iLoadX`, `iColour` The circuit should draw a 4×4 pixel square whose *upper-left* corner is at the `(X,Y)` coordinate specified. The square should be filled with the colour specified by the `iColour` input. Recall that coordinate `(0,0)` is the top left corner of the screen for our VGA adapter.

The input `iXY_Coord` is used to input the `X` and `Y` coordinates. This is done in two steps. First the `X` coordinate is input at `iXY_Coord` and `X` is loaded into a register when `iLoadX` is set high and then low. After `iLoadX` is pulsed, then the `Y` coordinate should be input at `iXY_Coord`. It will be loaded when the input `iPlotBox` is pulsed to start the drawing. The assumption is that `iXY_Coord` will be connected to some switches. Using one input port for both `X` and `Y` reduces the number of switches required. Also, assume `iLoadX` will be connected to a pushbutton.

`iPlotBox` The filled square should be drawn when `iPlotBox` is pulsed high and then low. The assumption is that `iPlotBox` will be connected to a pushbutton.

`iBlack` When pulsed high and low, this input will cause the entire screen to be cleared to black (*Black*). Note that black corresponds to colour (0,0,0). The assumption is that `iBlack` will be connected to a pushbutton.

Your circuit will have the following outputs: The outputs are intended to connect directly to the corresponding inputs of the VGA adapter or the corresponding ports in *fake_fpga*.

`oX`, `oY` The `X` and `Y` pixel coordinates of the display.

`oColour` The colour to draw at the specified `(oX,oY)` coordinate.

`oPlot` The write enable to tell the VGA Adapter to draw the `oColour` at the specified coordinate.

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 three major blocks:

1. The VGA adapter is responsible for the drawing of pixels on the screen. When using *fake_fpga*, the VGA adapter can be considered to be inside *fake_fpga*, i.e., you do not have to instantiate it.

2. The datapath that contains arithmetic circuitry and registers, controlled by the FSM, that compute 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 may put in whatever you require.

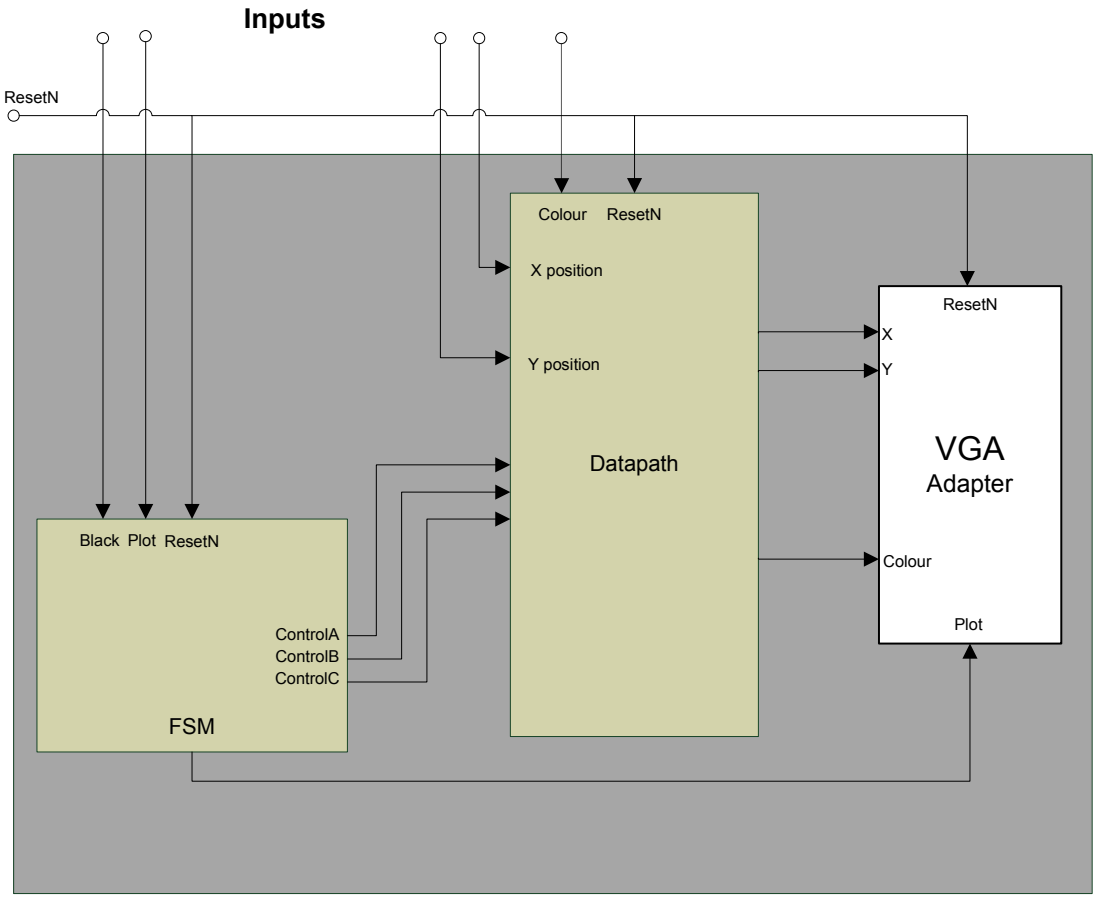

Figure 5: Design Overview - Finite State Machine, Datapath and VGA Adapter. Although not shown, the `Resetn` signal should be connected to all the registers in the circuit (including the FSM state registers).

3.4 How to think about this design

An important part of doing design is to first think through the steps you can take to get to the final design. As you start to deal with greater and greater complexity, you must realize that you cannot build everything at once and then hope that it will all work the first time you try to test it. You should take incremental steps.

To make the design of this Part II more incremental, what can you start with that demonstrates some basic functionality and helps you understand how the VGA adapter works? The ultimate goal is to draw a 4×4 filled square. How do you do that? You have to draw one pixel, move to the next pixel coordinate, draw that pixel, etc. Your FSM will need to be changing the pixel coordinate to reach the appropriate pixels and writing the colour value at each pixel. However, what can you do first that is simpler, but achieves important functionality?

1. Try just drawing one pixel on the screen. This will use a much simpler FSM, so it will be easier to get working. In simulation, you just want to see that the correct inputs make it to the VGA adapter interface. Make the first pixel red. This requires you to set `X`, set `Y`, and `iColour`. Then set `iPlotBox` to plot the pixel. Draw several pixels at different coordinates. Change the colour to green, set `iPlotBox`, then blue, set `iPlotBox` and see if the pixel colour changes accordingly each time. This will test that you can input `X` and `Y` correctly and input the correct colour. If you can do this, then you know that you are using the `X`, `Y` and `iColour` inputs correctly.

2. Now you can try testing the circuit with a display. You can do this with a real board and VGA monitor, or with *fake_fpga*.

   Note that one pixel will be quite small, so you may need to look very carefully to find it. If you are testing on the board this will also show that you are using the VGA adapter correctly.

3. Once you are confident that you are correctly drawing and displaying one pixel at any desired coordinate, you can try plotting the 4×4 square knowing that you already have the ability to draw one pixel. If you have problems with this step, then the problem is most likely in the new code that you have added, so you can focus your debugging on that code.

4. You also need to clear the screen, i.e., make the screen all black. Do this next. It will require a slightly different function that writes (0,0,0) into all pixels. If you think a bit about this, you can create a module that can draw any size box at a specified coordinate with a specified colour. It will take a coordinate, colour and `X` and `Y` dimensions as an input. You can use it to draw both the 4×4 box and to fill the screen, which is just a very large box!

5. At this point, you will have achieved all the functionality required for this part.

3.5 The Top-Level Module

The skeleton code for the top-level module of your design, called `part2`, is provided to you in the file `part2_template.v`. Copy this file and rename it to `part2.v`. This is the file you will submit to the Automarker after you add your code.

Note that the template includes two parameters, `X_SCREEN_PIXELS` and `Y_SCREEN_PIXELS`, which define the size of screen being used. The default values are set to the 160×120 screen size used by the provided VGA adapter and by *fake_fpga*. Use these parameters anywhere you need to specify the size of the screen in your code. The parameters can be changed when the module is instantiated so that the module can work for different sizes of screens. The Automarker will expect these parameters and set them to test for an 8×5 screen size, which reduces the time required for the Automarker. You will find that simulation using a smaller screen size will also be helpful so you should learn to use these parameters.

You should also observe that `iXY_Coord` is declared to only be seven bits even though the default screen size is 160 pixels wide, which would require eight bits to capture the full range. This is because for this part, the module is designed for testing on the DE1-SoC or *fake_fpga*, and there are not enough switches. Therefore, for `part2.v` you will only be able to access 128 columns, instead of the full 160 columns of pixels available, but when you output the `oX` coordinate, it must be eight bits to match the requirements of the VGA adapter. When loading your `X` coordinate register from the 7-bit `iXY_Coord` input, the register should be eight bits and you must set the most significant bit to 0 to fill out the required eight bits.

3.6 What to Do

Perform the following steps to implement your design. It is strongly recommended that you go through the incremental steps outlined in the previous section to arrive at the final design, i.e., you will use the steps here to do several simpler designs leading up to your final design.

1. Design (draw the schematic and write Verilog) and simulate a datapath that implements the required functionality.
2. Design (draw a state diagram and write Verilog) and simulate an FSM that controls the implemented datapath.
3. Build your top-level design and simulate the full system. If you feel you have met all the requirements, then you can submit to the Automarker.
4. You can now proceed to test your design on the real hardware or *fake_fpga*.
5. Create a new Quartus project for your circuit. You will need a top-level module to make connections from the instantiation of your `part2` module to the switches

and pushbuttons of the DE1-SoC board. Connect KEY[1] to `iPlotBox`. Connect KEY[2] to `iBlack`. Connect KEY[0] to `iResetn`. Use SW[9:7] to input the colour at `iColour`. Connect SW[6:0] to `iXY_Coord`. Connect KEY[3] to `iLoadX`. Connect KEY[1] to `iPlotBox`.

6. If you are using the DE1-SoC board, you must also add the VGA adapter so that you can drive the VGA output on the board. Since the VGA adapter connects to an external circuit, it is not so easy to simulate your entire top-level design. This is because the VGA adapter connects to external circuitry and you cannot simulate the external circuitry. In the real-world, this can often happen, and you have to think hard about how to do a simulation. Sometimes you can acquire a simulation model, such as for an SDRAM chip. Other times, you may need to write a model for the external circuitry that can properly respond to your circuit in the FPGA. Here, we will try to get away with checking that the INPUTs to the VGA adapter look correct and assume that the VGA adapter works. Since we've used the VGA adapter for many years, it is a safe bet that you can do this, so do this simulation without the VGA adapter.

   Simulate your new circuit with the DE1-SoC connections with ModelSim to ensure that you have done the connections correctly.

7. Now you can build your top-level design with the VGA adapter. You will find some skeleton code that incorporates the VGA adapter in the file called `fill.v` in the `lab7-part2.zip` file. Use `fill.v` as the template for your top-level that includes the VGA adapter.

   If you examine the 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.

8. Compile the project to generate a bitstream to make sure your code can at least be synthesized.

9. If you have a board, download the compiled circuit into the FPGA chip. Test the functionality of the circuit by drawing boxes at different coordinates with different colours.

10. If you do not have a board, you can use *fake_fpga* to test that your circuit behaves as expected.

4 Part III

In this next part, we will create a simple animation of the box from Part II by having it bounce around the screen.

The skeleton code for the top-level module of your design, called `part3`, is provided to you in the file `part3_template.v`. Copy this file and rename it to `part3.v`. This is the file you will submit to the Automarker after you add your code.

Note that the template includes three parameters, `X_SCREEN_PIXELS` and `Y_SCREEN_PIXELS`, which define the size of screen being used and `CLOCKS_PER_SECOND`, which specifies the clock frequency. The default frequency is set to 5 KHz. As in Part II, the pixel dimensions specify the screen size. The Automarker will change the parameters to test a 9×7 screen. The Automarker will also set `CLOCKS_PER_SECOND` to 1200. Again, these parameters should be used in your design and will be helpful for scaling the design for simulation. Some other parameters are defined and given as examples of how you can use parameters as part of other parameters. You may use these if you find them useful, but they are not required by the Automarker.

The colour of the box (4×4 pixels) will be selected by the input `iColour` but now the `(X,Y)` location of the box will be controlled by your circuit and will change over time. Since your circuit is providing the `X` value, you now can also generate all eight bits so that you can access the full screen in this part.

The module also has `iResetn` and `iClock` inputs used in the same way as in Part II. The outputs of the module are also used in the same way as in Part II.

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. Assuming the background starts out black, then erasing is simply to draw black pixels over top of the region being erased. An incremental step would be to just move the box without erasing it. This will draw all over your screen, but is easier because you do not have to figure out erasing.

4.1 VGA Update Rate

The actual VGA adapter updates the monitor at 60 Hz or 60 frames-per-second (fps), 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.

If you have ever looked at the specifications for a monitor that you buy, you will see common rates of 30 fps or 60 fps. In a more sophisticated VGA adapter, there are actually two frame buffers: one is being updated with the image for the next frame, while the other is being output to the monitor. This is called double-buffering and allows a much smoother transition from one frame to another. In our case, there is only one buffer and if you look really carefully you may see a moving image *tearing*, meaning that while you are redrawing it in the new position, part of it is displayed in the old position and part of it is in the updated position.

When using *fake_fpga*, the output to the VGA display does not happen asynchronously to your circuit, i.e., the *fake_fpga* VGA controller is not drawing the image at any particular frame rate. The image on the VGA is updated on the display every time *Plot* is set high.

4.2 How to think about this design

We would like the box to always move in a diagonal fashion at four pixels per second.

You should use a counter, called `DelayCounter`, to track how much time has passed. The `DelayCounter` should generate an enable pulse every 1/60th of a second, which corresponds to the period at which the VGA adapter updates a frame. You should also use a second counter, called `FrameCounter`, to track how many frame times have elapsed. If we want the box to move at four pixels per second, the box should only move one pixel every 15 frames.

Clock Rates Remember that if you are using a physical DE1-SoC board, `Clock_50` runs at 50 MHz, while with *fake_fpga*, `Clock_50` runs at 5 KHz. The Automarker will set the parameter `CLOCKS_PER_SECOND` to 1200. Keep these in mind when building the `DelayCounter`. The `DelayCounter` is at least one place where you should use the `CLOCKS_PER_SECOND` parameter that can be changed at the time the `part3` module is instantiated.

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, `CounterX` and `CounterY` that will contain the current (X,Y) location of the box as well as two single-bit direction registers, `DirH` for horizontal and `DirY` for vertical, that will track the direction the box is moving. The `CounterX` and `CounterY` counters will be able to count up or count down since the box can be moving in any direction on the screen. The two 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 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.

A rough outline of the algorithm is as follows:

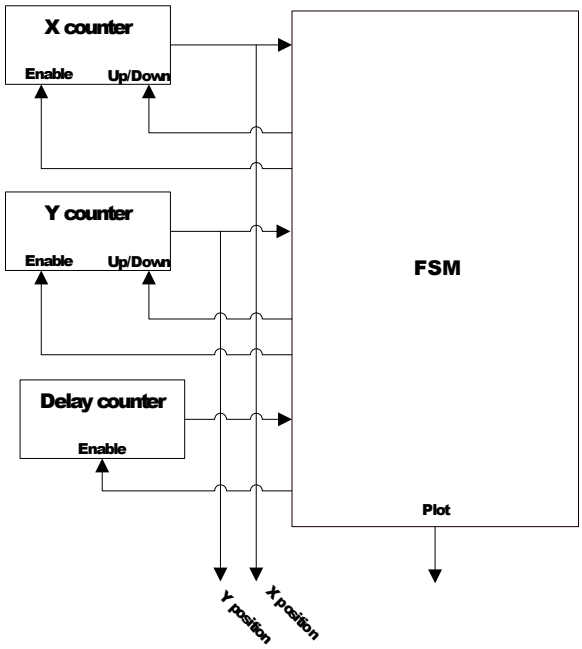

Figure 6: Rough schematic for your animated image circuit. There may be signals and pieces missing.

1. Reset `DelayCounter` and `FrameCounter` to 0. Reset `CounterX` and `CounterY` to 0. Reset `DirH` to 1 and `DirY` 0 to indicate down-right. By doing this, the box always starts moving from the upper-left corner of the screen and moves in a down-right direction.
2. Use the Part II datapath to draw the box in the current location.
3. Reset `DelayCounter` and `FrameCounter` to 0. Then count 15 frames.
4. Use your Part II datapath to erase the current box.
5. Update `CounterX` and `CounterY` based on the direction registers. Update the direction registers themselves (if necessary).
6. Go to Step 2.

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. Do your simulations with Modelsim, again without the VGA adapter. Aside from `iClock`, the only inputs you can set in your simulation are `iColour` and `iResetn`. Check that the outputs are behaving correctly. Once you are satisfied that your circuit is behaving correctly, you can submit it to the Automarker.
4. You can now proceed to test your design on the real hardware or *fake_fpga*.
5. Use the same switches and pushbutton that you used in Part II for `iColour` and `iReset`. Simulate your new circuit with the DE1-SoC connections with ModelSim to ensure that you have done the connections correctly.
6. Now you can build your top-level design with the VGA adapter as you did in Part II.
7. Compile the project to generate a bitstream to make sure your code can at least be synthesized.
8. If you have a board, download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the various inputs and observing the outputs.
9. If you do not have a board, you can use *fake_fpga* to test that your circuit behaves as expected.

5 A Note About Simulating with Quartus Libraries and the VGA Adapter

If you are adding IP blocks (functional blocks of Verilog) generated from Quartus, such as RAMs, you will need to include the simulation libraries for those IP blocks. The VGA adapter, and other modules you might use for projects using the DE1-SoC board, like a keyboard controller, may also include Quartus IP blocks. You will need to include some Altera simulation libraries (ex., `lpm_ver, altera_mf_ver`) to run your simulations.

The following shows how to add a library in your Modelsim scripts:

```
vsim -L altera_mf_ver <module-name>
```

If you need to add several libraries, then you will need additional `-L library_name` flags.

6 Submission

Please submit part1.v, part2.v and part3.v. For module names and port names, please follow the provided templates.