Objective: In this lab, you will design two combinational modules. These modules will be responsible for (a) drawing a simple image on a screen using the VGA circuitry and (b) making that image move around the screen and bounce off the edges like a 90’s screen saver.

Prelab
Complete the following exercises:

1. Draw a functional block diagram for the module `paint.v`. You diagram must show all major internal components.
2. Write pseudo code for the box and velocity update algorithm for `update.v`. Your pseudocode must correctly update box position, detect border collision, and reverse velocities if the drawing window collides with the border and should correlate well with a hardware implementation.

Introduction
In this lab, you will draw an object on a monitor and make that object move around the screen. To do this lab, you will need the Quartus Project in `lab4.zip` found on the course website.

Part 1 – Draw a Simple Image – `paint.v`
Your first objective is to draw a simple object at an arbitrary position on the screen. To do this, you will design and implement in Verilog a combinational module called `paint.v`. Figure 1 is a block diagram showing the inputs and outputs of `paint.v`. The table below describes the inputs and outputs.

---

**Figure 1: Inputs and Outputs for paint.v.**

---
<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Input/Output</th>
<th>Bit-Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Input</td>
<td>10</td>
<td>Current x coordinate of the pixel being drawn. Valid values are from 0 to 639.</td>
</tr>
<tr>
<td>y</td>
<td>Input</td>
<td>9</td>
<td>Current y coordinate of the pixel being drawn. Valid values are from 0 to 479.</td>
</tr>
<tr>
<td>box_x</td>
<td>Input</td>
<td>10</td>
<td>x coordinate of the upper left-hand corner of the drawing window.</td>
</tr>
<tr>
<td>box_y</td>
<td>Input</td>
<td>9</td>
<td>y coordinate of the upper left-hand corner of the drawing window.</td>
</tr>
<tr>
<td>color_select</td>
<td>Input</td>
<td>3</td>
<td>Used to select the color of the pixel being drawn. You get to determine how colors are chosen.</td>
</tr>
<tr>
<td>shape_select</td>
<td>Input</td>
<td>2</td>
<td>Select exactly what gets shown in the drawing window. The specifics of this are detailed later in this lab handout.</td>
</tr>
<tr>
<td>rgb</td>
<td>Output</td>
<td>12</td>
<td>Color of the pixel to be drawn at coordinates (x,y). Bits 11:8 are for the color red, bits 7:4 are for green, and 3:0 are for blue.</td>
</tr>
</tbody>
</table>

Figure 2 depicts the screen you will be drawing on and the precise window you should be drawing in.

![Figure 2: Demonstration of the screen and drawing window.](image)

The resolution of the screen is 640 pixels wide by 480 pixels tall. The pixels are assigned x and y coordinates with (0,0) being located at the upper left-hand corner of the screen. You will draw inside a 32 pixel by 32 pixel drawing window with the coordinates of the upper left-hand corner given by the signals box_x and box_y.

The module `paint.v` should check to see if the coordinate (x,y) – i.e. the coordinates of the pixel currently being drawn on the screen – is within the 32x32 drawing window defined by (box_x, box_y). If (x,y) is not in the drawing window, the signal `rgb` should be all zeros meaning nothing will be drawn to the screen. If (x,y) is in the drawing window, your module should decide whether or not to assign a non-zero value to `rgb` depending on the value of the signal `shape_select`. The exact behavior is given in Figure 3.

For shapes 00, 01, and 10, you must use a combination of comparison, equality, and inequality operators to determine when to assign a non-zero value to `rgb`. For shape 11, you will have to use a 32x32 ROM to
store the image. Because writing the Verilog for a 32x32 bit ROM is a little tedious, a template `rom.v` with all entries set to 0 has been provided. Feel free to use this template or write your own.

![Regions](image.png)

*Figure 3: Region of the drawing window that should be colored for different assignments to the signal shape_select.*

The exact assignment of the color assigned to `rgb` and how that is determined by the `color_select` input is left up to you.

You must write a testbench for your module. Writing a good testbench will make the design process much faster because you will not have to wait for Quartus to compile your project for every change you make, and you will have access to the internal signals in your module. The testbench must *at least* include:

1. Simulation of the behavior of `paint.v` for arbitrary valid coordinates `(box_x, box_y)`.
2. Iteration of `x` and `y` through all pixels in the 640x480 screen while printing the coordinates `(x,y)` to the console using the `$display` command whenever the output `rgb` of paint is non-zero. Not printing output when `rgb` is zero avoids printing all 300,000 pixels to the console.

Once you are satisfied that your design is working in simulation, add your file to the Quartus project `lab4`. This project can be found on the course webpage and the Quartus project file is in the `synthesis` subdirectory. The project will instantiate your module and handle the VGA timings for you. You should not have to change the top level project file just yet.

Compile the project in Quartus and program the DE10-LITE board. Once the board is programmed, press the reset button (KEY[0]). Connect the DE10-LITE board to your monitor using the VGA cable. Make sure SW[9] is set to 1. The drawing window should be on your monitor. Verify that your module paints within the drawing according to the specifications of the lab.

**Part 2 – Animating the Drawing – `update.v`**

Right now, the drawing window just sits in the corner of the screen. This is boring. We would like to make the box move around the screen and bounce off the border. To do this, you must design a new combinational module called `update.v`. A block diagram showing the inputs and outputs to `update.v` is shown in Figure 4. As before, a table describing these signals is given in a table below.
<table>
<thead>
<tr>
<th>Signal</th>
<th>Input/Output</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>box_x</td>
<td>Input</td>
<td>10</td>
<td>Current x position of the drawing window’s upper left corner.</td>
</tr>
<tr>
<td>box_y</td>
<td>Input</td>
<td>9</td>
<td>Current y position of the drawing window’s upper left corner.</td>
</tr>
<tr>
<td>velocity_x</td>
<td>Input</td>
<td>10</td>
<td>Current horizontal velocity of the box.</td>
</tr>
<tr>
<td>velocity_y</td>
<td>Input</td>
<td>9</td>
<td>Current vertical velocity of the box.</td>
</tr>
<tr>
<td>halt</td>
<td>Input</td>
<td>1</td>
<td>Stop box from moving around the screen. (SW[8])</td>
</tr>
<tr>
<td>next_box_x</td>
<td>Output</td>
<td>10</td>
<td>Next x position of the drawing window’s upper left corner.</td>
</tr>
<tr>
<td>next_box_y</td>
<td>Output</td>
<td>9</td>
<td>Next y position of the drawing window’s upper left corner.</td>
</tr>
<tr>
<td>next_velocity_x</td>
<td>Output</td>
<td>10</td>
<td>Next horizontal velocity of the box.</td>
</tr>
<tr>
<td>next_velocity_y</td>
<td>Output</td>
<td>9</td>
<td>Next vertical velocity of the box.</td>
</tr>
</tbody>
</table>

All the signals prefixed with the word “next” will be assigned to the non-prefixed signals (i.e. box_x will be assigned the value of next_box_x) when the next frame is drawn to the screen. Do not worry about this detail.

Velocities should be either +1 or -1. The drawing window should begin with x and y velocity of +1. Velocities should change only when the drawing window collides with a screen border. For example, when the drawing window reaches the bottom of the screen, velocity_y should change from +1 to -1, and when the drawing window reaches the top of the screen, velocity_y should change from -1 to 1. The same constraints apply to velocity_x with the right and left-hand sides of the screen. In general, the changing of velocity_x and velocity_y should be independent from one another. You are required to manually interpret the velocities as signed two’s complement numbers. You must not use the Verilog “signed” attribute. To make this easier, the bit widths of the velocity_x and velocity_y match the bit widths of box_x and box_y. When setting a velocity to +1 or -1, use the full width; for example to set velocity_x to -1:

```
velocity_x = 10’b1111111111;
```

Your module should just generate the “next” signals as a combinational function of the input signals in such a way as to move the drawing window around the screen. One way you might imagine doing this is through a simple update such as

```
next_box_x = box_x + velocity_x;
```
However, there are some things you will have to consider:

1. Your module will need to move the box left and right as well as up and down. This implies that the velocity signals should be interpreted as signed numbers.
2. You will have to detect when the drawing window collides with the borders of the screen (i.e. \( \text{box}_x = 0 \) or \( \text{box}_x = 639 – (32–1) \)), and similarly for \( \text{box}_y \). When this happens, the box should reverse direction. This must be considered both in the update to \( \text{next\_box\_x} \) and \( \text{next\_velocity\_x} \).

A testbench for the `update.v` module has been provided. It will automatically detect and generate an error if the drawing window goes out of bounds.

Once you are satisfied that `update.v` is working correctly in simulation, add the file to the Quartus project and uncomment the instantiation of the `update` module in the top level file `lab4.v`. Compile your design and configure the DE10-LITE board. Once the board is configured, press the reset button (KEY[0]). If all goes well, the drawing window should now be bouncing happily around the screen. To receive full credit:

1. The drawing window must bounce around the screen without ever going out of bounds.
2. The drawing window must stop moving when SW[8] (halt) is 1.
3. You must be able to change the color of the image in the drawing window using SW[2:0].
4. You must be able to change the shape of the image in the drawing window using SW[4:3].

Note that the mapping of the IO pins on the FPGA to the correct ports on the `paint` and `update` modules has already been done in the top level module. Your modules must simply ensure that they provide the correct functionality.

**Submitted Work**

[20 pts] **Prelab –** Solutions to prelab exercises.

[70 pts] **Lab Checkoffs**

**Part 1** [40 pts]

A) [20 pts] Demonstrate your testbench and successful simulation for `paint.v` to your TA.

B) [20 pts] Demonstrate that the module `paint` successfully integrates with the Quartus project `lab 4` and draws simple images in the upper left-hand corner of a monitor using the DE10-LITE board according to the lab specification.

**Part 2** [30 pts]

A) [15 pts] Demonstrate to your TA that your design simulates successfully using the provided testbench.

B) [15 pts] Demonstrate the final design on the DE10-LITE to your TA. The final result should bounce around the screen according to specification.
[10 pts] Lab Report

Submit all Verilog hardware and testbench code that you wrote. Do not include any code that you did not write. Uploading your verilog is essential to receive credit for the entire lab.

A) [5 points] Print and submit during your lab session.

B) [5 points] Upload to Smartsite using the following steps by the end of your lab session:

1. Make a folder on your computer
2. Place all verilog files you wrote into the folder
3. “zip” the folder into a single .zip file
4. Log onto Smartsite, click Assignments on the left side of the page, click on the correct lab number
5. Upload the .zip file as instructed