# Lab Assignment #5

## Guideline
This lab can be done with a partner.

## Objective
To develop a basic SNAKE game by interfacing a PS/2(USB) Keyboard and VGA display with the board

## Reading
Before you start working on this lab, please read the Basys3 Board User Manual regarding PS/2(USB) protocols.
More info on the PS/2(USB) protocol: http://pcbheaven.com/wikipages/The_PS2_protocol/
More info on the VGA Standard: http://www.ece.msstate.edu/~reese/EE4743/lectures/displays/displays.pdf
You may also obtain these documents in Canvas under the lab documentation page.

## Description
In this lab you will be required to create a simple keyboard controller and a VGA controller. The keyboard controller will enable communication from the keyboard. The VGA controller will be used to display some simple graphic patterns on the computer monitor attached to the board.

For the entire lab, keep in mind "How can I test this during early design and simulation stages?" It is recommended that you simulate the core components of your design to ensure the basic logic works correctly. After this, you can use the hardware to begin testing your design. Debugging through a relatively opaque hardware interface is difficult (e.g., trying to debug a graphics controller if the monitor doesn't display anything). Try to make your design very clear, simple, and modular. This allows you to relatively quickly diagnose problems and create potential solutions.

## Submission details
Submit the following things on Canvas:
- Verilog files for each part
- Bit files and XDC files for each part

## Checkout details
Demonstrate each part during the checkout to the TA.

## Grading:

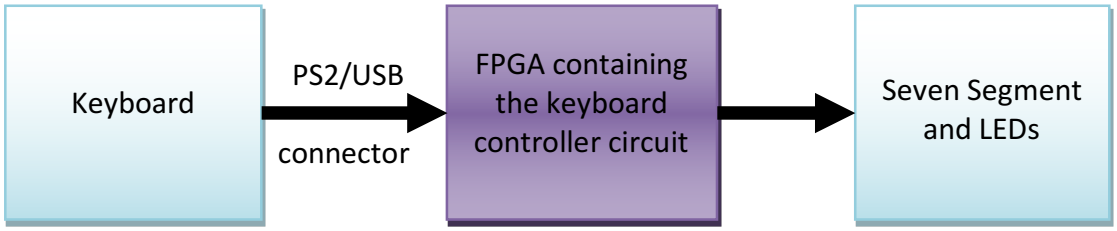
Part A = 50pts (28%)
Part B = 50pts (28%)
Part C = 71pts (39%) + 9pts (5%) = 80pts
Total: 180pts

**Note: 5% of your grade will be based on creativity/uniqueness you add to your design in part C!**

## Part A: Keyboard interface design (50pts)

In this part of the lab, you will be designing an interface for accepting values from the keyboard. In previous labs, we have been limited to accepting inputs from the 16 switches or the 5 buttons. In this lab, we will expand the input functionality by implementing a PS2 keyboard interface. The values sent from the keyboard will be displayed on the seven segment display on the Basys3 board. **Note that although the keyboard you will connect to the Basys3 board is a USB keyboard, there is a microcontroller on the Basys3 board that converts USB to PS2 signals automatically so the FPGA on the board actually will use PS2 inputs.**

The PS2 protocol is a simple two-wire scheme that uses serial transmission to transmit the data to the board. While the two-wire bus is bi-directional in design, we will only be using it as an input to the FPGA. [Typically writing to the keyboard is used to reset, turn on the various indicator lights, etc.]




When a key is pressed, a sequence of bytes is sent serially over the two-wire bus. Each key on the keyboard is given a unique "scancode" (see Basys3 board user manual). In order to detect when keys are initially pressed and then released, the keyboard will send a sequence of bytes for each key press. The first byte sent by the keyboard is typically called the "make code" and it represents the key that is pressed. The final byte sent by the keyboard is the "break code" which represents the key that was released.

For example, consider the situation where a user presses the letter 'a':

1) User presses the 'a' key
2) Keyboard sends "make code" (which is '1C' for the 'a' key) serially. The keyboard keeps sending the "make code" every 100ms until the user releases the key.
3) User releases the 'a' key
4) Keyboard sends the "key up" code 'F0'serially
5) Keyboard sends the "break code" (which is '1C' for the 'a' key) serially

We will only need to look for the "break code" bytes. So we can simply monitor the bits for the "key up" scan code which indicates that the key has been released. When this byte has been sent, the "break code" for the released key will be sent.

To transmit the sequence of bytes, the keyboard first forces the *DATA* line low to create the start bit. Bits are transmitted using the falling edge of *CLK* for synchronization. This is illustrated in Figure 1. The *DATA* signal changes state when the *CLK* signal is high, and *DATA* is valid for reading on the falling edge of *CLK*.

**Note: This *CLK* is not the same clock as the 100MHz oscillator on the board. In your XDC file, you will see two pins with placeholder names *PS2Clk and PS2Data*.**

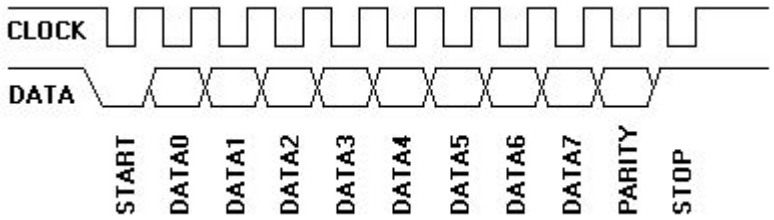



Figure 1. Device to Host Communication Timing for PS2 Inputs

So the basic operation of your design is as follows:

- On the falling edge of the PS2 clock, use a shift register to capture each bit of data
- When all 11 bits have been sent (start, scancode, parity, stop), you can look at the scancode and decide what to do
- If the scancode is a "key up" i.e. 'F0', you know that the next data sequence sent by the keyboard is the final scancode that you need.
- Capture the final scancode by following the same steps as above and output this value from your keyboard controller.

**Initially, the all 7 segment displays should be off.** After the first key is pressed, you will display the two hex digits of the scancode received by the controller on the lower two seven segment displays. **Throughout the demo, the lower two seven segments will change accordingly with key releases and the upper two seven segments will remain off.**

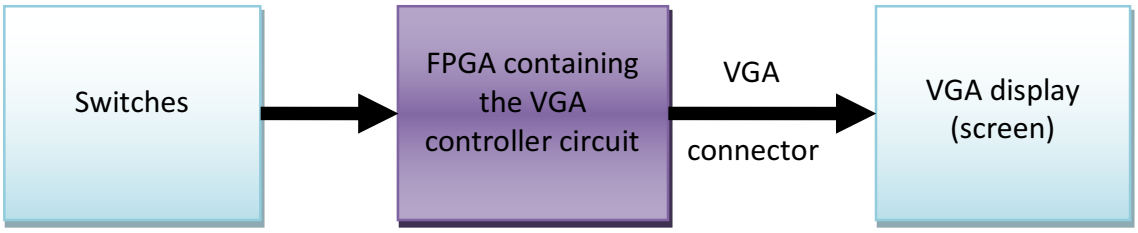
**You should also have a 100ms strobe signal to indicate that the keyboard controller is outputting a key release**. A strobe signal is a short single pulse on one of the board LEDs. **The strobe signal must appear every time a key on the keyboard is physically released, even if it is the same key as the previously released key.**

## Other Important Information

1. There is an easy way of implementing the keyboard interface by using a 22-bit shift register in your design. Think about it! Talk to the TAs about it.
2. The 7-segment display should show the keycode of a key until a new key is pressed then released, at which time it starts to show the keycode of the new key.
3. Some keys on the keyboard (like the arrow keys) are special in the sense that they send an additional code "E0" ahead of the scan code. Such keys are called extended keys. When an extended key is released and "E0 F0" code is sent followed by the scan code. So, irrespective of the type of key, the last two chunks of data when a key is released will be "F0" and the scan code.
4. The Basys3 Board Manual shows the keycode for the "z" key as "1Z". This is a typo; the actual code is "1A".
5. Although the keyboard data signal is birectional, we will only be using it as an input for this lab.
6. There is a certain key that will momentarily display 'F0' on the 7 segments on most students' implementation. If you manage to not get this glitch, then we will overlook one other mistake with your implementation, if any. However, if you get this glitch, please try to understand why it occurs.
7. If you use your own keyboard, please be aware that the Basys3 does not support some keyboards. If you are not getting any responses to the FPGA from your keyboard, please try using the lab keyboards.

## Part B: VGA Interface Design (50pts)

In this part of the lab you will design a VGA controller to output graphics to the computer monitor connected to the Basys3 board. In previous labs, we were limited to either the seven-segment display or the LEDs. In this lab, we expand on this functionality to allow graphical images to be displayed from the FPGA board.




A VGA monitor operates using an electron beam that scans the screen row by row, starting at the upper left corner and ending at the lower-right corner. This beam moves using two synchronization signals, called *hsync* (horizontal synchronization), and *vsync* (vertical synchronization). The *hsync* signal tells the beam when to move to the next row. The *vsync* signal tells the beam when to move to back to the top of the screen. To display a picture on the screen, we simply generate these synchronization signals and provide the pixel color to display on the screen.

**Please follow all the following steps down to the clock cycle. 99% of the time, the FPGA VGA port is not broken as students claim, so simulate your *vsync* and *hsync* to make sure that you do not have ANY discrepancies from the following timing charts. Even having one clock cycle off on your signals can cause the VGA monitor to display the dreaded 'No Input Signal Message'.**

In this lab, you are required to create a 640 pixel x 480 pixel screen display. A pixel clock operating at 25 MHz will be used. To get 640 pixels horizontally, a horizontal synchronization frequency of approximately 31.5 KHz is required. This corresponds to approximately 800 clock periods of the pixel clock. During the first 640 clock periods of the pixel clock for a row, visible pixels can be displayed; however, the last 160 clock periods for the row are called the "retrace period" or "blanking region" where nothing is displayed while the beam retraces back to the left of the screen. To get 480 pixels vertically, a vertical synchronization frequency of 60 Hz is required. This corresponds to approximately (800x)525 clock periods of the pixel clock. This can be thought of as generating 480 visible rows followed by 45 blank rows during which the beam retraces back to the top of the screen. This is illustrated in Fig. 2.

Figure 3 shows the timing of the *hsync* signal. It is made low starting on the 659$^{th}$ pixel clock period for the row and made high again on the 756$^{th}$ pixel clock period. During the first 640 pixel clock periods, visible pixels are generated. During the last 160 pixel clock periods, nothing is generated on the screen.

Figure 4 shows the relationship between the *vsync* signal and the *hsync* signal. The digits represent the line count. As stated earlier, the first 480 lines are displayed while the last 45 lines correspond to the retrace period for the beam to get back to the top left corner of the screen.

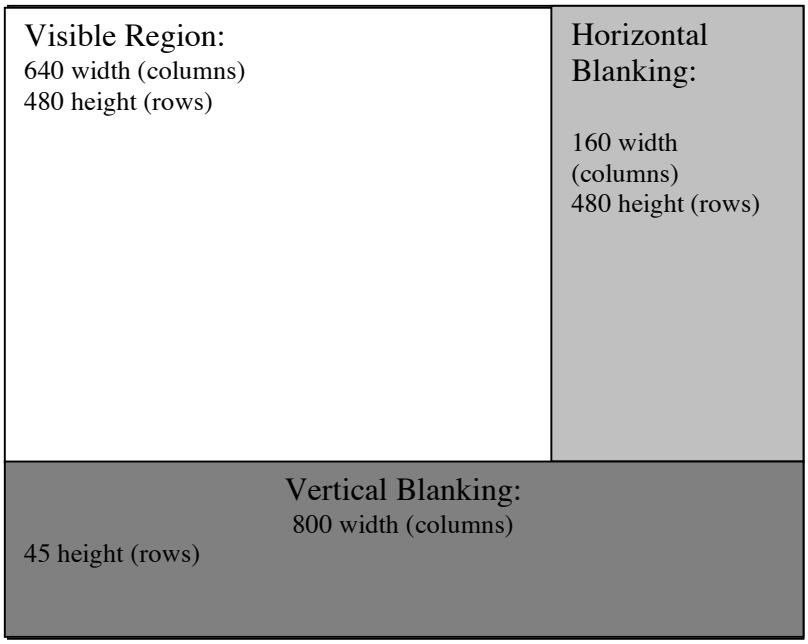



Figure 2. Display Regions

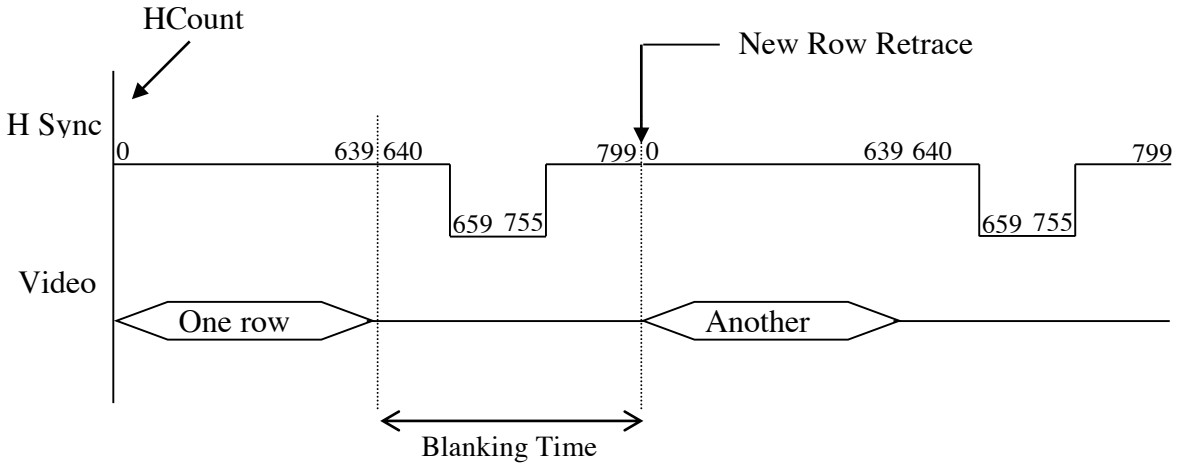



Figure 3. Horizontal Sync Timing

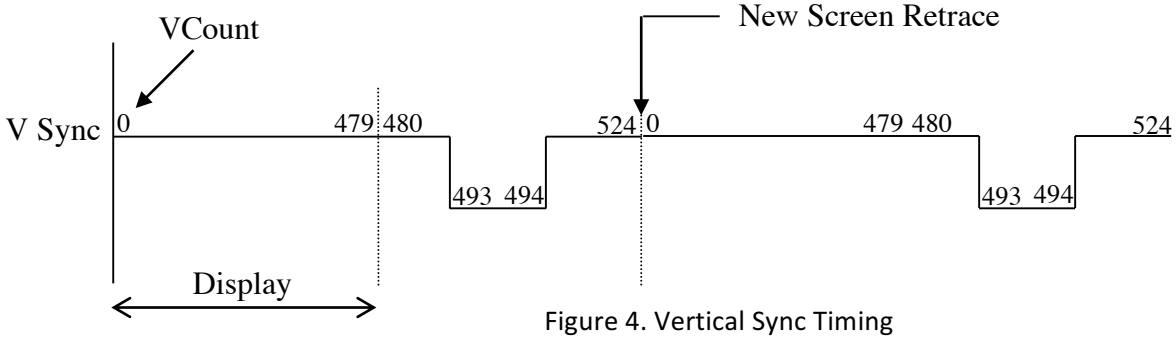



Figure 4. Vertical Sync Timing

So the basic operation of the design you implement is as follows:

- Use 2 counters to store the values of hcount and vcount
- Generate a 25MHz pixel clock by dividing the 100MHz clock
- On the rising edge of the pixel clock, increment the hcount. Increment vcount when hcount has reached the end of the row.
- Generate the hsync signal based on the value of hcount as illustrated in Fig. 3. vsync is generated in a similar fashion as illustrated in Fig. 4.
- Generate a signal to determine whether the pixel is in the visible region as illustrated in Figure 2.
- When in the visible region, output the pixel color value {R, G, B}, otherwise when in the blanking region, output {0, 0, 0}.
- Finally, put all of the outputs {R, G, B, hsync, vsync} thru flip-flops to ensure no combinational logic delays will interfere with the output display

The VGA controller that you design in this part of the lab will take as inputs the 25 MHz pixel clock and the pixel_color to display each clock cycle. The 25Mhz will be generated from the 100Mhz clock and the pixel_color will come from which switch is ON. The following table shows the color of the screen for each switch. The VGA controller will generate as an output the hsync, vsync, R, G, and B signals. The screen will display the color depending on which switch is ON (**complete screen filled with one color**).

**Fall 2016: Use Configuration 1**

| Switch | Configuration 1 | Switch | Configuration 2 |
|---|---|---|---|
| 0 | Black | 0 | Black |
| 1 | Magenta | 1 | Blue |
| 2 | Dark Green | 2 | Brown |
| 3 | Blue | 3 | Cyan |
| 4 | Red | 4 | Red |
| 5 | Orange | 5 | Magenta |
| 6 | Yellow | 6 | Yellow |
| 7 | White | 7 | White |
| None | White | None | Black |

**(Don't consider the cases when more than one switch is ON.)**

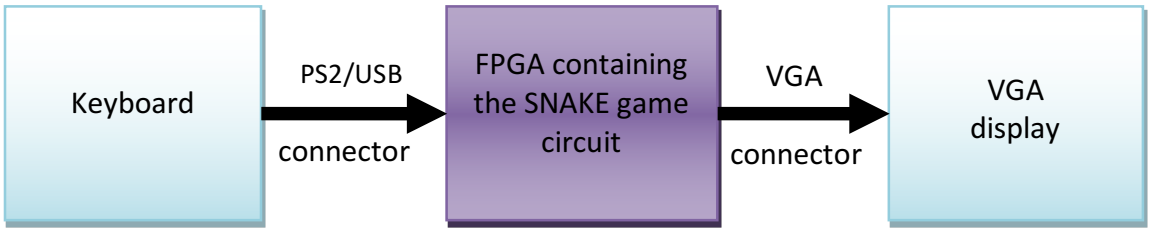
**Important:** It is mandatory to simulate this part before synthesizing and downloading to the FPGA. You can either use a testbench or the commands (like force, etc) directly.

## Useful Information

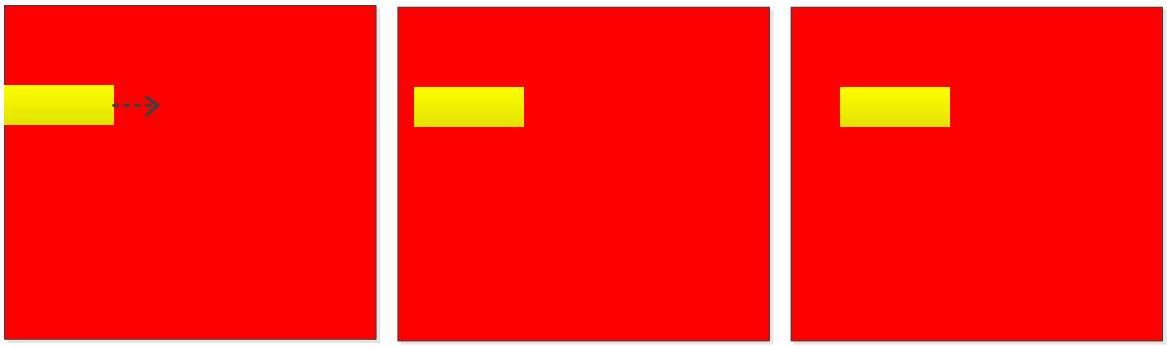
1. You must follow the VGA protocol exactly as mentioned in this document. Do not change the numbers for generating hsync and vsync.
2. To generate different colors, you can refer to the 8-bit VGA color codes where each R, G and B are encoded in 8 bits (total RGB = 24 bits). You can develop color codes (total RGB = 12 bits, which is what you need for this lab) using the following link: http://cloford.com/resources/colours/namedcol.htm

## Part C: Snake (80pts)

Congratulations on finishing part A and B! Now, in this final part of the design, you will implement the master controller which receives input from the keyboard controller and uses the VGA controller to output the appropriate pixels to the monitor. You will implement a moving snake in this part.



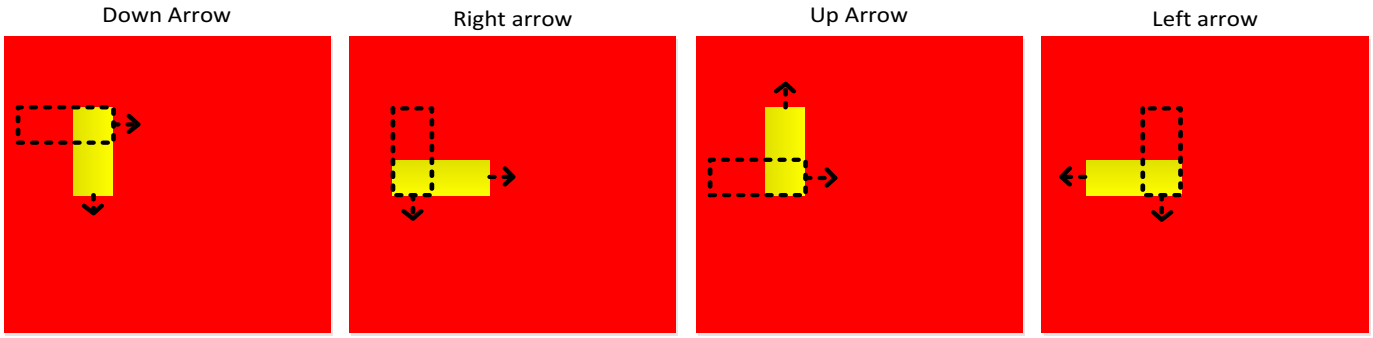

The screen is blank in the beginning. Pressing 'S' on the keyboard starts the snake "game"- a 'snake' graphic at the left edge of the screen that automatically starts scrolling right as shown in the figure below.



This scrolling graphic will respond to arrow key pushes in the following way:

| Original Orientation | Change | |
|---|---|---|
| Horizontal | Up arrow | Flip vertical and scroll up |
| | Down arrow | Flip vertical and scroll down |
| | Left arrow | No change |
| | Right arrow | No change |
| Vertical | Up arrow | No change |
| | Down arrow | No change |
| | Left arrow | Flip horizontal and scroll left |
| | Right arrow | Flip horizontal and scroll right |

The following figures show the change in the graphic due to a few arrow key pushes:




Note that the snake turns from its front head instead of the tail. Pressing the button 'P' on the keyboard pauses the game (freezes the screen) and pressing 'R' resumes the game from its paused state. Pressing 'ESC' exits from the game (blacks out the screen).

Other game requirements:

- The width of the snake, the color of the snake, the background color of the screen and the scrolling speed of the snake that you need to keep are given at the end of this lab description. Note that the snake should scroll smoothly. If the average speed of the snake is 40 (or 50) pixels per second, the snake should not jump 40 (or 50 pixels) every second. In fact, another requirement (also listed at the end of the lab description) is a minimum frame rate. However, if you implement a higher frame rate that the required minimum frame rate, your snake will appear to travel more smoothly.
- On a game over, make sure that only the required part of the snake is visible if dies from turning. For example, if the snake is moving towards the right direction very close to the top edge, and the user presses an up-arrow key, the game will end and only a part of the snake should be visible.
- When the snake touches any edge of the screen, the entire screen should freeze and no longer respond to arrow pushes or R/P presses. However, it should still respond to 'ESC' and 'S' presses.
- Pressing 'R' in the unpaused state does nothing. Pressing 'P' in the paused state does nothing. Pressing 'ESC' **anytime** exits the game (blacks out the screen) and pressing 'S' **anytime** starts the game.
- For sake of convenience, you can choose to disable features from part 1 and 2 as long as you meet the minimum requirements given at the end of this document. The TAs will have you demo each part of Lab 5 during checkout.

A block diagram for an **example** design is shown as follows. You can use this example design, or come up with your own.

**TOP Level Design**

**VGA Controller:**
Inputs: pixel_in, XCoord, YCoord, clk
Outputs: Hsync,Vsync, R, G, B

**Keyboard Controller:**
Inputs: clk, data
Outputs: keycode, newKeyStrobe

**Master Controller**
Takes inputs from keyboard controller, processes the keystrokes to decide what to do and generates the signals that the VGA Controller needs to display the correct color at the correct location.

Inputs: keycode, newKeyStrobe, clk
Outputs: pixel_out, XCoord, YCoord

## Minimum Parameters of the Snake Game

**Fall 2016: Use configuration 1**

**Configuration 1**

| | |
|---|---|
| Background color of the screen | Black |
| Color of the snake | Red |
| Length and width of the snake | 50 x 10 pixels |
| Average Speed of the snake | 40 pixels per second |
| Minimum Framerate | 4 fps |

**Configuration 2**

| | |
|---|---|
| Background color of the screen | White |
| Color of the snake | Blue |
| Length and width of the snake | 40 x 10 pixels |
| Average Speed of the snake | 50 pixels per second |
| Minimum Framerate | 5 fps |

## Creativity Factor (5%)

You will need to make some original modification to your snake game! You have freedom to modify any parameters as long as the original functionality of your game remains intact. You will NOT be compared to other groups; all grades given here are based on the TAs' subjective evaluation of your special feature on a case-by-case basis. Your grade in this portion will be based on the following rubric:

0 out of 9: Did not implement a special feature OR special feature is not original.
Example – Flipping the switches changes the screen color. This is essentially just Lab 5B!

3 out of 9: Special feature is easy to implement.
Example – Option to change game speed.

6 out of 9: Special feature is difficult to implement.
Example – Snake turns smoothly.

9 out of 9: Special feature is creative AND difficult to implement.
Example – The game is like the original snake game with apples and snake growth.

**Please note that these are not four discrete tiers, but rather form a continuous grading spectrum.**