Lab Assignment #10b

Thomas Douglas and Aaditya Watwe

douglas.th@husky.neu.edu watwe.a@husky.neu.edu

Submit date: 12/2/2019

Due Date: 12/4//2019

Abstract

In this lab, we learned to control a VGA display through the ZedBoard using a combination of a simulated circuit on Simulink and a C++ program. Our circuit was responsible for creating the v-sync and h-sync signals for the display, reading the ZedBoard's push buttons and/or switches (depending on the assignment), and pushing values to the VGA pins in the display. Our C++ program used memory locations on the ZedBoard to change integer values representing colors, and ultimately we were able to move a box, and control the color of the display background and the box with a combination of these tools.

Introduction

The goal of the final lab is to design a VGA controller, and use it to change the color and location of objects on the display, through a combination of hardware and software interactions. We learned how to utilize and manipulate Simulink blocks including HDL Counters, logical comparators, bit splitters, etc. These blocks allowed us to read button values, switch values, and memory locations to manipulate the color of the display and a small box which the user may move over it. We were ultimately able to test our circuits utilizing input constants or step blocks and displaying the output values on the scope; however these tests were limited in nature given the complexity of this circuitry. The primary form of testing was via downloading the design to the ZedBoard's FPGA, and using the C++ program to verify that it performs as anticipated. We did this with the System Generator functionality within Xilinx Tools to run the circuit design to the Zedboard and test manually each pushbutton and switch gate's effect on the display. There were no major constraints, yet time was the only challenging component to completing this lab assignment. This lab assumed background knowledge on complex circuit logic elements, experience with basic Simulink circuit (both of which were covered by the pre-lab), some creativity, and basic debugging knowledge skills (which we learned in C++ and applied to Simulink), knowledge of memory offsets in C++, experience with Makefile(s), and knowing how to use sudo.

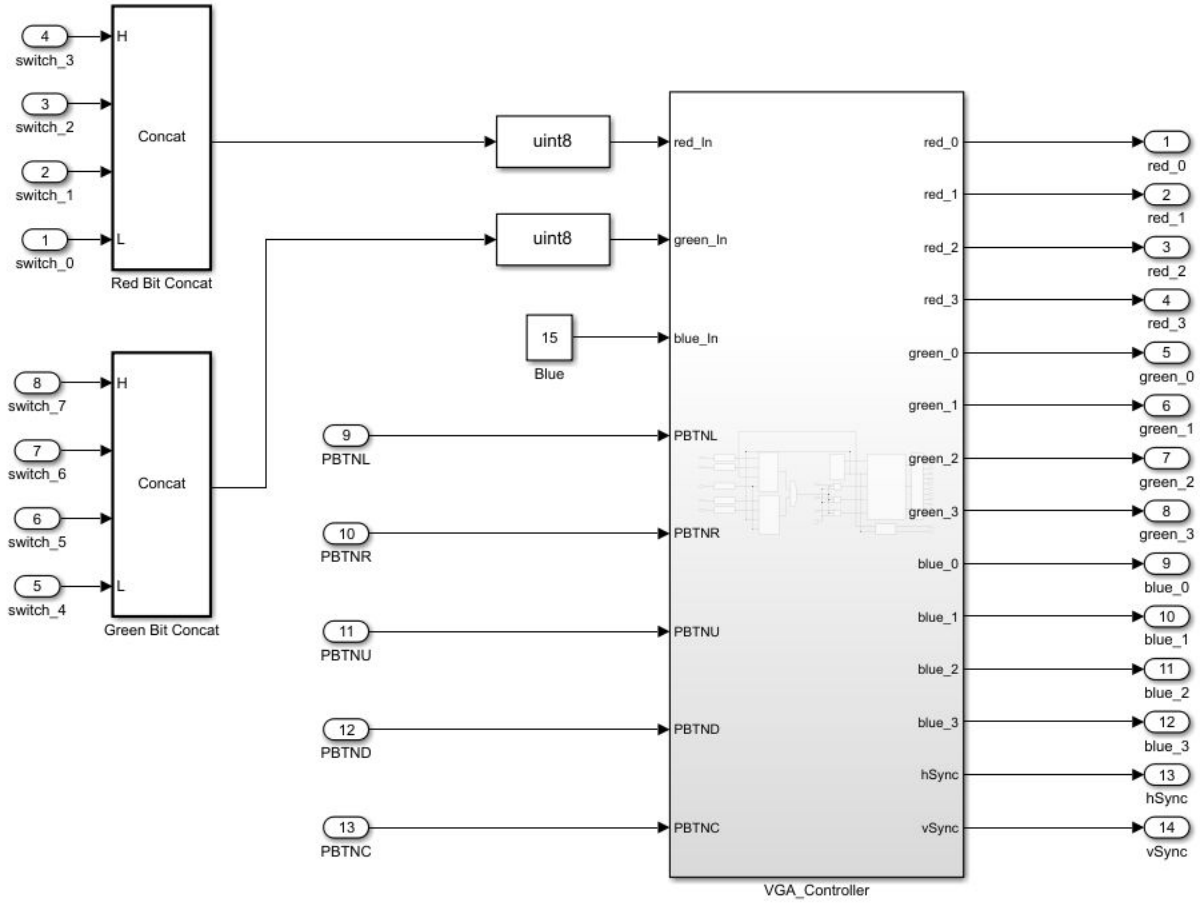
Lab Discussion

In this lab experiment, we used the class computers which contained the MATLAB software where the Simulink functionality was located. MobaXTerm, the C++ language, the gdb debugger, the g++ compiler, and the vim text editor were used. In addition, the Zedboard, a micro-USB cable, power cable for the Zedboard were all used.

Results

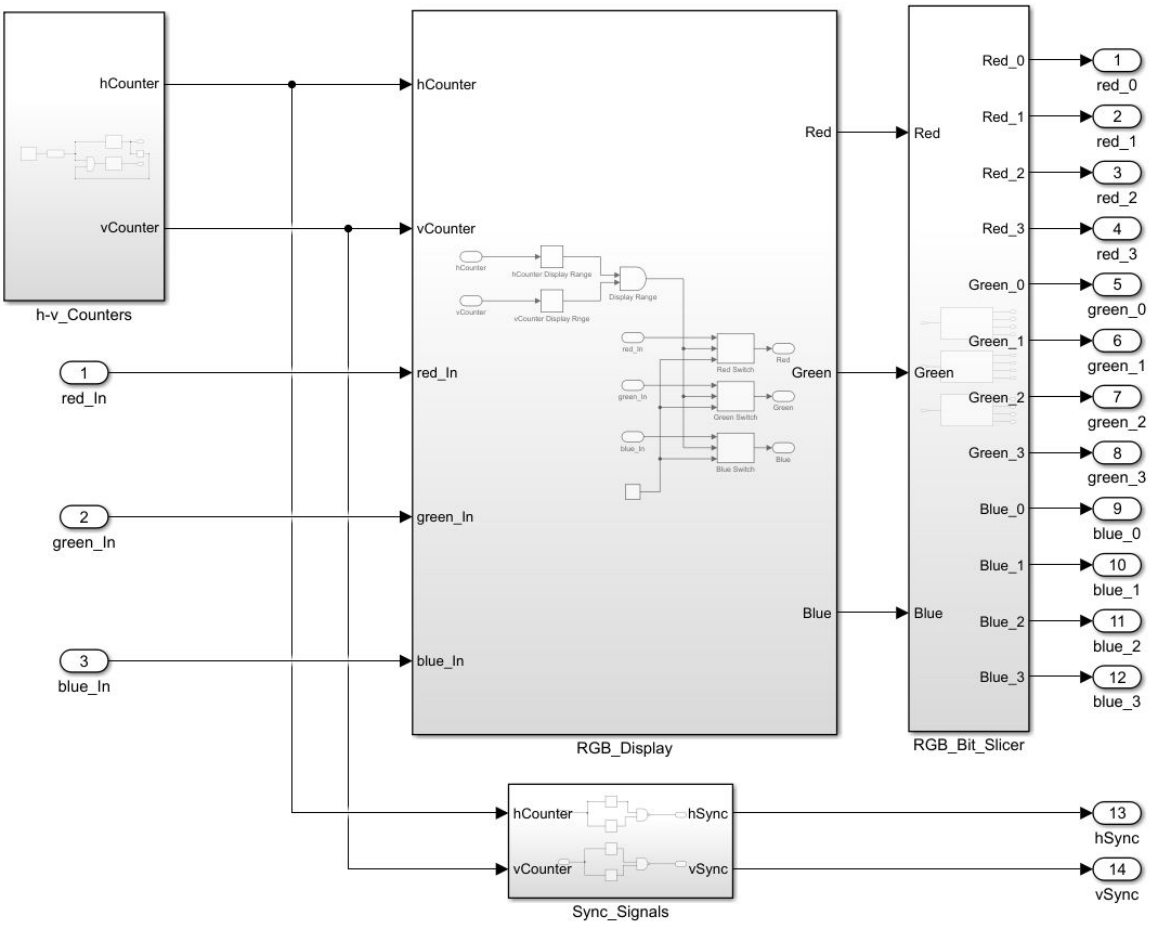
Assignment 1: Controlling the VGA Display Colors with Switches

In the first assignment, we downloaded the Simulink design to the ZedBoard, and checked that the monitor acted as expected. The color of the display, in terms of RGB components, was as follows: $(S_3S_2S_1S_0)_2$ as red, $(S_7S_6S_5S_4)_2$ as green, and 1111_2 as blue. These numbers are 4-bit binary values representing the red, green, and blue intensities. S_x is the value of switch number-x, and the subscript 2 shows that the values are binary. Upon manipulating the switch values to adjust the display's colors, we saw that the proper colors were displayed. With this design, the system displays 2^8 different colors, since there are 8 variables here (4 bits each for red and green), and 2 values for each variable (ON or OFF). This gives us 2^8, or 256 distinct colors. Our simulink design is below with captions explaining the blocks.

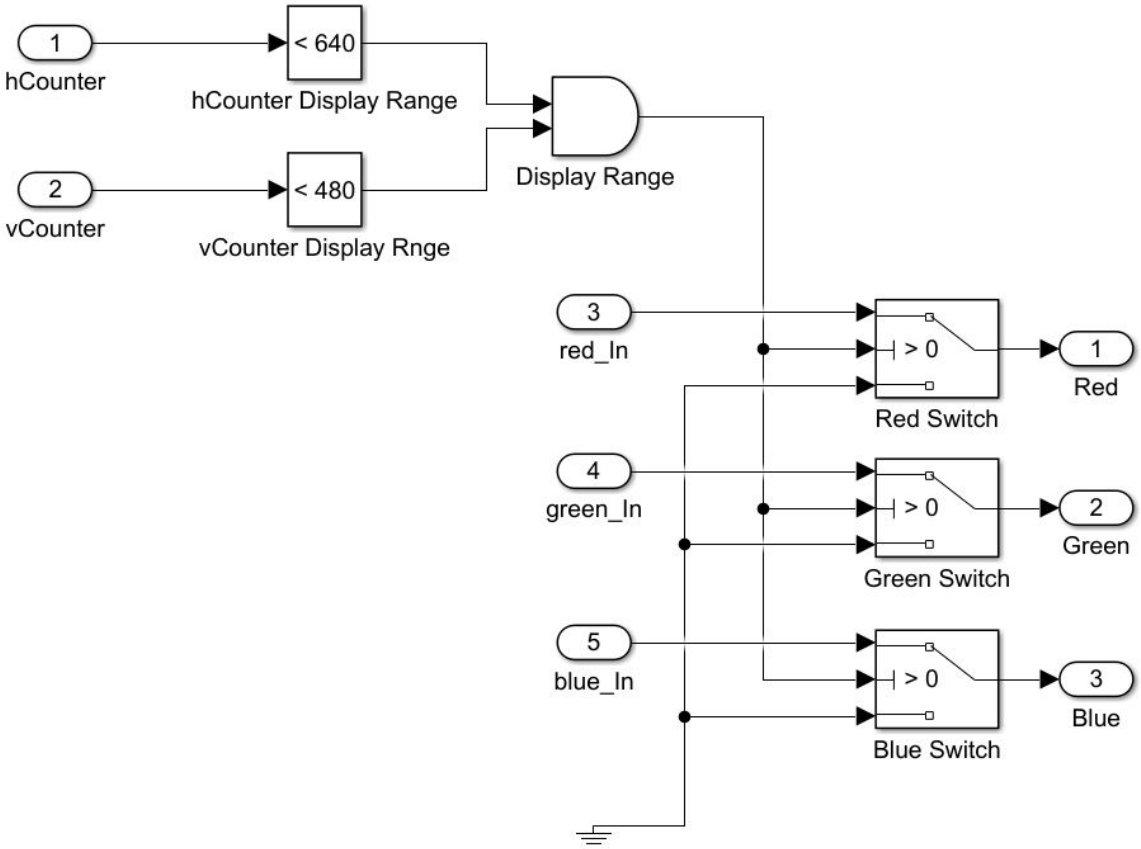


The above screenshot is our complete Simulink design for Assignment 1 named VGA_Interface_Switches.slx. The VGA_Interface_Switches design contains 8 inputs for the specific switches assigned to red and green intensities and 1 constant input assigned to the blue color valued at 15 (1111). Additionally, the design contains 5 inputs for the 5

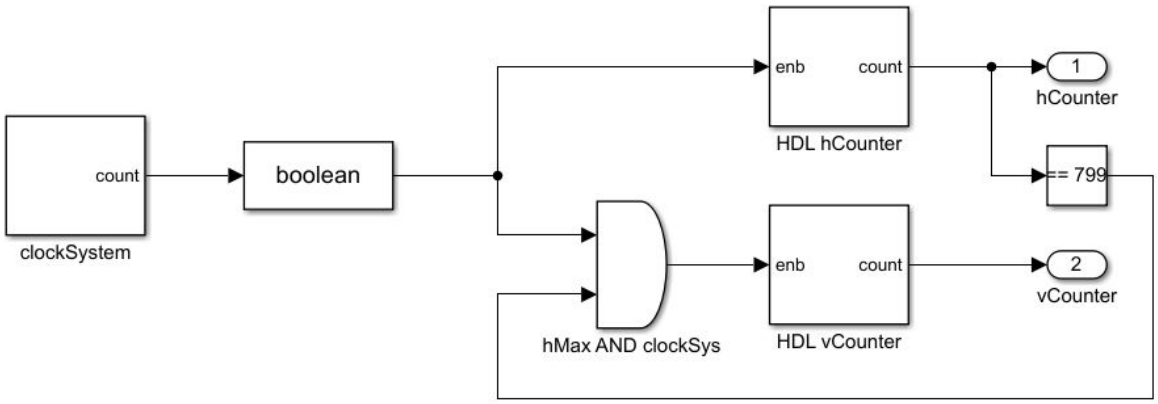
different push buttons on the Zedboard. All of the inputs go through the VGA_Controller subsystem and are split into the 14 pins assigned to the VGA interface.



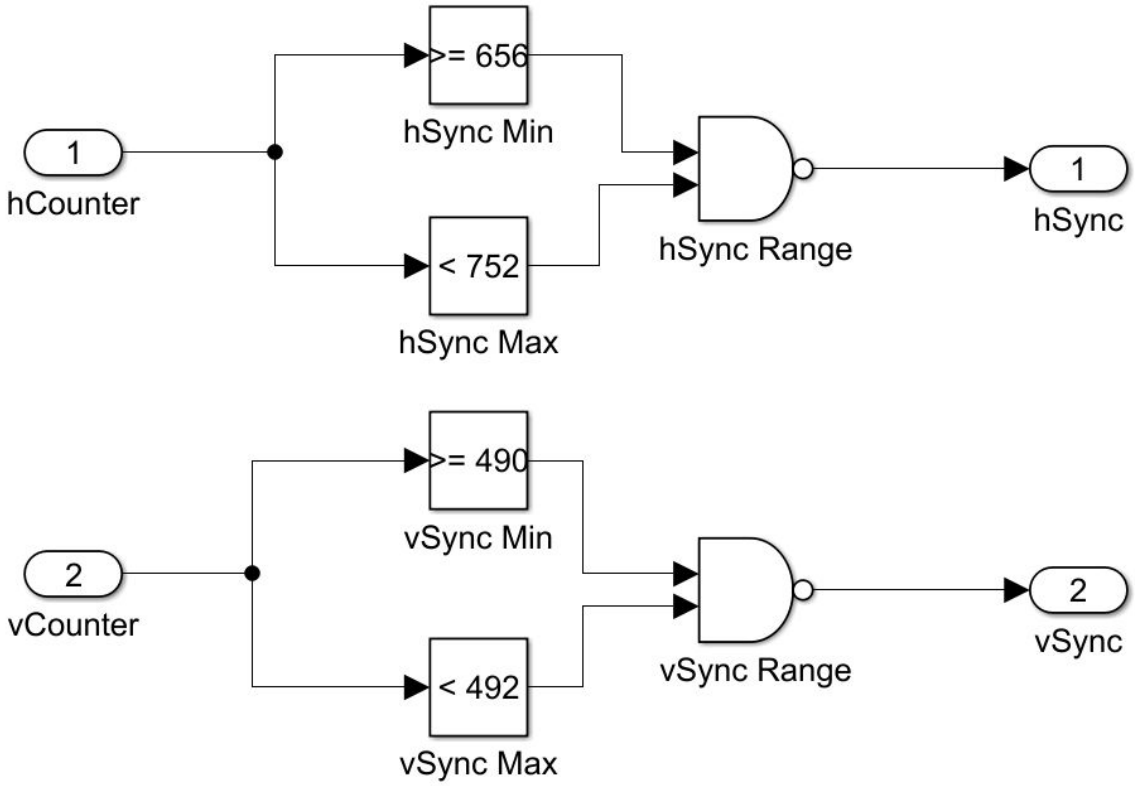
The above screenshot is the VGA_Controller block of our main Simulink design. We have subsystem called h-v_Counters which contains two outputs named hCounter and vCounter which are inputted to their respectively named input ports on our RGB_Display subsystem. The three color inputs for red, green, and blue are connected to the respective ports on our RGB_Display subsystem. The hCounter and vCounter values from the h-v_Counters subsystem is also connected to the input ports of our Sync_Signals subsystems. The outputted values from the Sync_Signals subsystem is connected to pins 13 and 14 assigned to hSync and vSync on the VGA controller. Finally, our RGB_Display outputs three values assigned to the red, green, and blue colors. These values are connected to our RGB_BitSlicer subsystem which splits the colors input their 4 affiliated pins (Red: 1-4, Green: 5-8, Blue: 9-12) on the VGA controller.



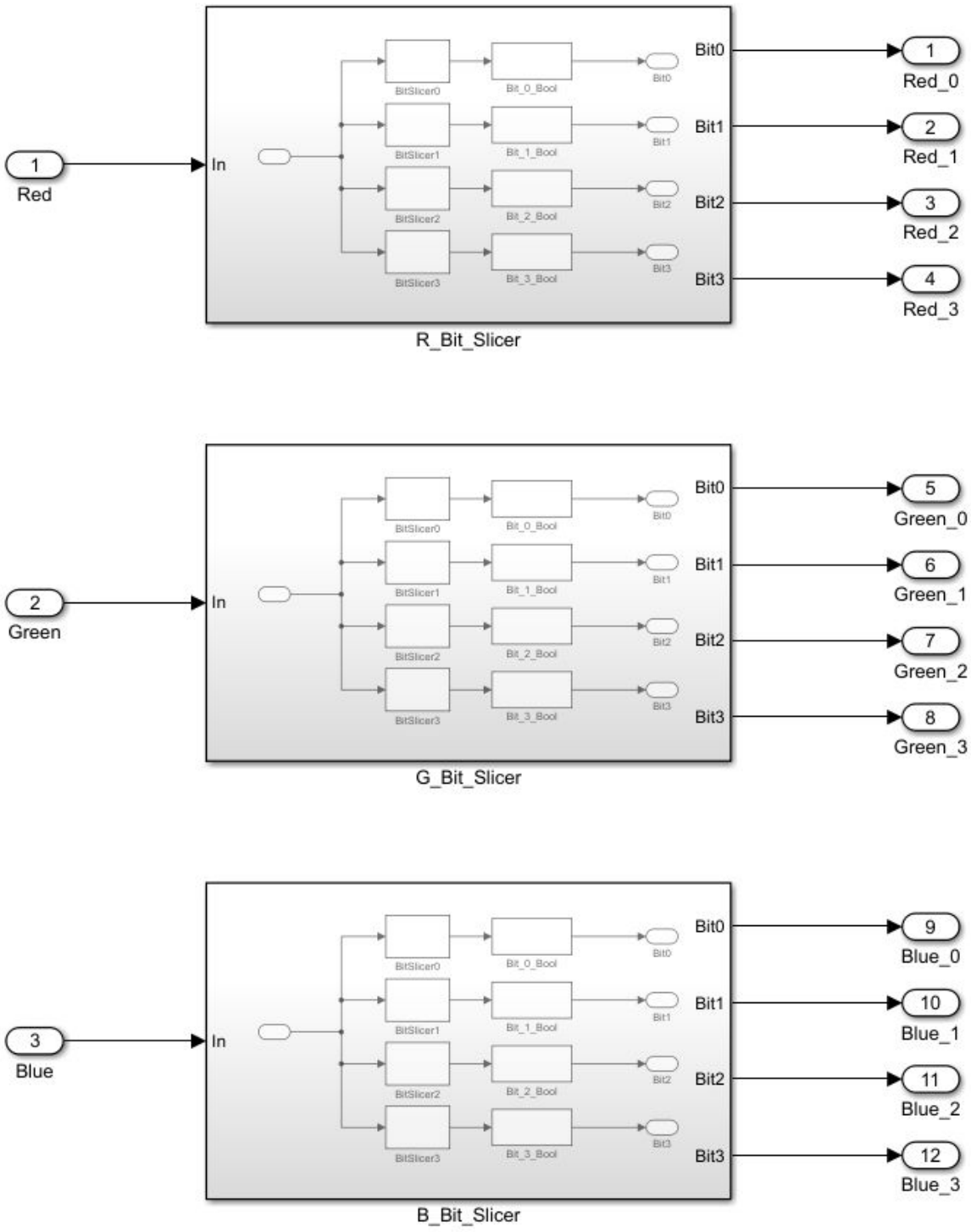
The above screenshot is the VGA_RGB_Display block of our main Simulink design. The subsystem contains 3 inputs for the 3 different colors in which similarly to our VGA_Controller design are inputted into the upper port of their three respective switches. The inputs of the hCounter and vCounter are defined and restricted to their maximum values, 640 for the horizontal max and 480 for the vertical max in order to define the display range. The display range is then inputted into the lower ports on the three color switches. The resultant output from the three switches are connected to the three different color outputs.



The above screenshot is the VGA_h-v_Counter block of our main Simulink design. The design starts with a free-running counter with initial value 1 and step value 1 which we defined as the clockSystem. The clockSystem is connected to a convert to boolean block. This is then connected to the enable ports on both the HDL hCounter and vCounter blocks. The HDL hCounter has a count to value of 799 and the HDL vCounter has a count to value of 524. The outputted count values from each counter is then outputted to their respective output blocks. The vCounter should be enabled only when the hCounter output equals to 799 and the output from the compare to constant block of the clock system is 1.



The above screenshot is the VGA_Sync_Signals block of our main Simulink design. The hCounter and vCounter inputs are sent through comparator blocks in order to define the hSync and vSync ranges. The hSync minimum value needs to be greater than or equal to 656 and the hSync maximum needs to be less than 752. The vSync minimum value needs to be greater than or equal to 490 and the vSync maximum needs to be less than 492. These two ranges are inputted into their respective hSync and vSync output blocks

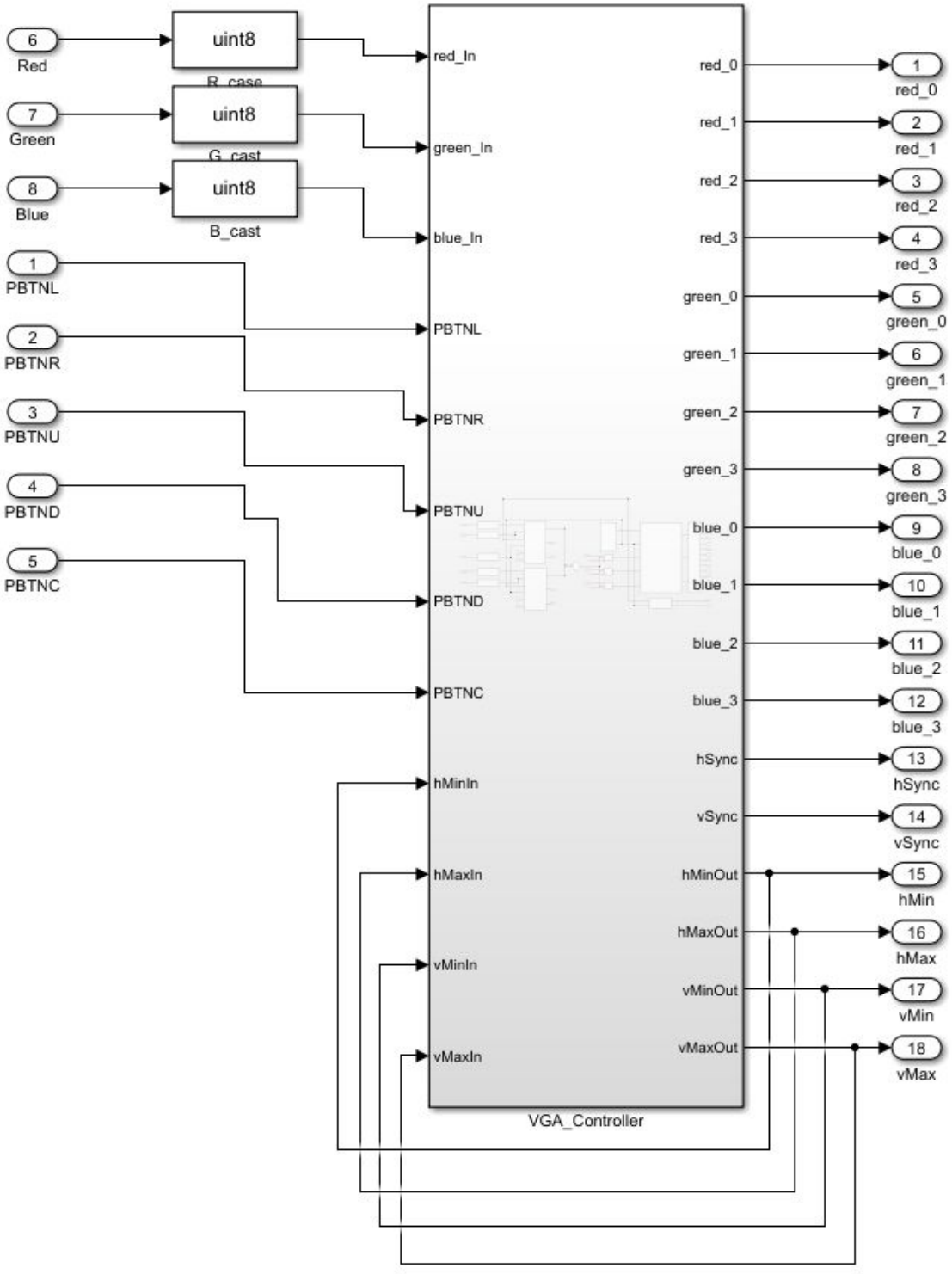


The above screenshot is the VGA_RGB_BitSlicer block of our main Simulink design. The design simply has the three color inputs going into their own respective color bit slicer. These three bit slicer subsystems takes in the input of their color value and converts and

splits the values into 4 bit values which are outputted to their respective pin numbers on the VGA controller.

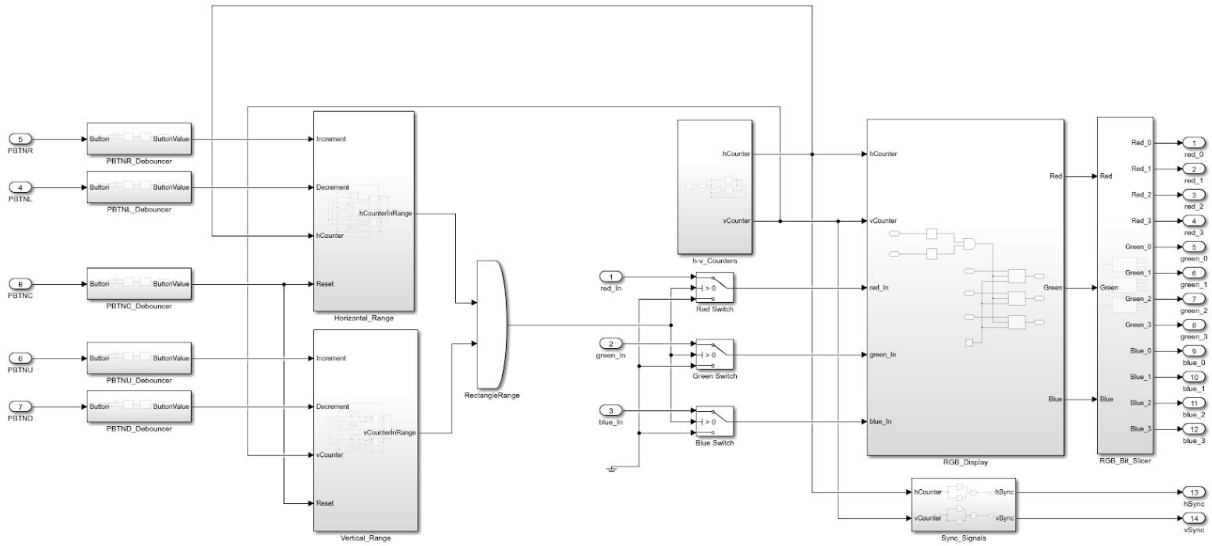
Assignment 2: Controlling the VGA Display Colors with Switches and PushButtons

In the second assignment, we built upon the prior assignment to restrict the VGA color signal to a fixed box interval. We also allowed this box to be moved using push buttons on the ZedBoard. To verify the design works as expected, we downloaded it to the FPGA, and looked at the display. All colors displayed were the ones expected (by using the switches), and the box moved properly using the push buttons. Our design to restrict the box's movement to stay on the screen worked, and the center push button brought the box back to the top left of the screen. Just like our last assignment, there were 256 possible colors using the switches to manipulate red and green (with a fixed blue value). Outside of the box range, the display was black, since we grounded the RGB input outside this range using switches. The sections of our simulink design which were modified are below with captions explaining the blocks present.



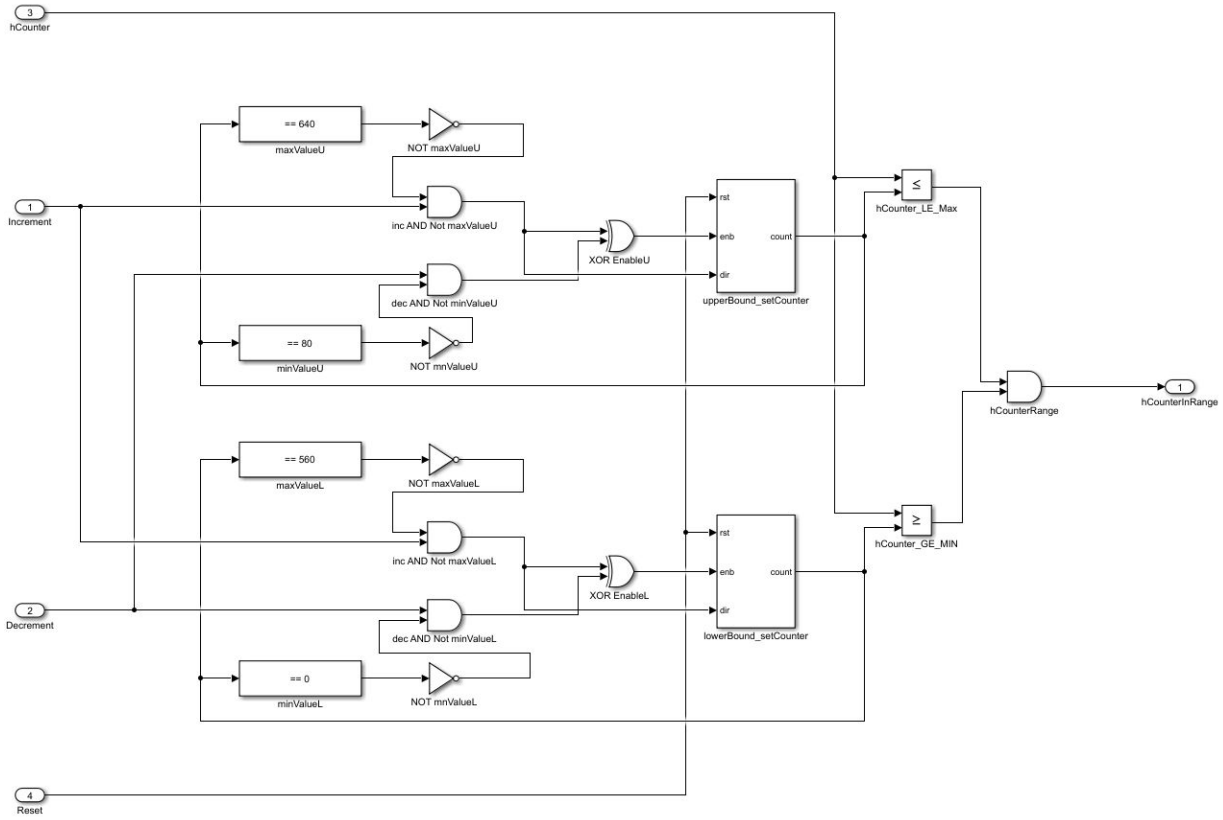
The above screenshot is our complete Simulink design for Assignment 2 named VGA_Interface.slx. Our design contains three color inputs for red, green, and blue which are converted to uint8 values before being inputted to their respective ports on the VGA

controller. Our design also contains 5 inputs for the 5 different push button on the Zedboard which are then connected to their respective inputs on the VGA_Controller. The three colors and 5 push buttons are affiliated with the 14 pins on the VGA interface. Additionally, we have counters for the maximum and minimum horizontal and vertical values which are fed back into the VGA_Controller and additionally outputted to their respective output blocks.



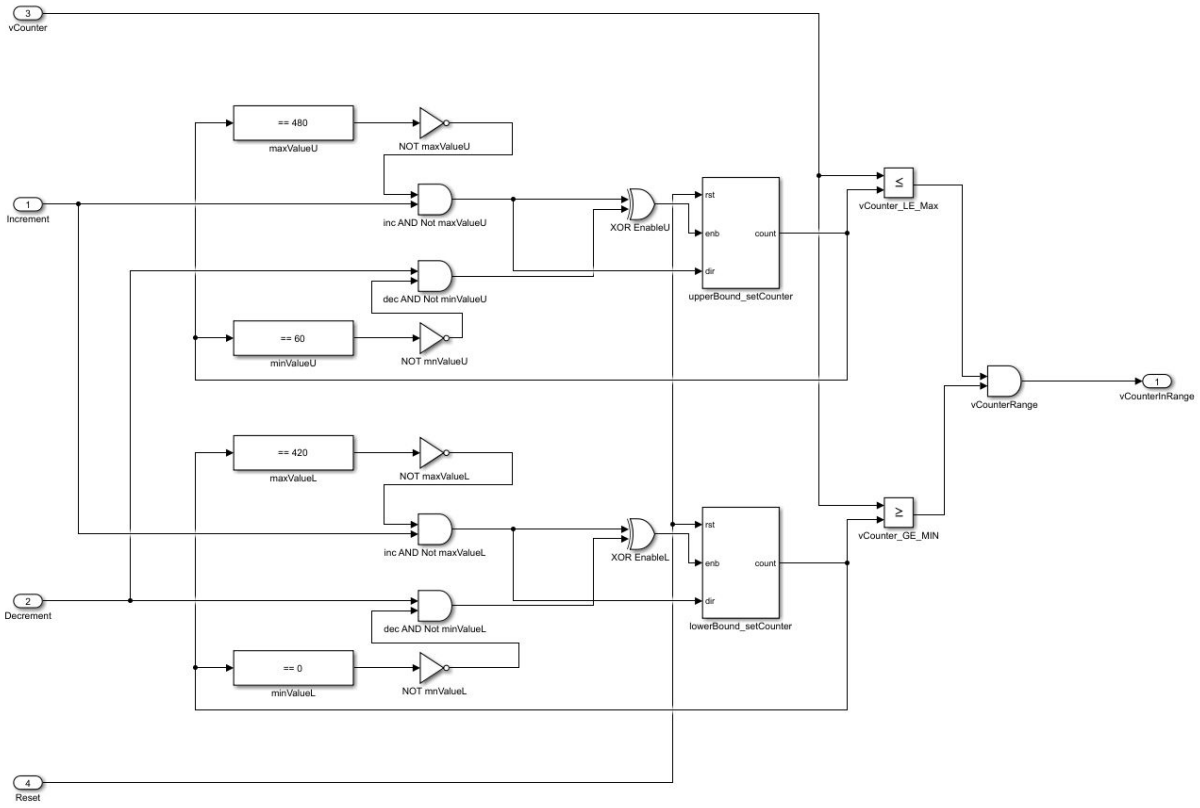
The above screenshot is the VGA_Controller block of our main Simulink design. Our PBTNU and PBTND inputs go through our debouncer subsystem and are ultimately inputted to the increment and decrement ports of our Vertical_Range subsystem. Our PBTNR and PBTNL inputs go through our debouncer subsystem and are ultimately inputted to the increment and decrement ports of our Horizontal_Range subsystem. Our PBTNC input is inputted to both reset ports on our Horizontal_Range and Vertical_Range subsystems. Additionally, we have subsystem called h-v_Counters which contains two outputs named hCounter and vCounter which are inputted to the hCounter and vCounter ports on our Horizontal_Range and Vertical_Range subsystems respectively. Also, the hCounter and vCounter values are inputted to their respectively named input ports on our RGB_Display subsystem. The three color inputs for red, green, and blue are connected to the upper ports of their three respective switches in which those values are true when they are within the rectangle range. The rectangle range is defined by the combination of the outputs from both the Horizontal_Range and Vertical_Range subsystems. The rectangle range value is inputted to the lower port on the red, green, and blue switches. The outputted values from the three color switches are

inputted to their respective ports on our RGB_Display subsystem. The hCounter and vCounter values from the h-v_Counters subsystem is also connected to the input ports of our Sync_Signals subsystems. The outputted values from the Sync_Signals subsystem is connected to pins 13 and 14 assigned to hSync and vSync on the VGA controller. Finally, our RGB_Display outputs three values assigned to the red, green, and blue colors. These values are connected to our RGB_BitSlicer subsystem which splits the colors input their 4 affiliated pins (Red: 1-4, Green: 5-8, Blue: 9-12) on the VGA controller.



The above image is the VGA_Horizontal_Range block of our main Simulink design. To define the horizontal range of the rectangle, we added two HDL Counters and exposed the enable, reset, and direction ports. The first counter keeps track of the minimum coordinate of the rectangle on the horizontal axis, while the second counter keeps track of the maximum coordinate on the horizontal axis. On the first counter, we set the Word length to 10. Also, we set the counter to be Count limited, and Count to value 560 with Initial value equal to 0 and a Step value of 80 (length of the rectangle). On the second counter, we set the Word length to 10. Also, set the counter to be Count limited, and Count to value 640 with Initial value equal to 80 and a Step value of 80 (length of the rectangle). The rectangle's horizontal axis is now defined by setting the horizontal range (set to 1) if the hCounter value is equal to or greater than the minimum coordinate value

(first counter output) and hCounter value is less than the maximum coordinate value (second counter output). The position of the rectangle is controlled by the push buttons which increment/decrement the counters as follows: The rectangle moves to the right or to the left (counter value is incremented or decremented) when the PBTNR or PBTNL are pressed. When PBNTC is pressed, the counters reset, moving the rectangle back to the original coordinates.

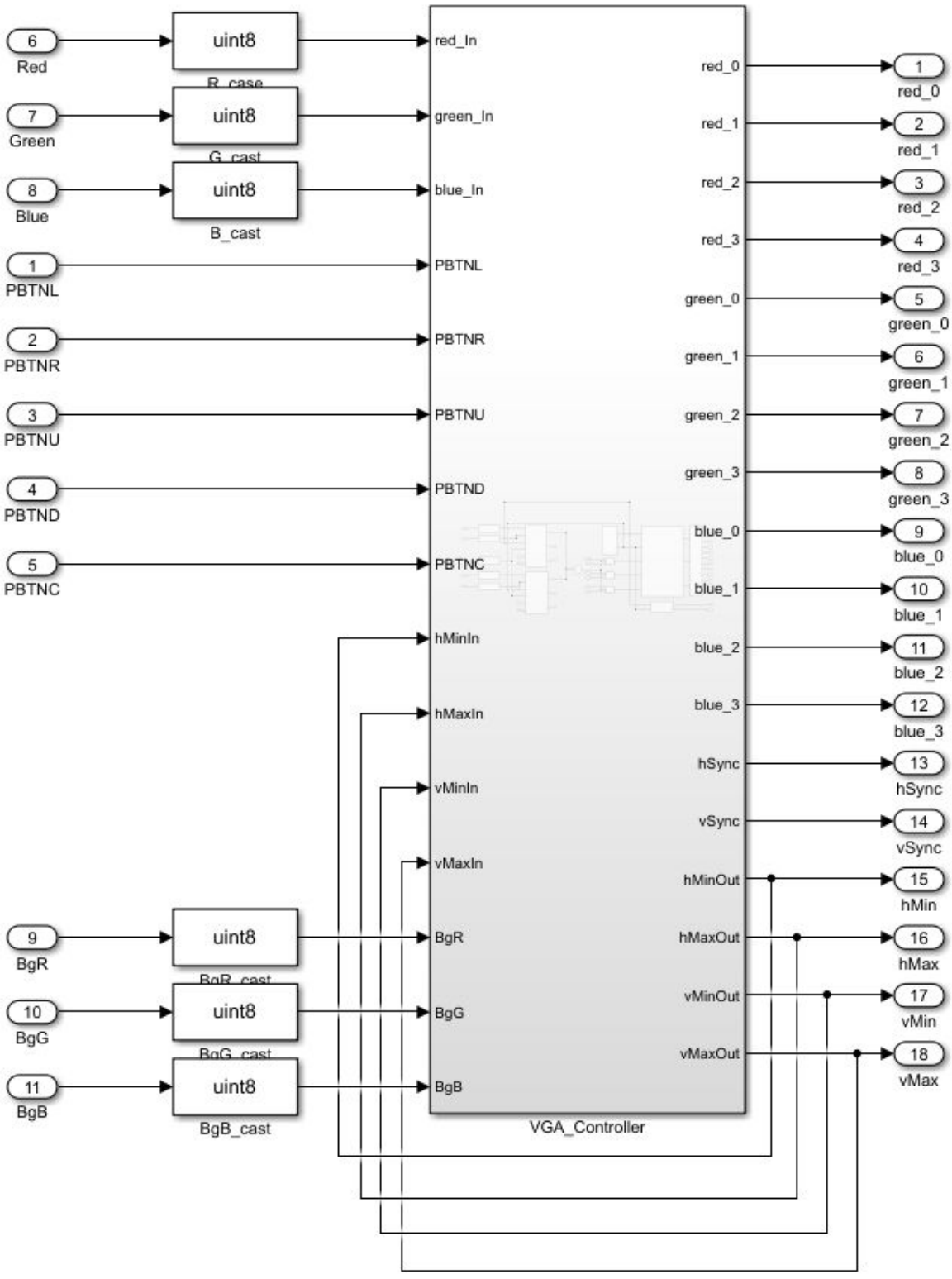


The above image is the VGA_Vertical_Range block of our main Simulink design. The rectangle's vertical axis can be defined in a similar manner. We add the two HDL Counters and expose the enable, reset, and direction ports. The first counter keeps track of the minimum coordinate of the rectangle on the vertical axis, while the second counter keeps track of the maximum coordinate on the vertical axis. On the first counter, we set the Word length to 10. Also, we set the counter to be Count limited, and Count to value 420 with Initial value equal to 0 and a Step value of 60 (width of the rectangle). On the second counter, we set the Word length to 10. Also, we set the counter to be Count limited, and Count to value 480 with Initial value equal to 60 and a Step value of 60 (width of the rectangle). The rectangle's vertical axis can now be defined by setting the vertical range (set to 1) if the vCounter value is equal to or greater than the minimum coordinate value (first counter output) and vCounter value is less than the maximum

coordinate value (second counter output). The vertical position of the rectangle is controlled by the push buttons which increment/decrement the counters as follows. The rectangle moves up the screen or down the screen (counter value is incremented or decremented) when the PBTNU or PBTND are pressed. When PBNTC is pressed, the counters reset, moving the rectangle back to the original coordinates.

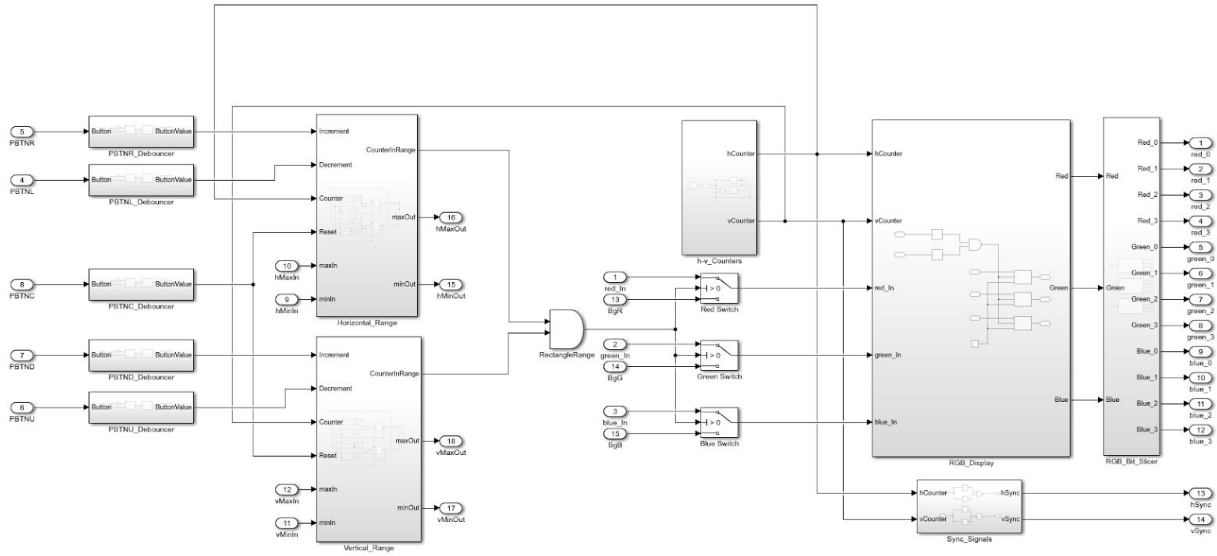
Assignment 3: Controlling the VGA Display With Software

In the final assignment, we modified the last Simulink design (which used switches and push buttons to control the color position of a box) to be controllable via a C++ program. A minor change made here was changing the debouncing period for all buttons, excluding the center button. We drastically reduced the time required to register a button press from 250ms to 3ms. Next, we were supposed to change the coordinates of this box to be taken from `uint16` memory locations instead of the `uint16` counters modified by push buttons. This seemed too simple, so with consent from Professor Kimani, we integrated the `uint16` memory locations with out push-button circuit. Thus instead of only using C++ to control the box position, we wrote the counter's value to the memory, and read that value when displaying the box to the screen, which allowed us to use push buttons to move the box. A demonstration of this is in the video located in our submission folder, labelled `` `Assignment3_Part1.mp4` ``. Additionally, we used `uint8` memory locations to replace the switches for controlling the color of this box. Instead of reading the switches and a constant block to decide which color the box would be, we created a C++ program which lets the user set the box and background colors. The red, green, and blue color inputs for the box and the background were connected to their individual `uint8` inputs, allowing us to change the colors on the fly via the C++ program. We simulated the design to ensure that there were no errors, then we sent it to the ZedBoard via the Workflow Advisor. The box coordinates and RGB values for the background/box were set to the appropriate AXI4-Lite memory locations. After testing the design on the VGA display via the ZedBoard, everything worked as anticipated. The C++ program properly set the RGB values to what we wanted, and the push buttons still functioned properly, as they had in the previous assignment. Our final source code is in the appendix, with screenshots of the terminal output below. Also, the sections of the Simulink design that were modified are below, with screenshots explaining the blocks changed.

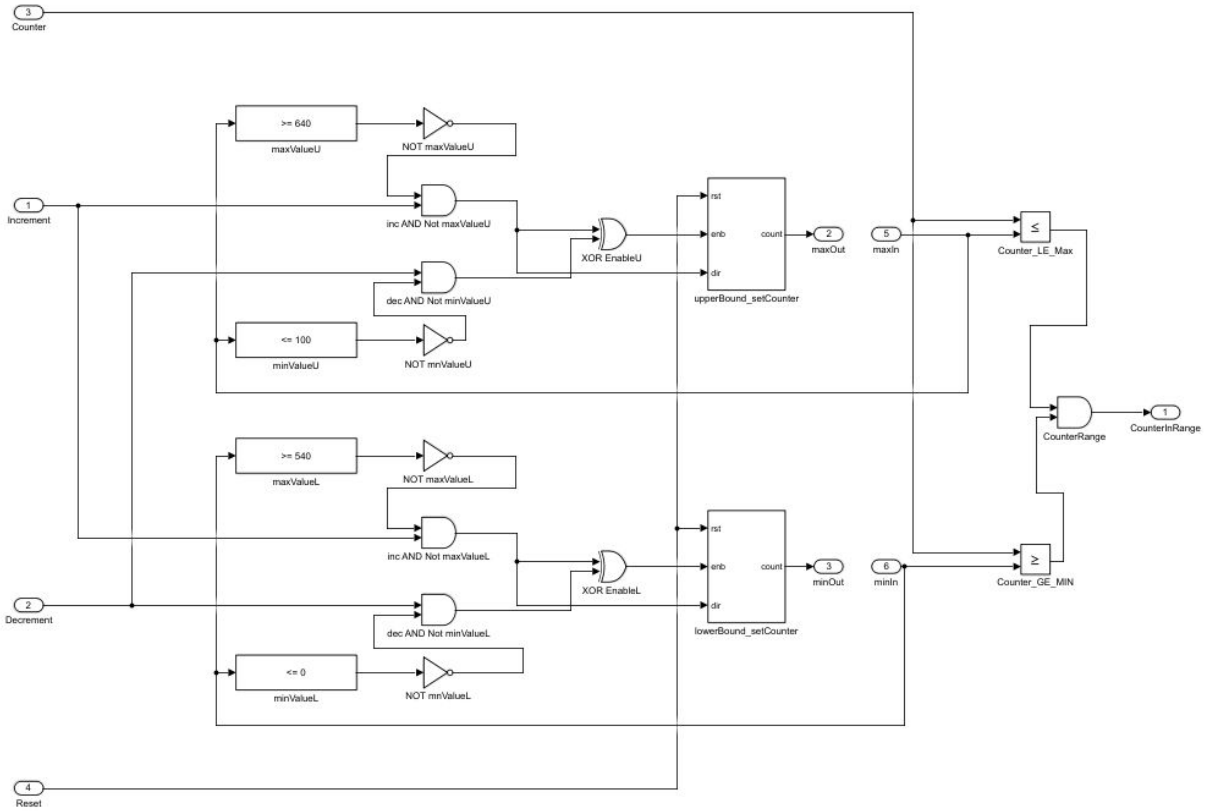


The above screenshot is our complete Simulink design for Assignment 3 named VGA_Interface_Software.slx. The design is this screenshot is very similar to that of

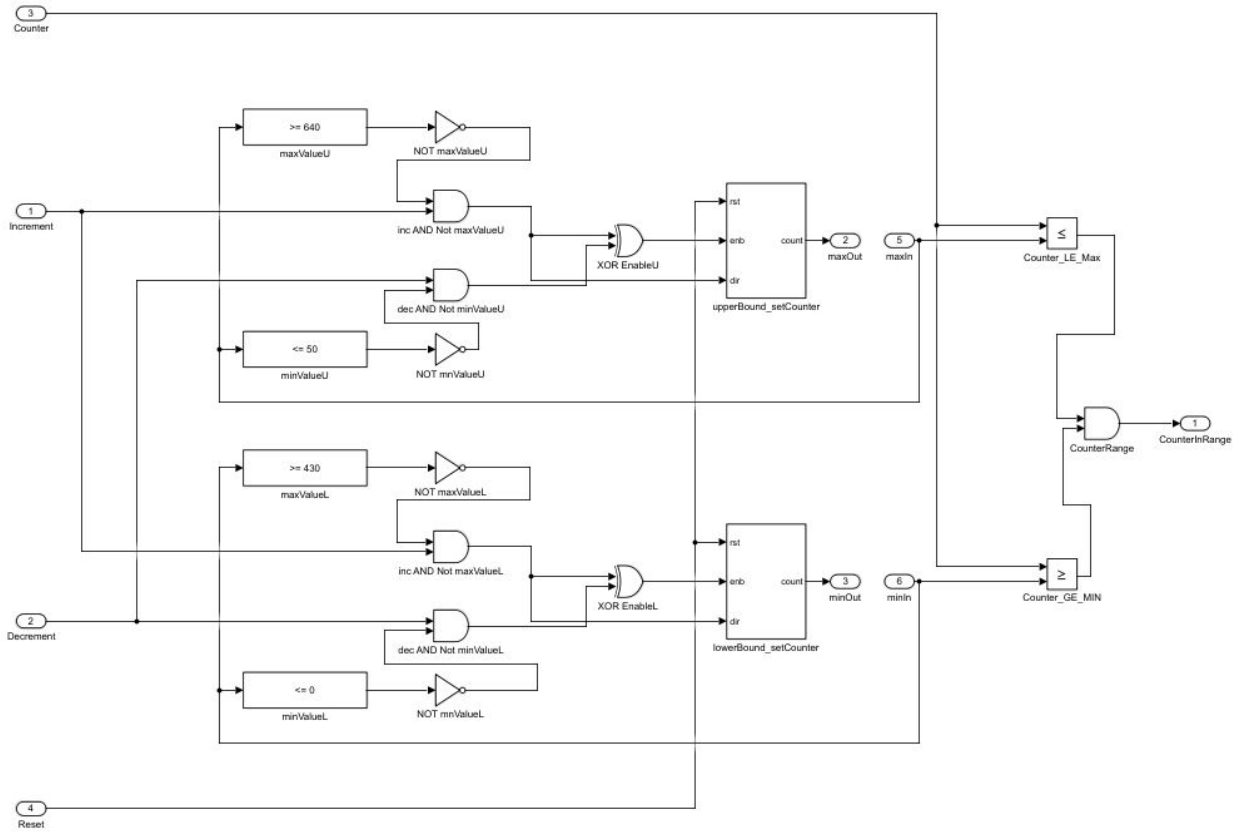
Assignment 2. However, we added three additional input for the red, green, and blue background colors for the background of the VGA display. These inputs are converted to uint8 values and then inputted into their respective ports on the VGA_Controller subsystem.



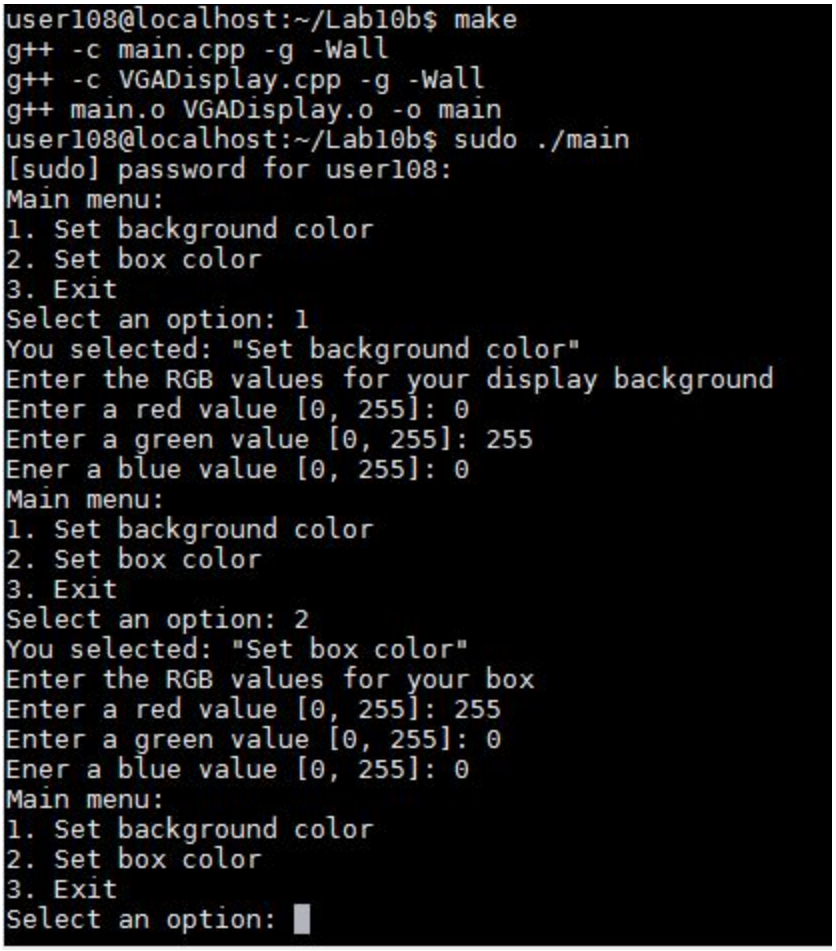
This screenshot shows the modified VGA controller design from the prior assignment. The changes seen are added inputs and outputs, which map to memory locations on the ZedBoard. On the Horizontal_Range subsystem, there are additional inputs of hMaxIn and hMinIn, and the respective outputs hMaxOut and hMinOut. The Vertical_Range subsystem also has its version of these memory locations added: vMinIn, vMaxIn, vMinOut and vMaxOut. The `In` and `Out` memory locations map to the same place allowing the circuit to manipulate the integer at that location, sl well as allowing that memory location to be written to by the C++ program if so desired. Additionally, there are three RGB-color inputs of type uint8: BgR, BgG, and BgB. There inputs replace the ground wiring from the prior assignment so that the C++ program can set the background to something other than black. There are no other major changes in this controller from the previous assignment.



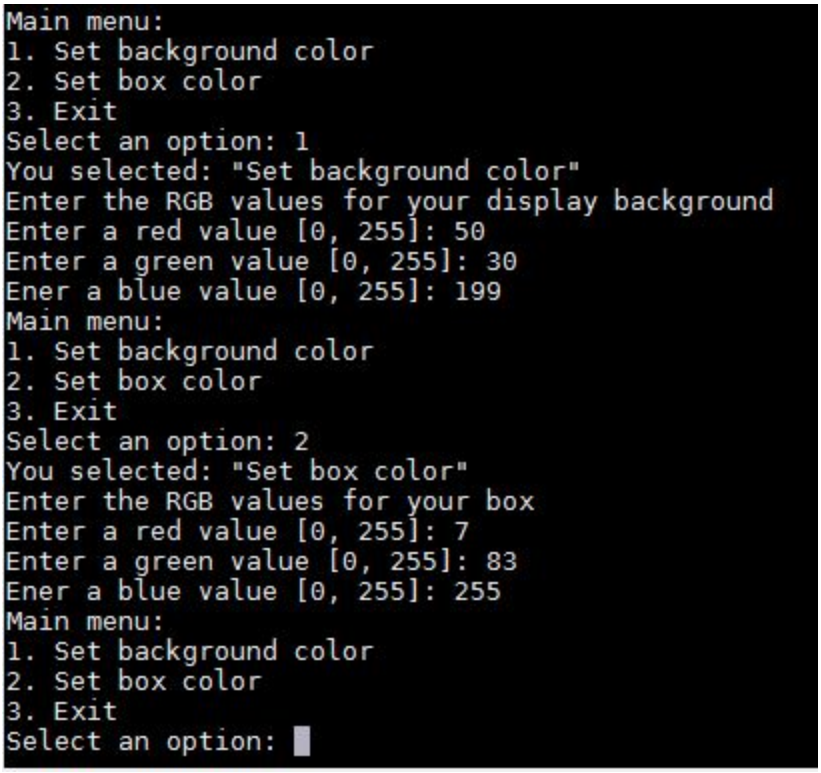
The above image is the VGA_Horizontal_Range block of our main Simulink design. As you can see, the counter's current value is fed into the memory location for the max and min bounds of the box's horizontal range, then that location is read into the Counter_LE_Max and Counter_LE_Min comparators (as well as the circuitry to prevent the counter from exceeding the bounds of the 640x480 display).



The above image is the VGA_Vertical_Range block of our main Simulink design. As you can see, the counter's current value is fed into the memory location for the max and min bounds of the box's vertical range, then that location is read into the Counter_LE_Max and Counter_LE_Min comparators (as well as the circuitry to prevent the counter from exceeding the bounds of the 640x480 display).



The above screenshot is from our MobaXterm shell terminal after running our makefile for the VGA Display files. We created an interface in which the user is prompted to enter the RGB values for either the background color or the box color. Additionally, we included an exit case to escape the program. In this screenshot which is affiliated with the Assignment3_Part1.mp4 video, we set the background color to be green by entering 0 RGB values for red and blue, and the maximum 255 for green. We then set the box color to be red by entering 0 RGB values for green and blue, and the maximum 255 for red. The results are seen in the attached video file called Assignment3_Part1.mp4.



The above screenshot is from the second part of our running program continued from MobaXterm shell terminal after running our makefile for the VGA Display files. In this screenshot which is affiliated with the Assignment3_Part2.mp4 video, we secondly set the background color to now be a shape of light blue by entering 50 for red, 30 for green, and 199 for blue. We then set the box color to a darker blue color by entering 7 for red, 83 for green, and 255 for blue. The results are seen in the attached video file called Assignment3_Part2.mp4

Analysis

All of the results and outputs from the three sections were what we expected them to be. We were able to remove any possible small sources of error during the design process testing constant inputs or step blocks, and displaying its output on a scope. One small error we caught in this way was a swapped increment and decrement wiring of the UP and DOWN push buttons. The UP button is meant to move the box up the screen, thus should decrement the counter value (since (0,0) is at the top left), but testing with constants showed us we had these buttons switched. We had no large errors designing the circuit, since we had these small tests throughout the building process. For all of the errors in C++, the compiler presented them to us. These included missing semicolons, misspelt variables, etc. There were no errors regarding segmentation faults, pointer algebra, etc. We used gdb to ensure the C++ program was writing to the correct memory offsets, and this allowed us to avoid experiencing complicated errors. We had a plan to add functionality to the program by allowing the user to change the size of the box, but due to time constraints between this course and others, we chose not to complete this functionality. This code is still present (as comments) in the appendix, and is highlighted red.

Conclusion

From our results in this experiment, we were able to design a digital circuit that controls a VGA display, and a C++ program that interacts with the simulated circuit to edit the objects on the display. We verified the output of each circuit by using scopes and constant blocks, and visually testing interaction between each push button and switch with the display. We used the constant blocks or step blocks (both converted to boolean datatypes) and scopes to check for minor errors within each subsystem. We implemented our knowledge of logic circuit elements and basic MATLAB skills to execute the tasks in this lab. Then for the final section (coding in C++), we used gdb, and the compiler's error messages to weed out errors. The testing for the final part was done with constant blocks and scopes (to ensure the Simulink design did not break), and visual testing with the display to ensure the memory mapped integer values for the background and the box got changed correctly. There was no difference between the experimental and theoretical results of the lab, as there was no empirical analysis of the results.

Appendix

1) *VGADisplay.h*

```
#ifndef VGA_DISPLAY_H
#define VGA_DISPLAY_H

// Physical base address of GPIO
      const unsigned gpio_address = 0x400d0000;

// Length of memory-mapped IO window
      const unsigned gpio_size = 0xff;
      const int red_color_offset = 0x100; // Offset for the box's red
      const int green_color_offset = 0x104; // Offset for box's green
      const int blue_color_offset = 0x108; // Offset for the box's blue
      const int bg_red_color_offset = 0x11c; // Offset for the background's red
      const int bg_green_color_offset = 0x120; // Offset for background's green
      const int bg_blue_color_offset = 0x124; // Offset for the background's blue
      const int h_min_offset = 0x10c; // Offset for x_min coordinate
      const int h_max_offset = 0x110; // Offset for x_max coordinate
      const int v_min_offset = 0x114; // Offset for v_min coordinate
      const int v_max_offset = 0x118; // Offset for v_max coordinate
/*    const int h_size_offset = 0x128; // Offset for box horizontal size
      const int v_size_offset = 0x12C; // Offset for box vetical size
*/

class VGADisplay
{
      // File descriptor for memory-mapped I/O
      int fd;

      // Mapped address
      char *pBase;

      // Write a value into the given memory offset in the memory-mapped I/O.
      void RegisterWrite(unsigned offset, unsigned value);

      // Read a value from the given memory offset in the memory-mapped I/O.
      int RegisterRead (unsigned offset);

      public:
      // Class constructor
      VGADisplay();
```

```
	// Destructor
	~VGADisplay();

	/*
	 * Writes RGB colors to the VGA Controller
	 * - Uses base address of I/O
	 * @param red Red color value
	 * @param green Green color value
	 * @param blue Blue color value
	 */
	void RGB_Write(int red, int green, int blue);

	/*
	 * Writes RGB colors to the background of the display
	 * - Uses base address of I/O
	 * @param red Red color value
	 * @param green Green color value
	 * @param blue Blue color value
	 */
	void Background_RGB_Write(int red, int green, int blue);

	/*
	 * Writes the values for the box's horizontal and
	 * vertical size
	 * - Uses base adresses of I/O
	 * @param h Box's horizontal size value
	 * @param v Box's vertical size value
	 */
//	void Size_Write(int h, int v);
};

#endif
```

2) VGADisplay.cpp

```
#include "VGADisplay.h"
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <iostream>

/**
 * Constructor Initialize general-purpose I/O
 * - Opens access to physical memory /dev/mem
 * - Maps memory at offset 'gpio_address' into virtual address space
 *
 * @param None Default constructor does not need arguments.
 * @return None Default constructor does not return anything.
 */
VGADisplay::VGADisplay()
{
	// Open memory mapped I/O
	fd = open("/dev/mem", O_RDWR);

	// Map physical memory
	pBase = (char *) mmap(NULL, gpio_size, PROT_READ | PROT_WRITE,
	MAP_SHARED, fd, gpio_address);

	// Check success
	if (pBase == (void *) -1)
	{
		std::cerr << "Error mapping memory - fogot sudo?\n";
		exit(1);
	}

	// Initial display colors
	RGB_Write(15, 15, 15);
	sleep(3); // Delay for 3 seconds
}

/**
 * Destructor to close general-purpose I/O.
 * - Uses virtual address where I/O was mapped.
 * - Uses file descriptor previously returned by 'open'.
```

```
 *
 * @param None Destructor does not need arguments.
 * @return None Destructor does not return anything.
 */
VGADisplay::~VGADisplay()
{
	munmap(pBase, 0xff); // Unmap physical memory
	close(fd); // Close memory mapped I/O
}

/**
 * Write a 4-byte value at the specified general-purpose I/O location.
 *
 * - Uses base address returned by 'mmap'.
 * @parem offset Offset where device is mapped.
 * @param value Value to be written.
 */
void VGADisplay::RegisterWrite(unsigned offset, unsigned value)
{
	* (volatile unsigned *) (pBase + offset) = value;
}

/**
 * Read a 4-byte value from the specified general-purpose I/O location.
 *
 * - Uses base address returned by 'mmap'.
 * @param offset Offset where device is mapped.
 * @return Value read.
 */
int VGADisplay::RegisterRead (unsigned offset)
{
	return * (volatile unsigned *) (pBase + offset);
}

/**
 * Writes RGB colors to the VGA Controller
 *
 * - Uses base address of I/O
 * @param red Red color value
 * @param green Green color value
 * @param blue Blue color value
 */
```

```
void VGADisplay::RGB_Write(int red, int green, int blue)
{
	// Set the box colors
	RegisterWrite(red_color_offset, red);
	RegisterWrite(green_color_offset, green);
	RegisterWrite(blue_color_offset, blue);
}

/**
 * Writes RGB colors to the background on the display
 *
 * - Uses base address of I/O
 * @param red Red color value
 * @param green Green color value
 * @param blue Blue color value
 */
void VGADisplay::Background_RGB_Write(int red, int green, int blue)
{
	// Set the background colors
	RegisterWrite(bg_red_color_offset, red);
	RegisterWrite(bg_green_color_offset, green);
	RegisterWrite(bg_blue_color_offset, blue);
}

/**
 * Writes the values for the box's horizontal and
 * vertical size
 * - Uses base adresses of I/O
 * @param h Box's horizontal size value
 * @param v Box's vertical size value
 */ /*
void VGADisplay::Size_Write(int h, int v)
{
	// Set the box range
	RegisterWrite(h_size_offset, h);
	RegisterWrite(v_size_offset, v);
}
*/
```

3) main.cpp

```
#include "VGADisplay.h"
#include <unistd.h>
#include <iostream>
using namespace std;

// Prompt the user to enter RGB values into the given pointer locations with error
checking
void promptColor(int* r, int* g, int* b)
{
	// Prompt for the colors
	cout << "Enter a red value [0, 255]: ";
	if(!(cin >> *r)) {
		cerr << "ERR: Invalid red value" << endl;
	} else {
		cout << "Enter a green value [0, 255]: ";
		if(!(cin >> *g)) {
			cerr << "ERR: Invalid green value" << endl;
		} else {
			cout << "Ener a blue value [0, 255]: ";
			if(!(cin >> *b)) {
				cerr << "ERR: Invalid blue value" << endl;
			} else {
				return;
			}
		}
	}
}

/*
// Prompt the user to enter the box's size woth error checking
void promptSize(int* h, int* v)
{
	// Prompt for the size
	cout << "Enter a horizontal value [0, 640]: ";
	if(!(cin >> *h)) {
		cerr << "ERR: Invalid horizontal value" << endl;
	} else if(*h > 640 || *h < 0) {
		cerr << "ERR: Horizontal value out of range" << endl;
	} else {
		cout << "Enter a vertical value [0, 480]: ";
		if(!(cin >> *v)) {
```

```
                  cerr << "ERR: Invalid vertical value" << endl;
            } else if(*v > 640 || *v < 0) {
                  cerr << "ERR: Vertical value out of range" << endl;
            } else {
                  return;
            }
      }
}
*/

int main()
{
      VGADisplay display;
      display.Background_RGB_Write(0,0,255);
//    display.Size_Write(100, 50);

      display.RGB_Write(255, 0, 0);

      string menu =
            "Main menu:\n"
            "1. Set background color\n"
            "2. Set box color\n"
//          "3. Set box dimensions\n"
            "3. Exit\n"
            "Select an option: ";
      string choice1 = "\"Set background color\"";
      string choice2 = "\"Set box color\"";
//    string choice3 = "\"Set box dimensions\"";
      string choice3 = "\"Exit\"";

      string option_str;
      int option;
      cout << menu;

      bool selected_exit = false;
      while(!(selected_exit)) {

            if(!(cin >> option)) {
                  cin.clear();
                  cin.ignore(10000, '\n');
                  cerr << "ERR1: Invalid option entered. Please select from the menu." <<
endl;
```

```
		}
		else {
			cout << "You selected: ";
		}

		switch (option) {
			case 1:
			{
				cout << choice1 << endl;
				// Prompt the user to enter a background color
				int r, g, b;
				cout << "Enter the RGB values for your display background" <<
endl;
				promptColor(&r, &g, &b);
				display.Background_RGB_Write(r, g, b);
				break;
			}
			case 2:
			{
				cout << choice2 << endl;
				// Prompt the user to enter a box color
				int r, g, b;
				cout << "Enter the RGB values for your box" << endl;
				promptColor(&r, &g, &b);
				display.RGB_Write(r, g, b);
				break;
			}
/*			case 3:
			{
				cout << choice3 << endl;
				// Prompt the user for sizes
				int h, v;
				cout << "Enter the box's dimensions" << endl;
				promptSize(&h, &v);
				display.Size_Write(h, v);
				break;
			}
*/			case 3:
			{
				cout << choice3 << endl;
				selected_exit = true;
				break;
```

```
                }
                default:
                {
                        cerr << "ERR: Invalid option. Please select from the menu." <<
endl;
                        break;
                }
        }

        if(!selected_exit) {
                cout << menu;
        }
    }
}
```

4) Makefile

```
main: main.o VGADisplay.o
      g++ main.o VGADisplay.o -o main

main.o: main.cpp
      g++ -c main.cpp -g -Wall

VGADisplay.o: VGADisplay.cpp
      g++ -c VGADisplay.cpp -g -Wall

clean:
      rm main main.o VGADisplay.o
```